Pong Code Breakdown

Every piece of code necessary to run the pong game is included in the pong.py file. This includes the Player, Ball, and Game classes, as well as other code necessary to run and display the game, and report on the game’s state.

Import Statements

import pygame  
import numpy as np  
import math

These statements import the Python modules used in creating the Pong game.

**Pygame** is a library for creating video games. It provides functionality for handling graphics and user input.

**Numpy** is a library for numerical computing in Python. It is used here to generate random numbers for ball movement angles. Importing numpy as “np” serves as an alias so the code can be shorter.

**Math** is a module that provides mathematical functions and constants. It is used for calculating angles and trigonometric functions in the ball's movement.

Constant Variables

# Constants  
SCREEN\_WIDTH = 640  
SCREEN\_HEIGHT = 480  
PADDLE\_WIDTH = 10  
PADDLE\_HEIGHT = 60  
BALL\_RADIUS = 5  
PADDLE\_SPEED = 3  
BALL\_SPEED = 5  
BALL\_SPEED\_MAX = 5  
WALL\_PADDING = 20  
WHITE = (255, 255, 255)  
BLACK = (0, 0, 0)

These variables will always remain constant and are therefore labeled as such in the game’s code. Many of these variables are used in more than one place, so they exist for ease of use by the programmer. This design practice adheres to the principle of "Don't Repeat Yourself" (DRY). Only one value has to be changed by the programmer to change that value everywhere it is used in the program. This allows for ease of maintenance and for more flexibility.

**SCREEN\_WIDTH and SCREEN\_HEIGHT:** These variables represent the dimensions of the game window in pixels. They define how wide and high the window will be, determining the game's resolution.

**PADDLE\_WIDTH and PADDLE\_HEIGHT:** These variables define the size of the paddles in pixels. They determine the width and height of the player-controlled paddles in the game.

**BALL\_RADIUS:** This variable sets the size of the ball's radius in pixels. The ball's radius determines the ball's overall size, which is used for collision detection and drawing the ball on the screen.

**PADDLE\_SPEED:** This variable represents the number of pixels a paddle will move up or down in a single frame when the player moves the paddle.

**BALL\_SPEED:** This variable is used to calculate the distance the ball will travel in each frame when it is moving. It defines the ball's base speed, determining how fast the ball moves across the screen.

**BALL\_SPEED\_MAX:** This variable is used to calculate the reduction factor used in handling collision with the paddles. It limits the maximum speed that the ball can reach when it bounces off a paddle, preventing the ball from moving too fast during intense gameplay.

**WALL\_PADDING:** This variable represents the number of pixels by which the paddles are offset from the outside edges of the screen. It creates a gap between the paddles and the screen edges, ensuring the ball does not collide with the paddles when they are at the edge of the window.

**WHITE and BLACK:** These variables set the color values used for drawing the game window. They represent the RGB color values for white and black, respectively. In this game, white is used to draw the paddles, ball, and scores on the black background

Player Class

# Player class  
class Player:  
 def \_\_init\_\_(self, x):  
    self.x = x  
    self.y = SCREEN\_HEIGHT // 2 - PADDLE\_HEIGHT // 2  
    self.score = 0  
  
 def move\_up(self):  
     if self.y > 0:  
         self.y -= PADDLE\_SPEED  
  
 def move\_down(self):  
     if self.y < SCREEN\_HEIGHT - PADDLE\_HEIGHT:  
         self.y += PADDLE\_SPEED  
  
 def update(self, ball):  
     # Track the height of the ball  
         if self.y + PADDLE\_HEIGHT // 2 < ball.y:  
         self.move\_down()  
     elif self.y + PADDLE\_HEIGHT // 2 > ball.y:  
         self.move\_up()  
  
 def score\_point(self):  
     self.score += 1

The player class represents a player in the Pong game. It has several functions which each define a behavior of the player or an action the player can take.

**\_\_init\_\_(self, x):** This is the constructor method of the Player class. It initializes a player object with the provided x coordinate. It sets the initial x position using its given x value and the initial y position using the PADDLE\_WIDTH and PADDLE\_HEIGHT constants to set it in the middle. The initial score of the player is set to 0.

**move\_up(self):** This method is called when the player wants to move the paddle upward. It checks if the paddle is not at the top of the screen. If it is not, it allows the paddle to move up.

**move\_down(self):** This method is similar to the move\_up(self) method. This method is called when the player wants to move the paddle downward. It checks if the paddle is not at the bottom of the screen. If it is not, it allows the paddle to move down.

**update(self, ball):** This method is responsible for updating the position of the player's paddle based on the position of the ball. This is used only for the “computer” player. It tells the paddle to move up or down depending on the y position of the ball by calling the move\_up(self) or move\_down(self) method. If the ball is above the paddle, the paddle moves up. If the ball is below the paddle, the paddle moves down.

**score\_point(self):** This method is called when the player scores a point. It increases the score attribute of the player by 1.

Ball Class

# Ball class  
 class Ball:  
 def \_\_init\_\_(self):  
     self.x = SCREEN\_WIDTH // 2  
     self.y = SCREEN\_HEIGHT // 2  
     self.speed = BALL\_SPEED  
     self.velocity\_x = 0  # Ball's velocity in the x-axis  
     self.velocity\_y = 0  # Ball's velocity in the y-axis  
     self.set\_random\_angle()  
  
 def set\_random\_angle(self):  
     angle = np.random.uniform(-math.pi / 4, math.pi / 4)  
     self.velocity\_x = self.speed \* math.cos(angle) \* np.random.choice([-1, 1])  
     self.velocity\_y = self.speed \* math.sin(angle) \* np.random.choice([-0.5, 0.5])  
  
 def draw(self, screen):  
     pygame.draw.circle(screen, WHITE, (self.x, self.y), BALL\_RADIUS)  
  
 def update(self, player1, player2):  
 self.x += self.velocity\_x  
 self.y += self.velocity\_y  
  
 # Collision with paddles  
 if (  
         player1.x <= self.x + WALL\_PADDING + BALL\_RADIUS <= player1.x + PADDLE\_WIDTH and  
         player1.y <= self.y <= player1.y + PADDLE\_HEIGHT  
 ):  
     self.velocity\_x \*= -1  
  
     middle\_y = player1.y + PADDLE\_HEIGHT / 2  
     difference\_in\_y = middle\_y - self.y  
     reduction\_factor = (PADDLE\_HEIGHT / 2) / BALL\_SPEED\_MAX  
     velocity\_y = difference\_in\_y / reduction\_factor  
     self.velocity\_y = -1 \* velocity\_y

elif (  
         player2.x <= self.x - WALL\_PADDING - BALL\_RADIUS <= player2.x + PADDLE\_WIDTH and  
         player2.y <= self.y <= player2.y + PADDLE\_HEIGHT  
 ):  
     self.velocity\_x \*= -1  
       
     middle\_y = player2.y + PADDLE\_HEIGHT / 2  
     difference\_in\_y = middle\_y - self.y  
     reduction\_factor = (PADDLE\_HEIGHT / 2) / BALL\_SPEED\_MAX  
     velocity\_y = difference\_in\_y / reduction\_factor  
     self.velocity\_y = -1 \* velocity\_y

    # Collision with walls  
 if self.y <= 0 or self.y >= SCREEN\_HEIGHT:  
     self.velocity\_y = -self.velocity\_y  
  
 # Scoring  
 if self.x < 0:  
     player1.score\_point()  
     self.\_\_init\_\_()  # Reset ball position  
     self.speed = BALL\_SPEED  # Reset ball speed  
     player1.y = SCREEN\_HEIGHT // 2 - PADDLE\_HEIGHT // 2  
     player2.y = SCREEN\_HEIGHT // 2 - PADDLE\_HEIGHT // 2  
 elif self.x > SCREEN\_WIDTH:  
     player2.score\_point()  
     self.\_\_init\_\_()  # Reset ball position  
     self.speed = BALL\_SPEED  # Reset ball speed  
     player1.y = SCREEN\_HEIGHT // 2 - PADDLE\_HEIGHT // 2  
     player2.y = SCREEN\_HEIGHT // 2 - PADDLE\_HEIGHT // 2

The ball class represents the ball. Like the player class, it has several functions that define its characteristics and behavior.

Initialization

**\_\_init\_\_(self):** This method is the constructor of the Ball class. It initializes the attributes of the ball, such as its initial position (x and y coordinates), speed, and velocities (velocity\_x and velocity\_y). Additionally, it calls the set\_random\_angle() method, which sets a random angle for the ball's serve.

Serving

**set\_random\_angle(self):** This method is responsible for determining the random angle at which the ball starts moving. It generates a random angle within a predefined range. It then takes this angle and uses it along with its speed value to calculate the ball’s x and y velocity. The random values add an element of variability to the game so that no two serves are exactly the same.

Draw Method

**draw(self, screen):** This method is used to visually represent the ball on the game screen. It utilizes the pygame.draw.circle() function to draw a circle with the specified x and y coordinates of the ball's center. The circle is drawn using a specified color (in this case, WHITE) and has a radius defined by the BALL\_RADIUS constant.

Update Method

**update(self, player1, player2):** This method is responsible for updating the status of the ball. The method first moves the ball by adding the x and y velocity values of the ball to the ball’s position, moving the ball left or right, and up or down. The class then checks for the three scenarios of the ball colliding with a paddle, colliding with a wall, and scoring a point.

**Paddle Collision:** This part checks for collision with either of the paddles. In short, if the ball’s position overlaps with the paddle’s position, they are in collision. The method then calculates the angle at which the ball should bounce off the paddle. This is done by determining first the y value of the middle of the paddle (middle\_y) then the difference between the ball’s y value and the calculated middle y value (difference\_in\_y). Then a reduction factor is calculated by dividing half the paddle’s height by the maximum ball speed. Finally, the ball’s new y velocity is calculated by dividing the difference\_in\_y value by the reduction factor. This value multiplied by -1 is set as the ball’s new y velocity.

**Wall Collision:** This checks for collision with either the top or bottom wall. If the ball’s y position is the same as the top or bottom of the screen, its y velocity is reversed (through multiplying it by -1).

**Scoring:** This checks if the ball’s x position exceeds the left or right edge of the screen. When it does, it adds a point to the opposing side’s player. It then sets up the next serve using the \_\_init\_\_() function and sets the ball’s speed value back to its original value. Finally, it resets the position of each paddle back to their starting position.

GameInformation Class

class GameInformation:  
 def \_\_init\_\_(self, player2\_hits, player1\_hits, player2\_score, player1\_score):  
         self.player2\_hits = player2\_hits  
     self.player1\_hits = player1\_hits  
     self.player2\_score = player2\_score  
     self.player1\_score = player1\_score

The GameInformation class stores player data for the current game. Specifically, it stores each player’s number of hits and points scored. They are stored in this way to make it easier to retrieve these values externally.

Game Class

# Game class  
 class Game:  
 def \_\_init\_\_(self):  
     self.running = False  
     self.screen = None  
     self.clock = None  
     self.player1 = None  
     self.player2 = None  
     self.ball = None  
  
     self.player2\_hits = 0  
     self.player1\_hits = 0  
     self.player2\_score = 0  
     self.player1\_score = 0  
  
 def initialize(self):  
     pygame.init()  
     self.screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))  
     pygame.display.set\_caption("Pong")  
     self.clock = pygame.time.Clock()  
     self.player1 = Player(SCREEN\_WIDTH - PADDLE\_WIDTH)  
     self.player2 = Player(0)  
     self.ball = Ball()  
     self.running = True  
  
 def handle\_events(self):  
     for event in pygame.event.get():  
         if event.type == pygame.QUIT:  
             self.running = False  
  
     keys = pygame.key.get\_pressed()  
  
     if keys[pygame.K\_UP]:  
         self.player1.move\_up()  
     elif keys[pygame.K\_DOWN]:  
         self.player1.move\_down()  
  
     if keys[pygame.K\_w]:  
         self.player2.move\_up()  
     elif keys[pygame.K\_s]:  
         self.player2.move\_down()  
  
 def update(self):  
     # self.player2.update(self.ball)  # player2 automatically tracks ball's position and moves accordingly  
     self.ball.update(self.player1, self.player2)  
  
 def draw(self):  
     self.screen.fill(BLACK)  
     dash\_length = 10  # Length of each dash segment  
     gap\_length = 10  # Length of each gap segment  
  
     # Draw the dashed line  
     y = 0  
     while y < SCREEN\_HEIGHT:  
         pygame.draw.line(  
             self.screen, WHITE, (SCREEN\_WIDTH // 2, y), (SCREEN\_WIDTH // 2, y + dash\_length), 1  
         )  
         y += dash\_length + gap\_length  
  
     # Draw the paddles with separation from the edges  
     pygame.draw.rect(self.screen, WHITE,  
                     (self.player1.x - WALL\_PADDING, self.player1.y, PADDLE\_WIDTH, PADDLE\_HEIGHT))  
     pygame.draw.rect(self.screen, WHITE,  
                      (self.player2.x + WALL\_PADDING, self.player2.y, PADDLE\_WIDTH, PADDLE\_HEIGHT))  
  
     self.ball.draw(self.screen)  
  
     font = pygame.font.Font(None, 36)  
     score1\_text = font.render(str(self.player1.score), True, WHITE)  
     score2\_text = font.render(str(self.player2.score), True, WHITE)  
     score1\_rect = score1\_text.get\_rect(center=(SCREEN\_WIDTH // 2 + 50, 30))  
     score2\_rect = score2\_text.get\_rect(center=(SCREEN\_WIDTH // 2 - 50, 30))  
     self.screen.blit(score1\_text, score1\_rect)  
     self.screen.blit(score2\_text, score2\_rect)  
  
     pygame.display.flip()  
  
 def run(self):  
     self.initialize()  
  
     while self.running:  
         ball\_x\_velocity = self.ball.velocity\_x  
         player1\_points = self.player1.score  
         player2\_points = self.player2.score  
  
         self.handle\_events()  
         self.update()  
         self.draw()  
         self.clock.tick(60)  
  
         if ball\_x\_velocity > 0 > self.ball.velocity\_x:  
             self.player1\_hits += 1  
         elif self.ball.velocity\_x > 0 > ball\_x\_velocity:  
             self.player2\_hits += 1  
  
         if player1\_points < self.player1.score:  
             self.player1\_score += 1  
         elif player2\_points < self.player2.score:  
             self.player2\_score += 1  
  
         game\_info = GameInformation(  
             self.player2\_hits, self.player1\_hits, self.player2\_score, self.player1\_score)  
  
         if self.player1.score >= 10 or self.player2.score >= 10:  
             self.running = False  
  
     pygame.quit()  
  
 def loop(self):  
     ball\_x\_velocity = self.ball.velocity\_x  
     player1\_points = self.player1.score  
     player2\_points = self.player2.score  
  
     self.handle\_events()  
     self.update()  
     self.draw()  
  
     if ball\_x\_velocity > 0 > self.ball.velocity\_x:  
         self.player1\_hits += 1  
     elif self.ball.velocity\_x > 0 > ball\_x\_velocity:  
         self.player2\_hits += 1  
  
     if player1\_points < self.player1.score:  
         self.player1\_score += 1  
     elif player2\_points < self.player2.score:  
         self.player2\_score += 1  
  
     game\_info = GameInformation(  
         self.player2\_hits, self.player1\_hits, self.player2\_score, self.player1\_score)  
     return game\_info

The Game class represents an instance of the Pong game. It is responsible for running every aspect of the game.

Initialization

**\_\_init\_\_(self):** The constructor method initializes the game's attributes and variables. It sets the initial state of the game to not running, initializes the game screen and clock, creates instances of the Player and Ball classes, and sets variables corresponding with the GameInformation class variables.

**initialize(self):** This method is responsible for initializing the Pygame module and setting up the game window. It creates a Pygame screen with specified dimensions (SCREEN\_WIDTH and SCREEN\_HEIGHT), sets the window title, initializes the clock for frame rate control, creates instances of the Player class for both players, passing the x values for their positions on screen, creates an instance of the Ball class, and sets the game state to running.

Handle Events Method

**handle\_events(self):** This method is responsible for handling various events.

**Quitting:** In the event that a player tries to quit the game early, this method will set the game’s running state to False. This will result in the post-game sequence starting.

**Keypresses:** The variable, keys, is defined here as the keyboard key that is currently being pressed.

**Player 1 Movement:** The method checks for if the up or down arrow keys are being pressed. These call the move\_up() and move\_down() functions respectively for player 1.

**Player 2 Movement:** The method checks for if the “w” or “s” keys are being pressed. These call the move\_up() and move\_down() functions respectively for player 2.

Updating and Drawing

**update(self):** This method is responsible for updating the state of the ball. It is called once every frame, and it calls the ball’s update function passing both players.

**draw(self):** This method is responsible for drawing the game elements on the screen. It fills the screen with the color defined as BLACK. It then draws a dashed line down the center of the screen to represent the boundary between each player’s side using a loop. The paddles for both players are drawn as rectangles using the pygame.draw.rect() function. The ball is drawn using the draw() method of the Ball class. The scores for both players are rendered as text using the pygame.font.Font.render() function and then added to the screen. Finally, the updated screen is displayed using pygame.display.flip().

Run Method

**run(self):** This method is the main game loop that controls the flow of the game. It first calls the initialize() method to set up the game, then it enters into the main game loop.

**Main Game Loop:** The main game loop is contained in a while loop that continues as long as the game is running. Within the loop, it sets variables for the ball’s velocity and each player’s current score, it calls the handle\_events(), update(), and draw() methods and the clock is ticked to control the frame rate, limiting it to 60 frames per second.

**Unused game\_info variable:** Code within the loop here checks if the ball has been hit by either player or if a point was scored in a given frame. It uses the variables set at the beginning of the while loop to check for changes. If necessary, the corresponding GameInformation variable is updated. The GameInformation instance is then set as the value for an unused game\_info variable.

**Post Game:** It checks if either player has reached a score of 10, and if so, it sets the game state to not running, to which will end the loop.

Loop Method

**loop(self):** This method is similar to the run(self) method except it only runs through a single frame in the game.

It sets variables for the ball’s velocity and each player’s current score.

It calls handle\_events(), update(), and draw()

It updates the GameInformation values then returns game\_info which contains an instance of GameInformation with its updated values.

NEAT Implementation Code Breakdown

NEAT was implemented into a separate python file (main.py), importing pong.py along with other modules as a dependency. Here, the NEAT algorithm is implemented which learns how to play the Pong game. With a few tweaks, the program can also test the saved best AI from a training run against either a human player or a hardcoded “computer” player that only follows the ball reactively.

Import Statements

import pygame  
from PongWithClasses import pong  
import neat  
import os  
import pickle

This code imports the required modules for the Pong game, NEAT algorithm, file system operations, and data serialization.

**pygame** is a library for creating video games. It provides functionality for handling graphics, user input, and sound.

**pong** is the python file containing the Pong game code. It is located within a folder called PongWithClasses, which is located in the same directory as this python program.

**neat** is a Python library for implementing the NEAT (NeuroEvolution of Augmenting Topologies) algorithm. NEAT is an evolutionary algorithm used for evolving artificial neural networks.

**os** provides functions for interacting with the operating system. It is used to handle file system operations.

**pickle** is used for serializing and deserializing Python objects. In this program, it is used to save and load the best AI player (genome) after the NEAT training process.

PongGame Class

class PongGame:  
 def \_\_init\_\_(self):  
     self.game = pong.Game()  
     self.game.initialize()  
     self.right\_paddle = self.game.player1  
     self.left\_paddle = self.game.player2  
     self.ball = self.game.ball  
  
 def test\_ai(self, genome, config):  
     net = neat.nn.FeedForwardNetwork.create(genome, config)  
  
     run = True  
     clock = pygame.time.Clock()  
     while run:  
         clock.tick(60)  
         for event in pygame.event.get():  
             if event.type == pygame.QUIT:  
                 run = False  
                 break  
  
         keys = pygame.key.get\_pressed()  
  
         if keys[pygame.K\_UP]:  
             self.right\_paddle.move\_up()  
         elif keys[pygame.K\_DOWN]:  
             self.right\_paddle.move\_down()  
  
         output = net.activate((self.left\_paddle.y, self.ball.y, abs(self.left\_paddle.x - self.ball.x)))  
         decision = output.index(max(output))  
  
         if decision == 0:  
             pass  
         elif decision == 1:  
             self.left\_paddle.move\_up()  
         else:  
             self.left\_paddle.move\_down()  
  
         game\_info = self.game.loop()  
         # print(game\_info.player2\_hits, game\_info.player1\_hits, game\_info.player2\_score, game\_info.player1\_score)  
         self.game.draw()  
  
 def train\_ai(self, genome1, genome2, config):  
     net1 = neat.nn.FeedForwardNetwork.create(genome1, config)  
     net2 = neat.nn.FeedForwardNetwork.create(genome2, config)  
  
     run = True  
     while run:  
         for event in pygame.event.get():  
             if event.type == pygame.QUIT:  
                 run = False  
                 break  
  
         output1 = net1.activate((self.right\_paddle.y, self.ball.y, abs(self.right\_paddle.x - self.ball.x)))  
         decision1 = output1.index(max(output1))  
  
         if decision1 == 0:  
             pass  
         elif decision1 == 1:  
             self.right\_paddle.move\_up()  
         else:  
             self.right\_paddle.move\_down()  
  
         output2 = net2.activate((self.left\_paddle.y, self.ball.y, abs(self.left\_paddle.x - self.ball.x)))  
         decision2 = output2.index(max(output2))  
  
         if decision2 == 0:  
             pass  
         elif decision2 == 1:  
             self.left\_paddle.move\_up()  
         else:  
             self.left\_paddle.move\_down()  
  
         game\_info = self.game.loop()  
  
         self.game.draw()  
  
         if game\_info.player1\_score >= 1 or game\_info.player2\_score >= 1 or game\_info.player1\_hits > 50:  
             self.calculate\_fitness(genome1, genome2, game\_info)  
             break  
  
 def calculate\_fitness(self, genome1, genome2, game\_info):  
     genome1.fitness += game\_info.player1\_hits  
     genome2.fitness += game\_info.player2\_hits

The PongGame class sets up a game environment for Pong and allows the training and testing of AI agents using the NEAT algorithm. During training, the AI agents control both paddles and try to play against each other to maximize their fitness based on the number of hits they make during the game. During testing, the AI genome that is passed plays against either a human or a hardcoded “computer” player.

Initialization

**\_\_init\_\_(self):** This method is what initializes an instance of the PongGame class. It sets *game* to an instance of the Game class from pong.py, it initializes *game*, it sets the right and left paddles to be players 1 and 2 respectively, and it sets the ball to the game’s ball.

AI Testing

**test\_ai(self, genome, config):** This method handles testing the AI once it has already been trained. It is passed a genome and the configuration file. It creates a neural network using the genome and the configuration, sets run equal to true and initializes a clock. It then enters a while loop.

The while loop checks if the game has been quit. It then checks for keypresses and moves the right paddle up or down depending on the user input. It uses the neural network to generate an output then uses the value of the output to move the left paddle.

It sets the game\_info value to be equal to what the game.loop() function returns, then has the option to print these values. Finally, the screen is drawn and the loop starts over.

The while loop ticks the clock at a rate of 60, meaning the loop will only run up to 60 times per second.

AI Training

**train\_ai(self, genome1, genome2, config):** This method handles training two AI players against each other. It is passed two genomes (representing the neural networks of the AI players) and the NEAT configuration file.

The method creates two neural networks net1 and net2 using the respective genomes and the configuration.

It sets run to true and enters a while loop.

The while loop checks if the game has been quit. It then activates net1 and net2 with their respective game states (right paddle's y-position, ball's y-position, and distance between the right paddle and ball, and left paddle's y-position, ball's y-position, and distance between the left paddle and ball).

The output of each neural network is used to determine the movement of each AI player (right and left paddles).

The game state is updated with the ball's movement and collision checks.

The screen is drawn to show the updated positions of the paddles and the ball.

The loop continues until one of the players scores a point or one of the genomes hits the ball 50 times. When this condition is met, the calculate\_fitness() method is called to update the fitness of the two genomes, and the loop is exited.

Calculate Fitness Method

**calculate\_fitness(self, genome1, genome2, game\_info):** This method calculates and updates the fitness of two AI players based on the game information. It is passed two genomes and game\_info. It then adds the number of hits each genome had to its fitness score.

Eval Genomes Function

def eval\_genomes(genomes, config):  
 for i, (genome\_id1, genome1) in enumerate(genomes):  
     if i == len(genomes) - 1:  
         break  
     genome1.fitness = 0  
     for genome\_id2, genome2 in genomes[i+1:]:  
        genome2.fitness = 0 if genome2.fitness == None else genome2.fitness  
         game = PongGame()  
         game.train\_ai(genome1, genome2, config)

This function evaluates the fitness of each AI player genome in the population against one another. It is passed a list of genomes and the NEAT configuration. It runs each genome against each other genome exactly once.

The function iterates through the list of genomes using a for loop with an enumeration. For each genome, it sets the fitness to 0, initializing it for the evaluation process.

Then, it enters another for loop that iterates through the remaining genomes in the list (after the current genome). This is done by slicing the list from i+1 onwards. For each pair of genomes, it sets the fitness of the second genome (genome2) to 0 if it is not already assigned (None). This ensures that each genome’s fitness starts at 0, but will never be set back to 0 if it is called again after being given a fitness value.

It creates a new PongGame instance (game) to allow the two AI players represented by genome1 and genome2 to compete against each other. Finally, it calls the PongGame’s train\_ai() method passing the two selected genomes and the configuration.

Run NEAT Function

def run\_neat(config):  
 # p = neat.Checkpointer.restore\_checkpoint('neat-checkpoint-27')  # starts at a saved checkpoint  
 p = neat.Population(config)  
 p.add\_reporter(neat.StdOutReporter(True))  
 stats = neat.StatisticsReporter()  
 p.add\_reporter(stats)  
 p.add\_reporter(neat.Checkpointer(1))  # saves a checkpoint after every nth generation  
  
 winner = p.run(eval\_genomes, 50)  
 # returns either the best genome after n generations or the first one to reach the fitness threshold  
 with open('best.pickle', 'wb') as f:  
     pickle.dump(winner, f)

This function runs the NEAT algorithm to evolve the AI players. It is passed the NEAT configuration. It runs for a specified number of generations, and saves the best AI player genome to a file for future use (e.g., testing the trained AI player's performance). The NEAT algorithm aims to improve the fitness of the AI players through successive generations by evolving their neural network architectures and weights.

It first creates a NEAT population using the provided configuration. The population represents a collection of AI player genomes (neural networks) that will be evolved through generations. Then it adds a reporter to the NEAT population. The StdOutReporter will print the progress of the NEAT algorithm to the console during training.

Then a checkpointer is added, which saves a checkpoint (snapshot) of the population after every nth generation.

Finally the NEAT algorithm is run by calling the run() method on the NEAT population. The eval\_genomes function is passed as the first argument, which will be used to evaluate the fitness of the genomes in each generation. The second argument 50 indicates the number of generations the algorithm will run.

After the NEAT algorithm finishes running, the best genome (AI player) from the final generation is saved to a file named 'best.pickle' using the pickle.dump() method. This best genome represents the AI player with the highest fitness achieved during the evolution process.

Test AI Function

def test\_ai(config):  
 with open('best.pickle', 'rb') as f:  
     winner = pickle.load(f)  
  
 game = PongGame()  
 game.test\_ai(winner, config)

This function is what starts the testing process after the AI is trained. It is passed the configuration. It opens the file containing the best genome then runs the PongGame’s test\_ai() method passing the genome and the configuration.

Running the Program

if \_\_name\_\_ == "\_\_main\_\_":  
 local\_dir = os.path.dirname(\_\_file\_\_)  
 config\_path = os.path.join(local\_dir, 'config.txt')  
  
 config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,  
                          neat.DefaultSpeciesSet, neat.DefaultStagnation,  
                      config\_path)  
  
 run\_neat(config)  
 # test\_ai(config)

This is what runs first when the program is run. It sets up the configuration then trains or tests the AI.

It first searches for the config.txt file in the local directory, which contains all of the configuration arguments for NEAT. It then sets the config object to be an instance of the neat Config class, passing the necessary arguments.

Next, run\_neat(config) is called, which starts the training process. However, the code can be modified to instead call test\_ai(config), which will test the saved best AI from a previous training run.