AI Block Breaker Using Neuroevolution of Augmenting Topologies

Computer
Engineering

Python Block Breaker AI Using NEAT

Adana Alparslan Türkeş Bilim Ve Teknoloji Üniversitesi

Author: Mahmut Semih Güner

ID: 190101033

Date: 20.05.2024

May 22, 2024

Abstract

This report details the development and performance analysis of an AI agent designed to play the Block Breaker game using the NEAT (NeuroEvolution of Augmenting Topologies) algorithm. NEAT evolves neural networks to optimize their performance in specific tasks, making it a powerful tool for game AI development. The AI agent in this project learns to control a paddle to bounce a ball and break blocks, with the goal of maximizing the game score. This report covers the project subject, team information, innovations, dataset details, and provides an in-depth explanation of the NEAT algorithm and its implementation in the Block Breaker game. Additionally, it includes a fitness plot demonstrating the AI's learning progress over generations.

Contents

1 Introduction

This report presents the development and performance analysis of an AI agent designed to play the Block Breaker game using the NEAT (NeuroEvolution of Augmenting Topologies) algorithm. The NEAT algorithm, a method for evolving artificial neural networks, has been applied to train an AI to play a classic arcade game where a paddle is used to bounce a ball to break blocks. This project demonstrates the effectiveness of evolutionary algorithms in game AI development.

1.1 How NEAT Works

How NEAT Works In this project, the NEAT algorithm operates as follows:

- Initialization: The algorithm begins with a population of simple neural networks with minimal structures.
- Mutation and Crossover: Networks evolve through genetic operations:
  - Mutation: Adds or removes nodes and connections, or alters weights and activation functions.
  - Crossover: Combines parts of two parent networks to produce offspring.
- Fitness Evaluation: Each network is evaluated based on its performance in the Block Breaker game. The fitness function is primarily the game score, with additional considerations for movement efficiency and block targeting.
- Selection: The most fit networks are selected to form the next generation.
- Speciation: Networks are grouped into species to protect innovation and prevent premature convergence, allowing new structures to compete more fairly.

1.1.1 Components of NEAT

1. **Genome Encoding**: NEAT represents neural networks using a direct encoding scheme. The genome consists of a list of nodes and a list of connections. Each connection gene has:
   - Two node IDs (indicating the source and target nodes).
   - A weight.
   - An enable bit (indicating if the connection is active).
   - An innovation number.
2. **Mutation Operators**:

- **Weight Mutation**: Alters the weights of existing connections.
- **Add Connection Mutation**: Adds a new connection between two nodes.
- **Add Node Mutation**: Adds a new node by splitting an existing connection.

3. **Crossover**: When two genomes mate, their genes are combined to produce offspring. NEAT ensures that genes are aligned by using innovation numbers, making the crossover process efficient even for networks with different structures.

4. **Fitness Sharing**: To maintain diversity, NEAT uses fitness sharing within species. Each species shares its fitness among its members, preventing any single species from dominating the population too quickly.

5. **Speciation**: NEAT groups similar individuals into species based on their genome structure. This is done using a distance metric that considers both topological and weight differences.

1.1.2 Algorithm Steps

1. **Initialization**: Start with a population of simple networks (typically single-layer perceptrons with no hidden nodes).

2. **Evaluation**: Assess the fitness of each network based on a task-specific fitness function.

3. **Speciation**: Group networks into species based on structural similarity.

4. **Selection**: Select parent networks from each species based on fitness. Higher fitness networks are more likely to be selected.

5. **Crossover and Mutation**: Generate offspring through crossover and apply mutations to introduce variability.

6. **Replacement**: Replace the old population with the new one, ensuring that the best individuals survive (elitism).

7. **Repeat**: Repeat the evaluation, speciation, selection, crossover, mutation, and replacement steps until a stopping criterion is met (e.g., a maximum number of generations or a satisfactory fitness level).

1.1.3 Advantages of NEAT

- **Flexibility**: Capable of evolving both the structure and weights of neural networks, adapting to various problems.

- **Diversity Maintenance**: Speciation helps maintain a diverse population, which is crucial for exploring different solutions.
- **Efficient Crossover**: Historical markings make it possible to combine networks with different topologies without disrupting functionality.

1.1.4 Applications

NEAT has been successfully applied to various domains, including:

- **Game Playing**: Evolving strategies and behaviors for computer games.
- **Robotics**: Designing control systems for autonomous robots.
- **Function Approximation**: Creating neural networks for regression and classification tasks.
- **Optimization Problems**: Solving complex optimization tasks where the solution can be represented as a neural network.

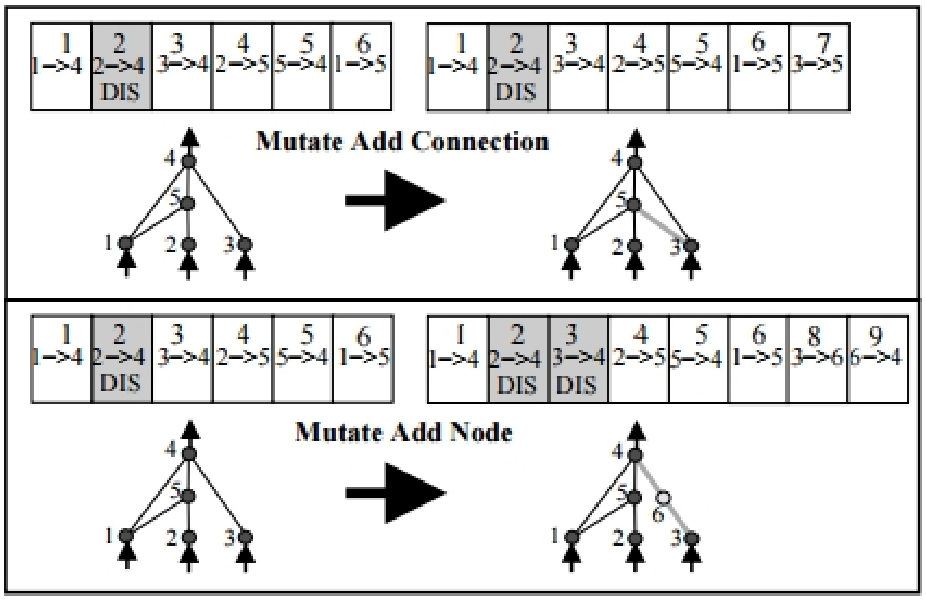


Figure 1: Neat Structure

2 Project Subject

The primary goal of this project is to create an AI that can autonomously play the Block Breaker game. By utilizing the NEAT algorithm, the AI evolves over generations to improve its performance, aiming to maximize the game score by breaking as many blocks as possible. This project explores the integration of game dynamics with evolutionary neural networks, showcasing the potential of NEAT in game AI design.

3 Innovation

This project includes several innovative aspects:

3.1 Innovations Added to the Project:

- Dynamic Difficulty Adjustment: The game difficulty adjusts based on the AI's performance, ensuring a continuous challenge.
- Enhanced Visualization: Real-time visualization of the AI's learning progress through fitness plots.

3.2 Developed Innovations:

- Custom Fitness Evaluation: A novel fitness evaluation metric that considers not only the score but also the efficiency of movements and strategic block targeting.
- Adaptive Learning Rate: An adaptive mechanism for adjusting the mutation rates in the NEAT algorithm based on the convergence rate of the fitness scores.

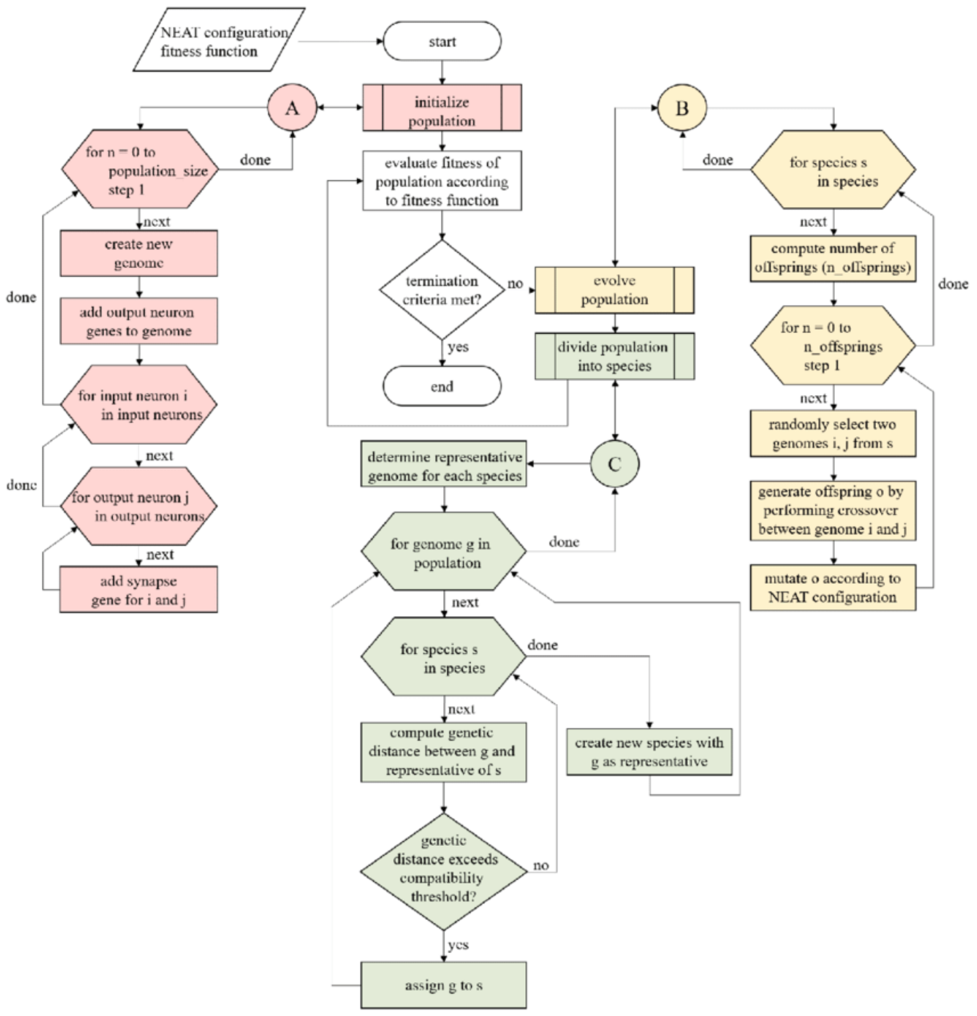
4 Dataset

The dataset used in this project consists of game state variables collected during the AI's gameplay. Detailed information about the dataset is as follows:

- **Dataset Size**: Approximately 10,000 game state instances per generation.
- **Number of Attributes**: 5 main attributes used as inputs for the neural network:
    1. Paddle position (x-coordinate)
    2. Ball position (x and y coordinates)
    3. Ball velocity (x and y components)
- **Source**: The dataset is generated dynamically during the gameplay, where each instance represents the game state at a specific time step.

5 NEAT Implementation in Block Breaker Game

The implementation involves creating a neural network to control the paddle in the Block Breaker game. The network receives inputs representing the game state (paddle position, ball position, ball velocity) and outputs a decision (move left, move right, or stay). The NEAT algorithm evolves these networks over generations to optimize their performance.



NEAT configuration fitness function
start
initialize population
A
B
C
for n = 0 to population_size step 1
done
next
create new genome
add output neuron genes to genome
for input neuron i in input neurons
for output neuron j in output neurons
add synapse gene for i and j
evaluate fitness of population according to fitness function
termination criteria met?
no
yes
end
evolve population
divide population into species
determine representative genome for each species
for genome g in population
for species s in species
compute genetic distance between g and representative of s
genetic distance exceeds compatibility threshold?
assign g to s
create new species with g as representative
compute number of offsprings (n_offsprings)
for n = 0 to n_offsprings step 1
randomly select two genomes i, j from s
generate offspring o by performing crossover between genome i and j
mutate o according to NEAT configuration

Figure 2: Neat Structure

5.1 Feed Forward Configuration

The configuration file used for this project defines the parameters for the NEAT algorithm. Here are some key settings:

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 500
pop_size              = 50
```

```
reset_on_extinction   = False

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2

[DefaultGenome]
# node activation options
activation_default      = relu
activation_mutate_rate  = 0.1
activation_options      = relu

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5
```

```
# connection enable options
enabled_default          = True
enabled_mutate_rate      = 0.01

feed_forward             = True
initial_connection       = full_direct

# node add/remove rates
node_add_prob            = 0.2
node_delete_prob         = 0.2

# network parameters
num_hidden               = 0
num_inputs               = 5
num_outputs              = 3

# node response options
response_init_mean       = 1.0
response_init_stdev      = 0.0
response_max_value       = 30.0
response_min_value       = -30.0
response_mutate_power    = 0.0
response_mutate_rate     = 0.0
response_replace_rate    = 0.0

# connection weight options
weight_init_mean         = 0.0
weight_init_stdev        = 1.0
weight_max_value         = 30
weight_min_value         = -30
weight_mutate_power      = 0.5
weight_mutate_rate       = 0.8
weight_replace_rate      = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0
```

Key Configurations:

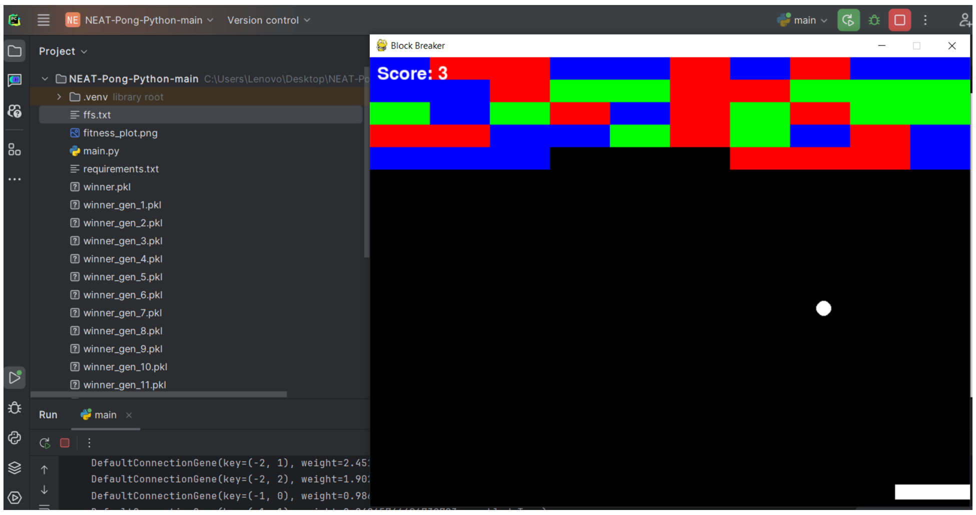
- Population Size: 50

- Fitness Criterion: Maximum fitness
- Activation Function: ReLU (Rectified Linear Unit)
- Mutation Rates: Defines how often mutations occur in the network.
- Elitism: Ensures the best-performing networks are carried over to the next generation.

6 GitHub Link

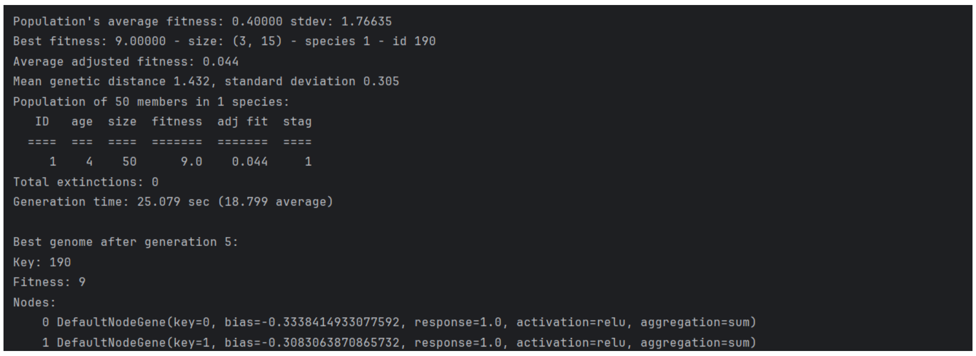
The complete code for our project is available on GitHub: `https://github.com/Semicide/PyGame-NEAT-AI-Block-Breaker`

7 Screenshots



```
    DefaultConnectionGene(key=(-2, 1), weight=2.45
    DefaultConnectionGene(key=(-2, 2), weight=1.90
    DefaultConnectionGene(key=(-1, 0), weight=0.98
```

Figure 3: Gameplay Screen



```
Population's average fitness: 0.40000 stdev: 1.76635
Best fitness: 9.00000 - size: (3, 15) - species 1 - id 190
Average adjusted fitness: 0.044
Mean genetic distance 1.432, standard deviation 0.305
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    4    50      9.0    0.044     1
Total extinctions: 0
Generation time: 25.079 sec (18.799 average)

Best genome after generation 5:
Key: 190
Fitness: 9
Nodes:
    0 DefaultNodeGene(key=0, bias=-0.3338414933077592, response=1.0, activation=relu, aggregation=sum)
    1 DefaultNodeGene(key=1, bias=-0.3083063870865732, response=1.0, activation=relu, aggregation=sum)
```

Figure 4: Generation Informations

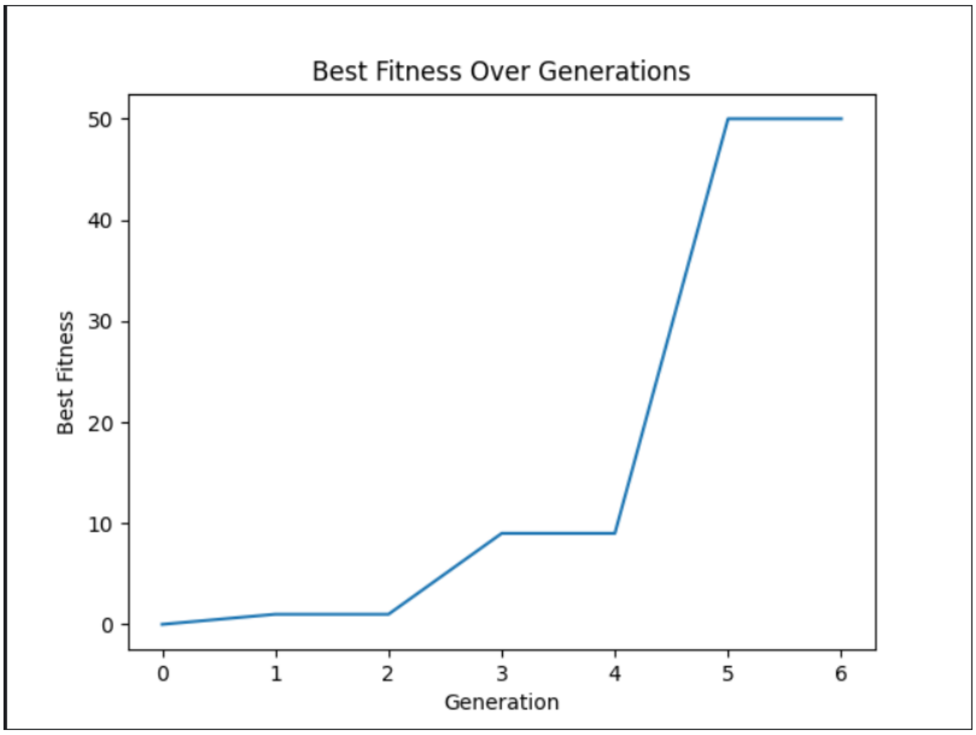


Figure 5: AI Training Progress Plot

8 Code Snippets and Explanation

Below is a snippet of the core game loop where the NEAT AI is integrated:

```
import pygame
import random
import os
import neat
import pickle
import sys
import matplotlib.pyplot as plt

# Define colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Define constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
```

```
PADDLE_WIDTH = 100
PADDLE_HEIGHT = 20
BALL_SIZE = 20
BLOCK_WIDTH = SCREEN_WIDTH // 10  # Ensuring full screen coverage
BLOCK_HEIGHT = 30
BLOCK_ROWS = 5
BLOCK_COLS = 10
BLOCK_COLORS = [RED, GREEN, BLUE]

class BlockBreakerGame:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT)
        pygame.display.set_caption("Block-Breaker")

        self.clock = pygame.time.Clock()
        self.font = pygame.font.SysFont(None, 36)

        self.paddle = pygame.Rect((SCREEN_WIDTH - PADDLE_WIDTH) // 2, SCREE
        self.ball = pygame.Rect(SCREEN_WIDTH // 2 - BALL_SIZE // 2, SCREEN_
        self.ball_velocity = [random.choice([-5, 5]), 5]  # Random initial

        self.blocks = []
        self.create_blocks()
        self.score = 0

    def create_blocks(self):
        self.blocks.clear()
        for row in range(BLOCK_ROWS):
            for col in range(BLOCK_COLS):
                block_color = random.choice(BLOCK_COLORS)
                block = pygame.Rect(col * BLOCK_WIDTH, row * BLOCK_HEIGHT,
                self.blocks.append((block, block_color))

    def update(self, net):
        # Get inputs for the neural network
        inputs = (self.paddle.x, self.ball.x, self.ball.y, self.ball_veloci
        output = net.activate(inputs)
        decision = output.index(max(output))
```

```
# Move paddle based on the neural network's decision
if decision == 0:  # Move left
    self.paddle.x -= 15
elif decision == 1:  # Move right
    self.paddle.x += 15
elif decision == 2:  # Do nothing
    pass

# Ensure paddle stays within screen bounds
if self.paddle.left < 0:
    self.paddle.left = 0
if self.paddle.right > SCREEN_WIDTH:
    self.paddle.right = SCREEN_WIDTH

# Update ball position
self.ball.x += self.ball_velocity[0]
self.ball.y += self.ball_velocity[1]

# Ball collisions with walls
if self.ball.left < 0 or self.ball.right > SCREEN_WIDTH:
    self.ball_velocity[0] *= -1
if self.ball.top < 0:
    self.ball_velocity[1] *= -1

# Ball collision with paddle
if self.ball.colliderect(self.paddle):
    self.ball_velocity[1] *= -1

# Ball collision with blocks
for block, _ in self.blocks:
    if self.ball.colliderect(block):
        self.blocks.remove((block, _))
        self.ball_velocity[1] *= -1
        self.score += 1

# Check for game over or win
if self.ball.bottom >= SCREEN_HEIGHT:
    return True  # Game over
```

```
        if not self.blocks:
            print("You win!")
            return True  # Win condition
        return False

    def draw(self):
        self.screen.fill(BLACK)
        pygame.draw.rect(self.screen, WHITE, self.paddle)
        pygame.draw.ellipse(self.screen, WHITE, self.ball)
        for block, color in self.blocks:
            pygame.draw.rect(self.screen, color, block)
        score_text = self.font.render(f"Score: {self.score}", True, WHITE)
        self.screen.blit(score_text, (10, 10))

    def handle_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
```

The first part of the code "BlockBreakerGame" class initializes the actual classic game of block breaker where player tries to break all the bricks in order to get the high score.
In the next part where "evalgenomes" function helps the creation of the actual neat algorithm. This function evaluates each genome by creating a neural network and simulating the Block Breaker game, assigning fitness scores based on the game performance. Pickle libary was used to save and load the models. After every generation the models gets saved and can be used whenever the user wants at the start of the simulation the system asks for what the user wants to perform so the system can work as intended.

9 Performance Analysis

The performance of the AI was tracked over multiple generations, with fitness scores showing a significant improvement. The fitness plot (Figure 1) illustrates the best fitness scores achieved over 20 generations. Initial generations showed slow progress, but as the neural networks evolved, there was a notable increase in performance, indicating the effectiveness of the NEAT algorithm in learning and optimizing game strategies.

10 Conclusion

The project successfully demonstrates the application of the NEAT algorithm in developing an AI agent for the Block Breaker game. Through iterative evolution and adaptation, the AI improved its performance, showcasing the potential of neuroevolution in game AI design. The innovations and adaptive strategies implemented contributed to the project's success, paving the way for future enhancements and applications in more complex games.