Artificial Intelligence Lab

PROJECT REPORT

Submitted to:- M.S. Swati Kumari

Submitted by:

102103350 (Mayank Aggarwal)

102103359(Saksham Mutneja)

102103364(Samarth Thakur)

102153001(Naman Goyal)

Problem Statement:

To make an AI-controlled Flappy Bird Game.

Link to the code: https://github.com/mayank-0407/AI-Project (We had added our code in this repository and we had mentioned there how to run this code.)

Description:

The problem of creating an AI-controlled Flappy Bird game involves designing an artificial neural network (ANN) that can learn to play the game automatically by itself. Flappy Bird is a simple game where the player controls a bird that needs to navigate through a series of pipes without colliding with them. The game is difficult because the bird needs to maintain a certain level of height while also avoiding obstacles that are moving towards it at different speeds.
To solve this problem, the first step would be to create a simulation of the Flappy Bird game that can be used to train and evaluate the performance of the ANN. The game simulation would need to have the same physics and rules as the original game, but it would need to be designed so that the neural network can interact with it and control the bird's movements.

Next, an ANN architecture would need to be chosen that can effectively learn to play the game. This could involve designing a feedforward or recurrent neural network with multiple layers that can process information from the game simulation and make decisions about the bird's movements.
Once the neural network architecture is chosen, it would need to be trained using a reinforcement learning algorithm. The algorithm would use a fitness function that evaluates the performance of the neural network based on its ability to play the game. The neural network would then be trained through a process of trial and error, with the reinforcement learning algorithm adjusting the weights and biases of the neural network to improve its performance.
Finally, once the neural network is trained, it can be used to play the game automatically without human intervention. The AI-controlled Flappy Bird game could be used for entertainment, education, or research purposes, and it could potentially be extended to other types of games and applications as well.

Requirements:

- numpy
- pygame
- neat-python
- graphviz
- matplotlib

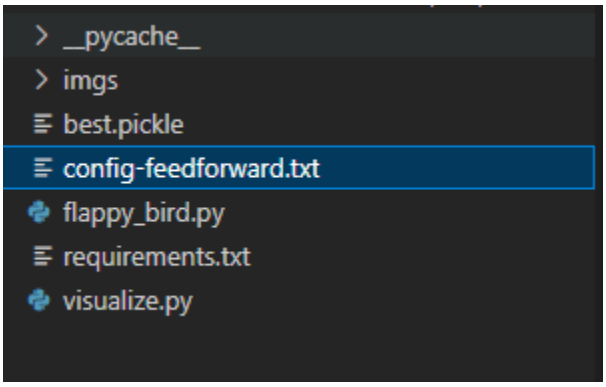
Code and Explanation:

Our Approach:

1. **Adding all the required images such as bird's,floor and pipes image .**
2. **Importing required modules.**
3. **Making functions to move birds and auto-generate random pipes on the way.**
4. **Writing functions in visualize to see how effective our model is getting trained.**
5. **At end writing a function to train the neural network for our bird.**
6. **End Result, the 4 birds will play this game for 40 gen and the**

best gen will be stored in the network with its threshold pop size.

The directory Structure:



The images folder consists of all the required images.

The best pickle consists of the data set for randomly rendering the images.

The flappy_bird.py consists of the main code to run the game.

The visualize.py consists of the codes to see a graph and the progress of the game.

Installing important modules:

Pip install pygame neat-python

IMPORTING USED MODULES

Brief description of the imported modules:

- Pygame: Pygame is a Python library designed for game development. It provides a variety of functions and tools for handling graphics, sound, and input events. Pygame can be used to create both 2D and 3D games.
- Random: The random module is a Python library that provides functions for generating random numbers and selecting random elements from a list or sequence. It can be used in a variety of applications, including game development and simulation.

- OS: The OS module is a Python library that provides a way to interact with the operating system. It provides functions for creating, deleting, and modifying files and directories, as well as for accessing system information and executing shell commands.
- Neat: NEAT (NeuroEvolution of Augmenting Topologies) is a Python library for evolving neural networks. It uses genetic algorithms to optimize the structure and parameters of neural networks. NEAT is commonly used in game development and robotics.
- Graphviz: Graphviz is a Python library for visualizing graph data structures. It provides tools for creating and rendering graphs in various formats, including PDF, SVG, and PNG.
- Numpy: Numpy is a Python library for scientific computing. It provides powerful tools for working with arrays and matrices, as well as for performing numerical computations such as linear algebra, Fourier analysis, and random number generation.
- Warnings: The warnings module is a Python library that provides a way to control warning messages generated by Python code. It can be used to display warnings when certain conditions are met or to suppress warnings altogether.

Bird Class

```
class Bird:
    """
    Bird class representing the flappy bird
    """
    MAX_ROTATION = 25
    IMGS = bird_images
    ROT_VEL = 20
    ANIMATION_TIME = 5

    def __init__(self, x, y):
        """
        Initialize the object
        :param x: starting x pos (int)
        :param y: starting y pos (int)
        :return: None
        """
        self.x = x
```

```
        self.y = y
        self.tilt = 0  # degrees to tilt
        self.tick_count = 0
        self.vel = 0
        self.height = self.y
        self.img_count = 0
        self.img = self.IMGS[0]

    def jump(self):
        """
        make the bird jump
        :return: None
        """
        self.vel = -10.5
        self.tick_count = 0
        self.height = self.y

    def move(self):
        """
        make the bird move
        :return: None
        """
        self.tick_count += 1

        # for downward acceleration
        displacement = self.vel*(self.tick_count) +
0.5*(3)*(self.tick_count)**2  # calculate displacement

        # terminal velocity
        if displacement >= 16:
            displacement = (displacement/abs(displacement)) * 16

        if displacement < 0:
            displacement -= 2

        self.y = self.y + displacement

        if displacement < 0 or self.y < self.height + 50:  # tilt up
            if self.tilt < self.MAX_ROTATION:
                self.tilt = self.MAX_ROTATION
```

```
        else:  # tilt down
            if self.tilt > -90:
                self.tilt -= self.ROT_VEL

    def draw(self, win):
        """
        draw the bird
        :param win: pygame window or surface
        :return: None
        """
        self.img_count += 1

        # For animation of bird, loop through three images
        if self.img_count <= self.ANIMATION_TIME:
            self.img = self.IMGS[0]
        elif self.img_count <= self.ANIMATION_TIME*2:
            self.img = self.IMGS[1]
        elif self.img_count <= self.ANIMATION_TIME*3:
            self.img = self.IMGS[2]
        elif self.img_count <= self.ANIMATION_TIME*4:
            self.img = self.IMGS[1]
        elif self.img_count == self.ANIMATION_TIME*4 + 1:
            self.img = self.IMGS[0]
            self.img_count = 0

        # so when bird is nose diving it isn't flapping
        if self.tilt <= -80:
            self.img = self.IMGS[1]
            self.img_count = self.ANIMATION_TIME*2


        # tilt the bird
        blitRotateCenter(win, self.img, (self.x, self.y), self.tilt)

    def get_mask(self):
        """
        gets the mask for the current image of the bird
        :return: None
        """
        return pygame.mask.from_surface(self.img)
```

The Bird class represents the player's character, a flappy bird. The class contains several attributes such as the bird's position, tilt, and velocity. The Bird class also contains methods for jumping and moving the bird, as well as for drawing the bird on the screen and getting the mask for collision detection.

Pipe Class

```
class Pipe():
    """
    represents a pipe object
    """
    GAP = 200
    VEL = 5

    def __init__(self, x):
        """
        initialize pipe object
        :param x: int
        :param y: int
        :return" None
        """
        self.x = x
        self.height = 0

        # where the top and bottom of the pipe is
        self.top = 0
        self.bottom = 0

        self.PIPE_TOP = pygame.transform.flip(pipe_img, False, True)
        self.PIPE_BOTTOM = pipe_img

        self.passed = False

        self.set_height()

    def set_height(self):
        """
        set the height of the pipe, from the top of the screen
        :return: None
```

```
        """
        self.height = random.randrange(50, 450)
        self.top = self.height - self.PIPE_TOP.get_height()
        self.bottom = self.height + self.GAP

    def move(self):
        """
        move pipe based on vel
        :return: None
        """
        self.x -= self.VEL

    def draw(self, win):
        """
        draw both the top and bottom of the pipe
        :param win: pygame window/surface
        :return: None
        """
        # draw top
        win.blit(self.PIPE_TOP, (self.x, self.top))
        # draw bottom
        win.blit(self.PIPE_BOTTOM, (self.x, self.bottom))


    def collide(self, bird, win):
        """
        returns if a point is colliding with the pipe
        :param bird: Bird object
        :return: Bool
        """
        bird_mask = bird.get_mask()
        top_mask = pygame.mask.from_surface(self.PIPE_TOP)
        bottom_mask = pygame.mask.from_surface(self.PIPE_BOTTOM)
        top_offset = (self.x - bird.x, self.top - round(bird.y))
        bottom_offset = (self.x - bird.x, self.bottom - round(bird.y))

        b_point = bird_mask.overlap(bottom_mask, bottom_offset)
        t_point = bird_mask.overlap(top_mask,top_offset)

        if b_point or t_point:
```

```
            return True

        return False
```

The Pipe class represents the obstacles in the game. The class contains attributes for the position of the pipes and their height, as well as methods for setting the height of the pipes and moving them.

Game Functions

```
class Base:
    """
    Represnts the moving floor of the game
    """
    VEL = 5
    WIDTH = base_img.get_width()
    IMG = base_img

    def __init__(self, y):
        """
        Initialize the object
        :param y: int
        :return: None
        """
        self.y = y
        self.x1 = 0
        self.x2 = self.WIDTH

    def move(self):
        """
        move floor so it looks like its scrolling
        :return: None
        """
        self.x1 -= self.VEL
        self.x2 -= self.VEL
        if self.x1 + self.WIDTH < 0:
            self.x1 = self.x2 + self.WIDTH

        if self.x2 + self.WIDTH < 0:
            self.x2 = self.x1 + self.WIDTH
```

```
    def draw(self, win):
        """
        Draw the floor. This is two images that move together.
        :param win: the pygame surface/window
        :return: None
        """
        win.blit(self.IMG, (self.x1, self.y))
        win.blit(self.IMG, (self.x2, self.y))


def blitRotateCenter(surf, image, topleft, angle):
    """
    Rotate a surface and blit it to the window
    :param surf: the surface to blit to
    :param image: the image surface to rotate
    :param topLeft: the top left position of the image
    :param angle: a float value for angle
    :return: None
    """
    rotated_image = pygame.transform.rotate(image, angle)
    new_rect = rotated_image.get_rect(center = image.get_rect(topleft =
topleft).center)

    surf.blit(rotated_image, new_rect.topleft)
```

There are several functions in the game file that control the game mechanics, such as the main game loop, collision detection, and score keeping.

Main Function

```
def draw_window(win, birds, pipes, base, score, gen, pipe_ind):
    """
    draws the windows for the main game loop
    :param win: pygame window surface
    :param bird: a Bird object
    :param pipes: List of pipes
    :param score: score of the game (int)
    :param gen: current generation
    :param pipe_ind: index of closest pipe
    :return: None
```

```
    """
    if gen == 0:
        gen = 1
    win.blit(bg_img, (0,0))

    for pipe in pipes:
        pipe.draw(win)

    base.draw(win)
    for bird in birds:
        # draw lines from bird to pipe
        if DRAW_LINES:
            try:
                pygame.draw.line(win, (255,0,0),
(bird.x+bird.img.get_width()/2, bird.y + bird.img.get_height()/2),
(pipes[pipe_ind].x + pipes[pipe_ind].PIPE_TOP.get_width()/2,
pipes[pipe_ind].height), 5)
                pygame.draw.line(win, (255,0,0),
(bird.x+bird.img.get_width()/2, bird.y + bird.img.get_height()/2),
(pipes[pipe_ind].x + pipes[pipe_ind].PIPE_BOTTOM.get_width()/2,
pipes[pipe_ind].bottom), 5)
            except:
                pass
        # draw bird
        bird.draw(win)

    # score
    score_label = STAT_FONT.render("Score: " + str(score),1,(255,255,255))
    win.blit(score_label, (WIN_WIDTH - score_label.get_width() - 15, 10))

    # generations
    score_label = STAT_FONT.render("Gens: " + str(gen-1),1,(255,255,255))
    win.blit(score_label, (10, 10))

    # alive
    score_label = STAT_FONT.render("Alive: " +
str(len(birds)),1,(255,255,255))
    win.blit(score_label, (10, 50))

    pygame.display.update()
```

The main function initializes the game window, sets the frame rate, and creates the birds and pipes for the game.

NEAT AI

```
def eval_genomes(genomes, config):
    """
    runs the simulation of the current population of
    birds and sets their fitness based on the distance they
    reach in the game.
    """
    global WIN, gen
    win = WIN
    gen += 1

    # start by creating lists holding the genome itself, the
    # neural network associated with the genome and the
    # bird object that uses that network to play
    nets = []
    birds = []
    ge = []
    for genome_id, genome in genomes:
        genome.fitness = 0  # start with fitness level of 0
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        nets.append(net)
        birds.append(Bird(230,350))
        ge.append(genome)

    base = Base(FLOOR)
    pipes = [Pipe(700)]
    score = 0

    clock = pygame.time.Clock()

    run = True
    while run and len(birds) > 0:
        clock.tick(30)

        for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                run = False
                pygame.quit()
                quit()
                break

        pipe_ind = 0
        if len(birds) > 0:
            if len(pipes) > 1 and birds[0].x > pipes[0].x +
pipes[0].PIPE_TOP.get_width():  # determine whether to use the first or
second
                pipe_ind = 1
# pipe on the screen for neural network input

        for x, bird in enumerate(birds):  # give each bird a fitness of
0.1 for each frame it stays alive
            ge[x].fitness += 0.1
            bird.move()

            # send bird location, top pipe location and bottom pipe
location and determine from network whether to jump or not
            output = nets[birds.index(bird)].activate((bird.y, abs(bird.y
- pipes[pipe_ind].height), abs(bird.y - pipes[pipe_ind].bottom)))

            if output[0] > 0.5:  # we use a tanh activation function so
result will be between -1 and 1. if over 0.5 jump
                bird.jump()

        base.move()

        rem = []
        add_pipe = False
        for pipe in pipes:
            pipe.move()
            # check for collision
            for bird in birds:
                if pipe.collide(bird, win):
                    ge[birds.index(bird)].fitness -= 1
                    nets.pop(birds.index(bird))
                    ge.pop(birds.index(bird))
```

```
                    birds.pop(birds.index(bird))

            if pipe.x + pipe.PIPE_TOP.get_width() < 0:
                rem.append(pipe)

            if not pipe.passed and pipe.x < bird.x:
                pipe.passed = True
                add_pipe = True

        if add_pipe:
            score += 1
            # can add this line to give more reward for passing through a
pipe (not required)
            for genome in ge:
                genome.fitness += 5
            pipes.append(Pipe(WIN_WIDTH))

        for r in rem:
            pipes.remove(r)

        for bird in birds:
            if bird.y + bird.img.get_height() - 10 >= FLOOR or bird.y <
-50:
                nets.pop(birds.index(bird))
                ge.pop(birds.index(bird))
                birds.pop(birds.index(bird))

        draw_window(WIN, birds, pipes, base, score, gen, pipe_ind)

        # break if score gets large enough
        # if score > 20:
        #     pickle.dump(nets[0],open("best.pickle", "wb"))
        #     break
```

This project also includes a NEAT AI system, which is a machine learning algorithm for creating neural networks. The NEAT AI is used to train the birds to play the game on their own.

Brief about Visualise.py

The code for the game includes several functions that plot statistics, plot spikes, plot species, and draw the neural network. The plot_stats() function is responsible for plotting the population's average and best fitness. The plot_spikes() function plots the trains for a single spiking neuron. The plot_species() function visualizes speciation throughout evolution. Lastly, the draw_net() function receives a genome and draws a neural network with arbitrary topology.

Plot_status()

```
def plot_stats(statistics, ylog=False, view=False,
filename='avg_fitness.svg'):
    """ Plots the population's average and best fitness. """
    if plt is None:
        warnings.warn("This display is not available due to a missing
optional dependency (matplotlib)")
        return

    generation = range(len(statistics.most_fit_genomes))
    best_fitness = [c.fitness for c in statistics.most_fit_genomes]
    avg_fitness = np.array(statistics.get_fitness_mean())
    stdev_fitness = np.array(statistics.get_fitness_stdev())

    plt.plot(generation, avg_fitness, 'b-', label="average")
    plt.plot(generation, avg_fitness - stdev_fitness, 'g-.', label="-1
sd")
    plt.plot(generation, avg_fitness + stdev_fitness, 'g-.', label="+1
sd")
    plt.plot(generation, best_fitness, 'r-', label="best")

    plt.title("Population's average and best fitness")
    plt.xlabel("Generations")
    plt.ylabel("Fitness")
    plt.grid()
    plt.legend(loc="best")
    if ylog:
        plt.gca().set_yscale('symlog')

    plt.savefig(filename)
    if view:
        plt.show()
```

```
    plt.close()
```

The plot_stats() function takes in the statistics object and plots the population's average and best fitness. The function uses the matplotlib library to create the plot. The

plot_spikes()

```
def plot_spikes(spikes, view=False, filename=None, title=None):
    """ Plots the trains for a single spiking neuron. """
    t_values = [t for t, I, v, u, f in spikes]
    v_values = [v for t, I, v, u, f in spikes]
    u_values = [u for t, I, v, u, f in spikes]
    I_values = [I for t, I, v, u, f in spikes]
    f_values = [f for t, I, v, u, f in spikes]

    fig = plt.figure()
    plt.subplot(4, 1, 1)
    plt.ylabel("Potential (mv)")
    plt.xlabel("Time (in ms)")
    plt.grid()
    plt.plot(t_values, v_values, "g-")

    if title is None:
        plt.title("Izhikevich's spiking neuron model")
    else:
        plt.title("Izhikevich's spiking neuron model
({0!s})".format(title))

    plt.subplot(4, 1, 2)
    plt.ylabel("Fired")
    plt.xlabel("Time (in ms)")
    plt.grid()
    plt.plot(t_values, f_values, "r-")

    plt.subplot(4, 1, 3)
    plt.ylabel("Recovery (u)")
    plt.xlabel("Time (in ms)")
    plt.grid()
    plt.plot(t_values, u_values, "r-")
```

```
    plt.subplot(4, 1, 4)
    plt.ylabel("Current (I)")
    plt.xlabel("Time (in ms)")
    plt.grid()
    plt.plot(t_values, I_values, "r-o")

    if filename is not None:
        plt.savefig(filename)

    if view:
        plt.show()
        plt.close()
        fig = None

    return fig
```

plot_spikes() function takes in a list of spikes and plots the potential, fired, recovery, and current values over time.

Plot_species()

```
def plot_species(statistics, view=False, filename='speciation.svg'):
    """ Visualizes speciation throughout evolution. """
    if plt is None:
        warnings.warn("This display is not available due to a missing optional dependency (matplotlib)")
        return

    species_sizes = statistics.get_species_sizes()
    num_generations = len(species_sizes)
    curves = np.array(species_sizes).T

    fig, ax = plt.subplots()
    ax.stackplot(range(num_generations), *curves)

    plt.title("Speciation")
    plt.ylabel("Size per Species")
    plt.xlabel("Generations")

    plt.savefig(filename)
```

```
    if view:
        plt.show()

    plt.close()
```

The plot_species() function takes in the statistics object and visualizes speciation throughout evolution using the stackplot() method.

Draw_net()

```
def draw_net(config, genome, view=False, filename=None, node_names=None,
show_disabled=True, prune_unused=False,
             node_colors=None, fmt='svg'):
    """ Receives a genome and draws a neural network with arbitrary
topology. """
    # Attributes for network nodes.
    if graphviz is None:
        warnings.warn("This display is not available due to a missing
optional dependency (graphviz)")
        return

    if node_names is None:
        node_names = {}

    assert type(node_names) is dict

    if node_colors is None:
        node_colors = {}

    assert type(node_colors) is dict

    node_attrs = {
        'shape': 'circle',
        'fontsize': '9',
        'height': '0.2',
        'width': '0.2'}

    dot = graphviz.Digraph(format=fmt, node_attr=node_attrs)

    inputs = set()
```

```
    for k in config.genome_config.input_keys:
        inputs.add(k)
        name = node_names.get(k, str(k))
        input_attrs = {'style': 'filled', 'shape': 'box', 'fillcolor':
node_colors.get(k, 'lightgray')}
        dot.node(name, _attributes=input_attrs)

    outputs = set()
    for k in config.genome_config.output_keys:
        outputs.add(k)
        name = node_names.get(k, str(k))
        node_attrs = {'style': 'filled', 'fillcolor': node_colors.get(k,
'lightblue')}

        dot.node(name, _attributes=node_attrs)

    if prune_unused:
        connections = set()
        for cg in genome.connections.values():
            if cg.enabled or show_disabled:
                connections.add((cg.in_node_id, cg.out_node_id))

        used_nodes = copy.copy(outputs)
        pending = copy.copy(outputs)
        while pending:
            new_pending = set()
            for a, b in connections:
                if b in pending and a not in used_nodes:
                    new_pending.add(a)
                    used_nodes.add(a)
            pending = new_pending
    else:
        used_nodes = set(genome.nodes.keys())

    for n in used_nodes:
        if n in inputs or n in outputs:
            continue

        attrs = {'style': 'filled',
                 'fillcolor': node_colors.get(n, 'white')}
```

```
        dot.node(str(n), _attributes=attrs)

    for cg in genome.connections.values():
        if cg.enabled or show_disabled:
            #if cg.input not in used_nodes or cg.output not in used_nodes:
            #    continue
            input, output = cg.key
            a = node_names.get(input, str(input))
            b = node_names.get(output, str(output))
            style = 'solid' if cg.enabled else 'dotted'
            color = 'green' if cg.weight > 0 else 'red'
            width = str(0.1 + abs(cg.weight / 5.0))
            dot.edge(a, b, _attributes={'style': style, 'color': color,
'penwidth': width})

    dot.render(filename, view=view)

    return dot
```

Lastly, the draw_net() function receives a genome and draws a neural network with arbitrary topology using the graphviz library.

Main Function to run the game:

```
def run(config_file):

    """

    runs the NEAT algorithm to train a neural network to play
flappy bird.

    :param config_file: location of config file

    :return: None

    """

    config = neat.config.Config(neat.DefaultGenome,
neat.DefaultReproduction,

                        neat.DefaultSpeciesSet,
neat.DefaultStagnation,
```

```
                           config_file)


    # Create the population, which is the top-level object for a
NEAT run.
    p = neat.Population(config)


    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    #p.add_reporter(neat.Checkpointer(5))


    # Run for up to 50 generations.
    winner = p.run(eval_genomes, 50)


    # show final stats
    print('\nBest genome:\n{!s}'.format(winner))



if __name__ == '__main__':
    # Determine path to configuration file. This path manipulation
is
    # here so that the script will run successfully regardless of
the
    # current working directory.
```

```
    local_dir = os.path.dirname(__file__)

    config_path = os.path.join(local_dir, 'config-feedforward.txt')

    run(config_path)
```

This function calls the neat and sends the required information about the genome, reproduction, species ,stagnation and configurations. It also calls the reporter function to get the reports side by side in terminal about generation of the bird.

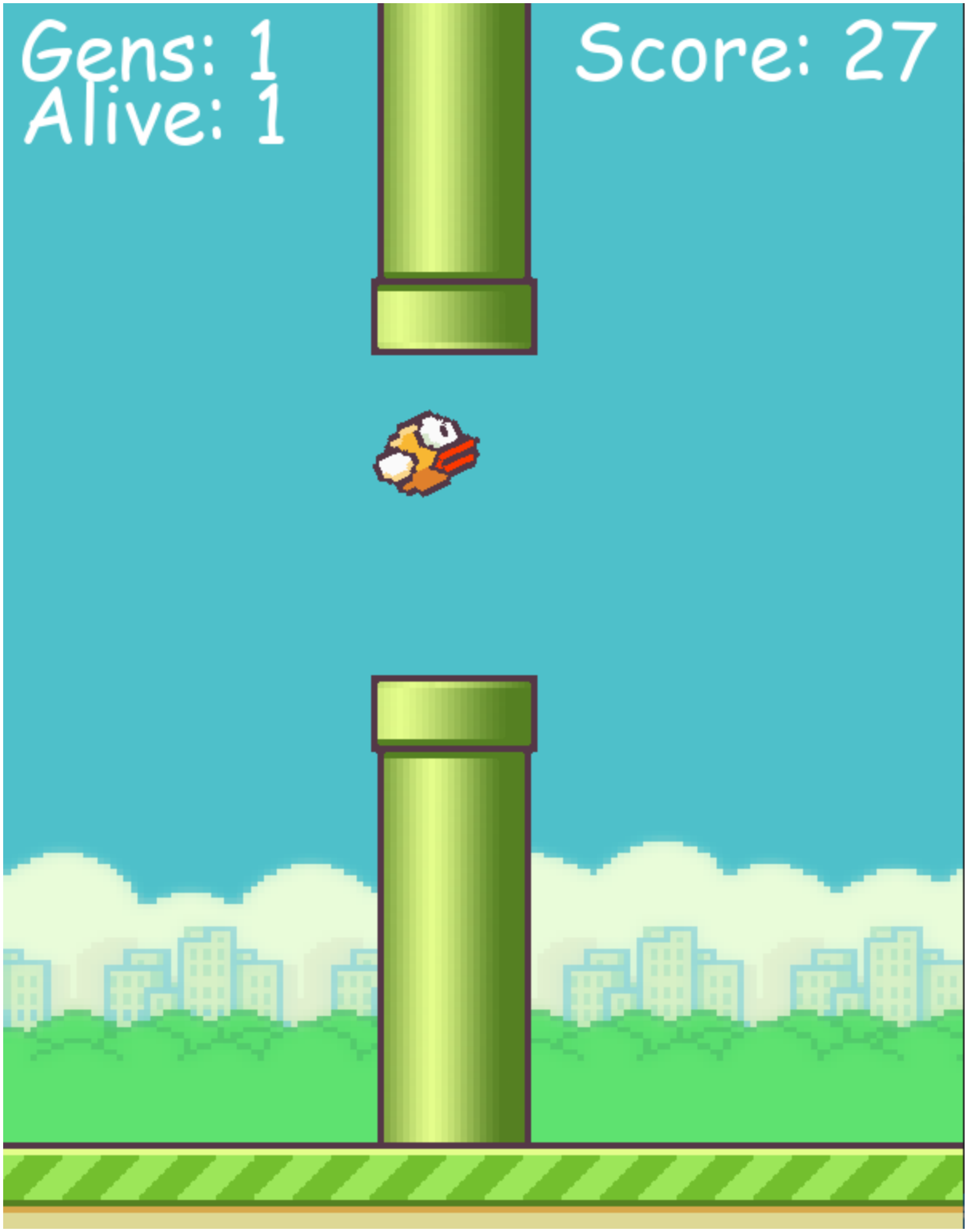
Running the Game.

```
[Running] python -u "e:\AI_project\Flappy_bird\main\flappy_bird.py"
pygame 2.3.0 (SDL 2.24.2, Python 3.10.10)
Hello from the pygame community. https://www.pygame.org/contribute.html

 ****** Running generation 0 ******

Population's average fitness: 3.55000 stdev: 1.60515
Best fitness: 7.90000 - size: (1, 3) - species 1 - id 9
Average adjusted fitness: 0.209
Mean genetic distance 1.058, standard deviation 0.385
Population of 10 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    0    10      7.9    0.209     0
Total extinctions: 0
Generation time: 2.987 sec

 ****** Running generation 1 ******
```


Gens: 1
Alive: 1
Score: 27