УДК 004

Применение нейроэволюционных алгоритмов для оптимизации поведения игровых персонажей на Python

Эрдман Александр Алексеевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данном исследовании рассматривается применение нейроэволюционных алгоритмов для оптимизации поведения игровых персонажей. Реализация эволюционных нейронных сетей происходит с помощью языка программирования Python и двух библиотек «neat-python» и «pygame». Результатом исследования является применение изучаемых алгоритмов на практическом примере.
Ключевые слова: python, нейронные сети, neat-python

The use of neuroevolutionary algorithms to optimize the behavior of game characters in Python

Erdman Alexander Alekseevich
Sholom-Aleichem Priamursky State University
Student

Abstract
This study examines the use of neuroevolutionary algorithms to optimize the behavior of game characters. The implementation of evolutionary neural networks takes place using the Python programming language and two libraries "net-python" and "pygame". The result of the study is the application of the studied algorithms on a practical example.
Keywords: python, neural networks, net-python

1 Введение

1.1 Актуальность

В современном мире игровая индустрия продолжает стремительно развиваться, предлагая пользователям все более сложные и реалистичные игровые миры. Одним из ключевых аспектов, определяющих качество и увлекательность игр, является искусственный интеллект, который управляет поведением игровых персонажей и других агентов. Традиционные методы программирования ИИ часто ограничены в своей способности адаптироваться к динамическим и непредсказуемым условиям игрового процесса. В этом контексте нейроэволюционные алгоритмы представляют собой перспективное направление, позволяющее автоматически создавать и

оптимизировать нейронные сети для управления поведением игровых персонажей.

Нейроэволюционные алгоритмы, такие как NEAT (NeuroEvolution of Augmenting Topologies), предоставляют мощные инструменты для разработки адаптивных и эффективных стратегий поведения. Эти методы позволяют нейронным сетям самостоятельно находить оптимальные решения, что значительно упрощает процесс разработки и повышает качество конечного продукта. В условиях растущей конкуренции на рынке компьютерных игр, использование нейроэволюционных алгоритмов может стать важным конкурентным преимуществом, позволяя создавать более увлекательные и реалистичные игровые миры.

1.2 Обзор исследований

А.В. Рягузов и Д.В. Борисенков в своём труде привели реализацию нейронных сетей в классической игре шахматы [1]. Г.М. Громова исследовала возможности нейронных сетей в игре арканоид [2]. Н.И. Миндияров и В.С. Дороганов разработали программу обучения искусственного интеллекта на примере игры [3]. Р.С. Гумерова в своей работе привела реализацию нейросети на базе программных модулей Python [4]. А.З. Пирматов, Б.А. Азимов и С.С. Камалов изучили роль Python в разработке и применении искусственного интеллекта [5].

1.3 Цель исследования

Целью исследования является практическое применение нейроэволюции для оптимизации управления персонажами в компьютерной игре.

2 Материалы и методы

Для реализации программы используется язык программирования Python, библиотека для упрощённого создания игр pygame, а также модуль с инструментами нейроэволюции neat-python. В качестве IDE выступает PyCharm.

3 Результаты и обсуждения

Перед началом работы из общего доступа сети интернет скачиваются заготовки игры – изображения труб, земли, фона и самих птиц. Затем импортируются библиотеки pygame и neat-python, где первая библиотека предоставляет широкий функционал для создания игр (управление, события, методы и т.п.), а вторая библиотека позволит работать с эволюционными алгоритмами.

После загрузки всего необходимого материала начинается программирование непосредственно самой игры, в частности поведение всех игровых объектов. Первым реализуется фон игры (рис. 1).

```python
import os
import random

import pygame
import utils
import config

class Background:  2 usages
    SPEED = 0.5
    CHUNKS = 4

    def __init__(self):
        self.bg_sprite = None
        self.load_sprite()
        self.chunk_width = self.bg_sprite.get_width()

        self.y = 0
        self._chunks = []
        for x in range(self.CHUNKS):
            self._chunks.append([self.bg_sprite, x * self.chunk_width])

    def load_sprite(self):  1 usage
        self.bg_sprite = pygame.transform.scale(pygame.image.load(os.path.join(config.TEXTURES_DIR,
            "background-day.png")).convert_alpha(),  size: (288, 512))


    def get_last_chunk_offset(self):  1 usage
        offset = 0

        for chunk in self._chunks:
            if chunk[1] > offset:
                offset = chunk[1]

        return offset


    def move(self):  3 usages (3 dynamic)
        for i, chunk in enumerate(self._chunks):
            chunk[1] -= self.SPEED

            if chunk[1] < -self.chunk_width:
                chunk[1] = self.get_last_chunk_offset() + self.chunk_width


    def draw(self, screen):  4 usages (4 dynamic)
        for chunk in self._chunks:
            screen.blit(chunk[0], (chunk[1], self.y))
```

Рисунок 1. Реализация заднего фона

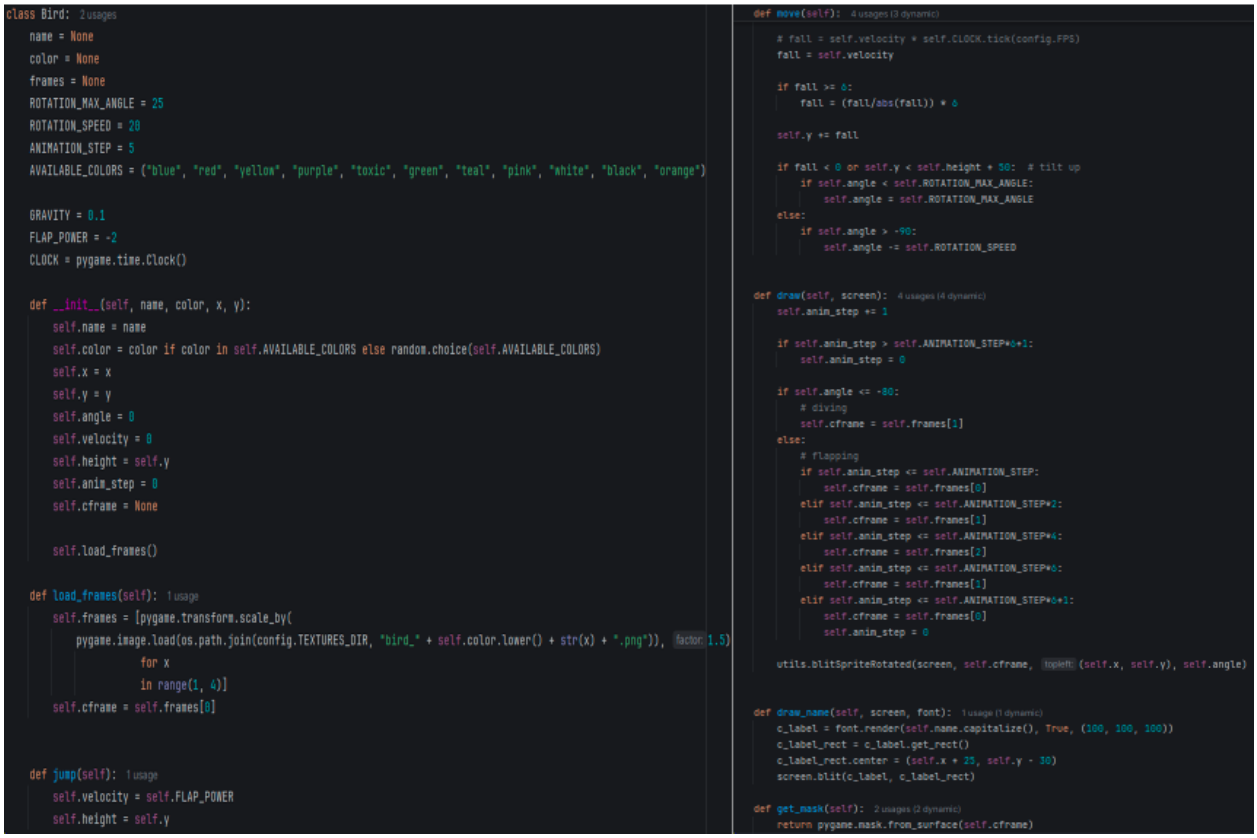
В данном коде представлен класс ‘Background’, который отвечает за создание и управление фоном в игре. Этот класс использует библиотеку

‘pygame’ для работы с графикой и взаимодействия с игровым окном. Класс ‘Background’ начинается с определения двух констант: ‘SPEED’, которая задает скорость движения фона, и ‘CHUNKS’, которая определяет количество сегментов фона. В конструкторе класса инициализируются основные переменные. Сначала загружается изображение фона с помощью функции ‘load_sprite’, которая масштабирует изображение до нужных размеров. Затем определяется ширина одного сегмента фона (‘chunk_width’). Далее создается список ‘_chunks’, который содержит части фона. Каждый сегмент представляет собой список, состоящий из изображения и его позиции по оси X.

Функция ‘load_sprite’ отвечает за загрузку и масштабирование изображения фона. Она использует ‘pygame.image.load’ для загрузки изображения из файла, расположенного в директории, указанной в конфигурационном файле. Затем изображение масштабируется до размеров 288x512 пикселей с помощью функции ‘pygame.transform.scale’. Это позволяет адаптировать изображение под размеры игрового окна и обеспечивает его корректное отображение. Метод ‘get_last_chunk_offset’ возвращает смещение последнего сегмента фона. Она проходит по всем сегментам и находит максимальное значение позиции по оси X, которое и является смещением последнего сегмента. Это необходимо для правильного перемещения сегментов фона и создания эффекта бесконечного движения.

Функционал ‘move’ отвечает за перемещение сегментов фона. Он уменьшает позицию каждого сегмента на значение ‘SPEED’, что создает эффект движения фона влево. Если сегмент выходит за левую границу экрана, его позиция устанавливается за последним сегментом, что создает бесконечный фон. Это достигается за счет постоянного перемещения сегментов и их повторного размещения за последним. Функция ‘draw’ отвечает за отображение фона на экране. Она использует ‘screen.blit’ для рисования каждого сегмента фона на экране. Позиция каждого сегмента определяется его координатой по оси X и фиксированной координатой по оси Y (‘self.y’). Таким образом, класс ‘Background’ обеспечивает создание и управление фоном в игре, создавая эффект бесконечного движения фона.

Для создания и управления птицами (игровыми персонажами) создаётся класс ‘Bird’ (рис. 2). Класс использует библиотеку ‘pygame’ для работы с графикой и взаимодействия с игровым окном. В конструкторе класса инициализируются основные переменные, включая имя птицы, цвет, начальные координаты, угол поворота, скорость, высоту, шаг анимации и текущий кадр. Если указанный цвет не входит в список доступных цветов, выбирается случайный цвет из этого списка. Затем загружаются кадры анимации птицы с помощью ‘load_frames’.



```
class Bird:  2 usages
    name = None
    color = None
    frames = None
    ROTATION_MAX_ANGLE = 25
    ROTATION_SPEED = 20
    ANIMATION_STEP = 5
    AVAILABLE_COLORS = ("blue", "red", "yellow", "purple", "toxic", "green", "teal", "pink", "white", "black", "orange")

    GRAVITY = 0.1
    FLAP_POWER = -2
    CLOCK = pygame.time.Clock()

    def __init__(self, name, color, x, y):
        self.name = name
        self.color = color if color in self.AVAILABLE_COLORS else random.choice(self.AVAILABLE_COLORS)
        self.x = x
        self.y = y
        self.angle = 0
        self.velocity = 0
        self.height = self.y
        self.anim_step = 0
        self.cframe = None

        self.load_frames()

    def load_frames(self):  1 usage
        self.frames = [pygame.transform.scale_by(
            pygame.image.load(os.path.join(config.TEXTURES_DIR, "bird_" + self.color.lower() + str(x) + ".png")), factor: 1.5)
                       for x
                       in range(1, 4)]
        self.cframe = self.frames[0]


    def jump(self):  1 usage
        self.velocity = self.FLAP_POWER
        self.height = self.y
```

```
    def move(self):  4 usages (3 dynamic)

        # fall = self.velocity * self.CLOCK.tick(config.FPS)
        fall = self.velocity

        if fall >= 0:
            fall = (fall/abs(fall)) * 6

        self.y += fall

        if fall < 0 or self.y < self.height + 50:  # tilt up
            if self.angle < self.ROTATION_MAX_ANGLE:
                self.angle = self.ROTATION_MAX_ANGLE
        else:
            if self.angle > -90:
                self.angle -= self.ROTATION_SPEED


    def draw(self, screen):  4 usages (4 dynamic)
        self.anim_step += 1

        if self.anim_step > self.ANIMATION_STEP*6+1:
            self.anim_step = 0

        if self.angle <= -80:
            # diving
            self.cframe = self.frames[1]
        else:
            # flapping
            if self.anim_step <= self.ANIMATION_STEP:
                self.cframe = self.frames[0]
            elif self.anim_step <= self.ANIMATION_STEP*2:
                self.cframe = self.frames[1]
            elif self.anim_step <= self.ANIMATION_STEP*4:
                self.cframe = self.frames[2]
            elif self.anim_step <= self.ANIMATION_STEP*6:
                self.cframe = self.frames[1]
            elif self.anim_step <= self.ANIMATION_STEP*6+1:
                self.cframe = self.frames[0]
                self.anim_step = 0

        utils.blitSpriteRotated(screen, self.cframe, topleft: (self.x, self.y), self.angle)


    def draw_name(self, screen, font):  1 usage (1 dynamic)
        c_label = font.render(self.name.capitalize(), True, (100, 100, 100))
        c_label_rect = c_label.get_rect()
        c_label_rect.center = (self.x + 25, self.y - 30)
        screen.blit(c_label, c_label_rect)

    def get_mask(self):  2 usages (2 dynamic)
        return pygame.mask.from_surface(self.cframe)
```

Рисунок 2. Класс ‘Bird’

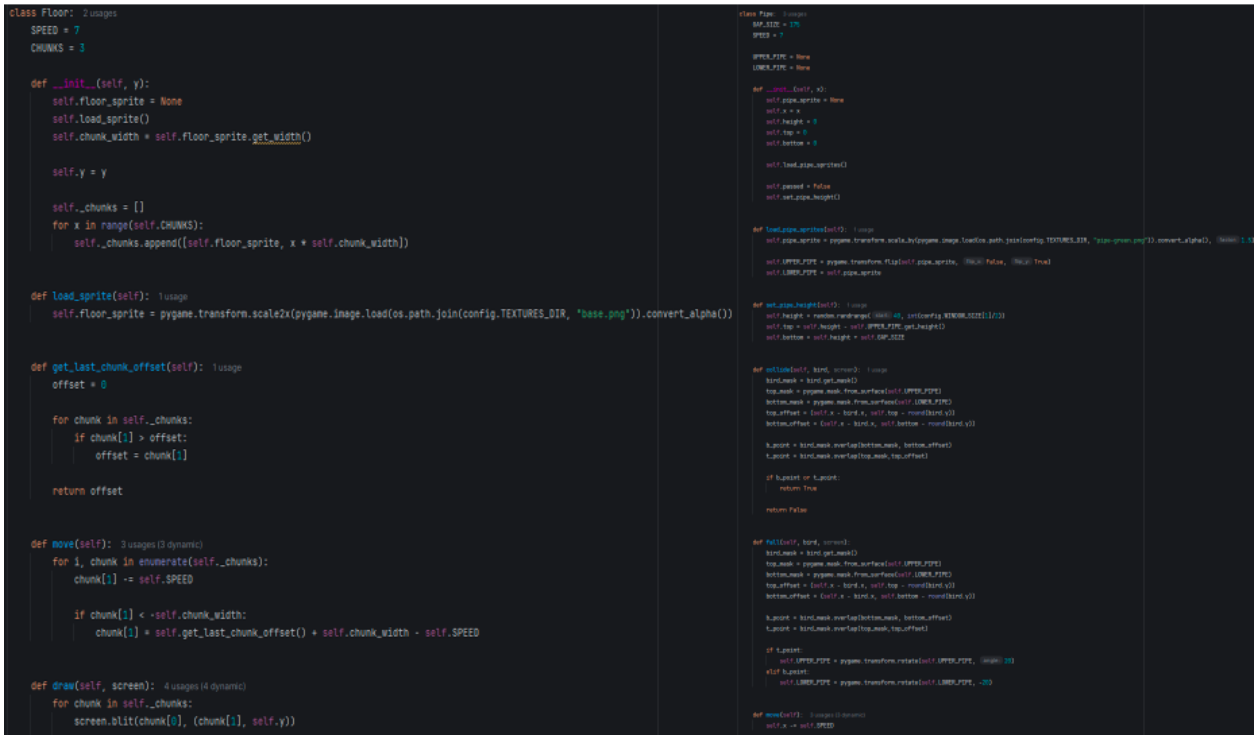
Метод ‘load_frames’ загружает и масштабирует кадры анимации птицы. Он использует ‘pygame.image.load’ для загрузки изображений из файлов и ‘pygame.transform.scale_by’ для масштабирования.

Функция ‘jump’ отвечает за прыжок птицы, устанавливая скорость птицы на значение ‘FLAP_POWER’ и обновляя высоту. Функционал ‘move’ отвечает за перемещение птицы, увеличивая скорость птицы на значение ‘GRAVITY’, что создает эффект падения. Если скорость превышает определенное значение, она ограничивается. Затем обновляется координата y птицы и угол поворота в зависимости от скорости и высоты.

‘Draw’ реализует отображение птицы на экране. Он увеличивает шаг анимации и обновляет текущий кадр в зависимости от шага анимации. Если угол поворота меньше или равен -80 градусов, используется кадр, соответствующий пикированию. В противном случае используются кадры, соответствующие маханию крыльев. Затем птица рисуется на экране с учетом поворота с помощью ‘utils.blitSpriteRotated’.

Функция ‘draw_name’ отвечает за отображение имени птицы на экране. Она создает текстовую метку с именем птицы, устанавливает ее позицию и рисует на экране. Реализация ‘get_mask’ возвращает маску текущего кадра птицы, что позволяет определять столкновения с другими объектами в игре.

Далее создаются классы ‘Floor’ и ‘Pipe’, которые отвечают за создание и управление полом и трубами в игре (рис. 3).



```python
class Floor:
    SPEED = 7
    CHUNKS = 3

    def __init__(self, y):
        self.floor_sprite = None
        self.load_sprite()
        self.chunk_width = self.floor_sprite.get_width()

        self.y = y

        self._chunks = []
        for x in range(self.CHUNKS):
            self._chunks.append([self.floor_sprite, x * self.chunk_width])


    def load_sprite(self):
        self.floor_sprite = pygame.transform.scale2x(pygame.image.load(os.path.join(config.TEXTURES_DIR, "base.png")).convert_alpha())


    def get_last_chunk_offset(self):
        offset = 0

        for chunk in self._chunks:
            if chunk[1] > offset:
                offset = chunk[1]

        return offset


    def move(self):
        for i, chunk in enumerate(self._chunks):
            chunk[1] -= self.SPEED

            if chunk[1] < -self.chunk_width:
                chunk[1] = self.get_last_chunk_offset() + self.chunk_width - self.SPEED


    def draw(self, screen):
        for chunk in self._chunks:
            screen.blit(chunk[0], (chunk[1], self.y))
```

```python
class Pipe:
    GAP_SIZE = 175
    SPEED = 7

    UPPER_PIPE = None
    LOWER_PIPE = None

    def __init__(self, x):
        self.pipe_sprite = None
        self.x = x
        self.height = 0
        self.top = 0
        self.bottom = 0

        self.load_pipe_sprites()

        self.passed = False
        self.set_pipe_height()


    def load_pipe_sprites(self):
        self.pipe_sprite = pygame.transform.scale_by(pygame.image.load(os.path.join(config.TEXTURES_DIR, "pipe-green.png")).convert_alpha(),

        self.UPPER_PIPE = pygame.transform.flip(self.pipe_sprite, False, True)
        self.LOWER_PIPE = self.pipe_sprite


    def set_pipe_height(self):
        self.height = random.randrange(
        self.top = self.height - self.UPPER_PIPE.get_height()
        self.bottom = self.height + self.GAP_SIZE


    def collide(self, bird, screen):
        bird_mask = bird.get_mask()
        top_mask = pygame.mask.from_surface(self.UPPER_PIPE)
        bottom_mask = pygame.mask.from_surface(self.LOWER_PIPE)
        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


    def fall(self, bird, screen):
        bird_mask = bird.get_mask()
        top_mask = pygame.mask.from_surface(self.UPPER_PIPE)
        bottom_mask = pygame.mask.from_surface(self.LOWER_PIPE)
        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 t_point:
            self.UPPER_PIPE = pygame.transform.rotate(self.UPPER_PIPE,
        elif b_point:
            self.LOWER_PIPE = pygame.transform.rotate(self.LOWER_PIPE, -20)


    def move(self):
        self.x -= self.SPEED
```

Рисунок 3. Класс ‘Floor’ (слева) и ‘Pipe’ (справа)

Класс ‘Floor’ отвечает за загрузку и масштабирование изображения пола, а также за управление его движением и отображением на экране. Функционал управляет сегментами пола, перемещая их и рисуя на экране, создавая иллюзию непрерывного движения.

Класс ‘Pipe’ отвечает за создание и управление трубами в игре. Он загружает и масштабирует изображения верхней и нижней труб, а также проверяет столкновения птицы с трубами, управляет их движением и отображением на экране. Помимо этого, класс может изменять ориентацию труб при столкновении с птицей, создавая визуальный эффект падения.

Игра создана. Теперь нужно реализовать нейронную сеть, чтобы игровые персонажи (птицы) могли самостоятельно управляться и обучаться на своих ошибках для последующего улучшения.

Перед реализацией нейроэволюции создаётся файл с конфигурациями сети, где определяются параметры и настройки процесса эволюции сети (рис. 4).

Данный файл включает в себя настройки, которые выполняют следующие действия: устанавливают критерии для оценки фитнеса (приспособленности) нейронных сетей, что позволяет определить, насколько хорошо сеть выполняет задачу; определяют размер популяции и параметры для управления популяцией, такие как порог фитнеса для завершения эволюции и флаг для перезапуска популяции при вымирании; устанавливают вероятности и параметры для мутаций функций активации, агрегации, смещений, весов и других элементов нейронной сети; определяют коэффициенты совместимости для разъединенных генов и весов, что позволяет определить, насколько геномы похожи друг на друга; устанавливают параметры для воспроизводства, включая количество

элитных особей и порог выживания для особей; определяют количество входных, выходных и скрытых узлов, а также параметры для добавления и удаления узлов и соединений.

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 10000
pop_size              = 25
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.01
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.01
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

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

# network parameters
num_hidden              = 0
num_inputs              = 4
num_outputs             = 1

# 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
```

Рисунок 4. Фрагмент файла конфигурации эволюции нейронной сети

Этот файл необходим для настройки и управления процессом эволюционного обучения нейронных сетей. Он позволяет пользователю задавать параметры, которые влияют на эффективность и результаты эволюционного процесса. Правильная настройка этих параметров может значительно улучшить производительность и качество обученных нейронных сетей.

Далее следует программирование главного файла проекта ‘main.py’, в котором находится функция ‘run_game’ и является центральной частью кода, отвечающей за управление игровым процессом и эволюцию нейронных сетей с использованием библиотеки NEAT (рис. 5).

```
def run_game(genomes, neat_config):  1 usage
    global gen
    gen += 1
    screen = pygame.display.set_mode((config.WINDOW_SIZE[0], config.WINDOW_SIZE[1]))
    clock = pygame.time.Clock()
    pygame.display.set_caption("Flappy Bird AI - Neuroevolution")
    background = Background()
    birds = []
    pipes = [Pipe(config.WINDOW_SIZE[0]-50),]
    floor = Floor(config.FLOOR_POS)
    score = 0

    nets = []
    for i, g in genomes:
        net = neat.nn.FeedForwardNetwork.create(g, neat_config)
        nets.append(net)
        g.fitness = 0

        birds.append(Bird(random.choice(NAMES), color: "random",random.randrange( start: 150,  stop: 300), random.randrange( start: 0,  stop: 300)))

    while True:
        clock.tick(config.FPS)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
        cpipe = 0
        if len(birds):
            if len(pipes) > 1 and birds[0].x > pipes[0].x + pipes[0].UPPER_PIPE.get_width():
                cpipe = 1

        user_input = pygame.key.get_pressed()
        for bi, bird in enumerate(birds):
            genomes[bi][1].fitness += 0.1
            bird.move()

            output = nets[bi].activate(
                (bird.y, abs(bird.y - pipes[cpipe].height), abs(bird.y - pipes[cpipe].bottom), abs(pipes[cpipe].x - bird.x)))
            if output[0] > 0.5:
                bird.jump()

                genomes[bi][1].fitness -= 1

        tbd_pipes = []
        add_new_pipe = False
        for pipe in pipes:
            # collision check
            for bi, bird in enumerate(birds):
                if pipe.collide(bird, screen):
                    print(f"Bird {bird.name} is downn ... PIPE GOT HER!")
                    genomes[bi][1].fitness -= 1
                    nets.pop(bi)
                    genomes.pop(bi)
                    birds.pop(bi)

            if pipe.x + pipe.UPPER_PIPE.get_width() < 0:
                tbd_pipes.append(pipe)

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

        if add_new_pipe:
            if birds:
                score += 1

            pipes.append(Pipe(config.WINDOW_SIZE[0]))
            for g in genomes:
                g[1].fitness += 1

        for dp in tbd_pipes:
            pipes.remove(dp)

        for bi, bird in enumerate(birds):
            if bird.y + bird.cframe.get_height() - 10 >= config.FLOOR_POS or bird.y <= -50:
                print(f"Bird {bird.name} has FALLEN ...")
                nets.pop(bi)
                genomes.pop(bi)


                birds.pop(bi)

        draw_all(screen, birds, pipes, floor, background, score,  generation: None, cpipe)

        if not birds:
            time.sleep(2)
            break


if __name__ == '__main__':
    neat_config_path = "./config-feedforward.txt"
    neat_cfg = neat.config.Config(neat.DefaultGenome, neat.DefaultReproduction, neat.DefaultSpeciesSet,
                                  neat.DefaultStagnation, neat_config_path)
    p = neat.Population(neat_cfg)
    best = p.run(run_game,  n: 1000)
    print('\nWINNER (chicken dinner) IS - \n{!s}'.format(best))
```

Рисунок 5. Реализация нейроэволюции

В начале функции происходит объявление переменной ‘gen’, которая отвечает за счёт поколений. Далее происходит инициализация игровых объектов таких, как экран, часы и заголовок окна. Затем для каждого генома создается нейронная сеть и добавляется в список ‘nets’. Каждой птице присваивается начальный фитнес (метрика успешности выполнения задачи) 0, и она добавляется в список ‘birds’.

После следует функционал, который отвечает за управление птицами с использованием нейронных сетей. Он получает текущее состояние клавиш на клавиатуре, хотя это не используется напрямую. Затем код перебирает всех птиц, увеличивает их фитнес (приспособленность) на 0.1 за каждый кадр, перемещает птиц в соответствии с физикой игры и активирует нейронные сети для принятия решений о прыжках. Нейронные сети получают входные данные, такие как текущая высота птицы, расстояние до верхней и нижней трубы, а также расстояние до ближайшей трубы. Если выходное значение сети превышает 0.5, птица прыгает, и ее фитнес уменьшается на 1, так как частые прыжки могут привести к столкновению с трубами или падению на пол. Этот процесс позволяет нейронным сетям обучаться и улучшать свою способность управления.

Для управления трубами и взаимодействием персонажами используется соответствующие методы: проверка столкновения птиц с трубами; удаление труб, которые вышли за пределы экрана; добавление новых труб при необходимости; обновление фитнеса нейронных сетей в зависимости от действий птиц.

Код начинается с инициализации списка труб, которые нужно удалить (‘tbd_pipes’), и флага для добавления новой (‘add_new_pipe’). Затем он перебирает все трубы и птиц. Если птица сталкивается с трубой, ее фитнес

уменьшается, и она удаляется из игры вместе с соответствующей нейронной сетью и геномом. Если труба выходит за пределы экрана, она добавляется в список для удаления. Если птица проходит мимо трубы, добавляется новая труба, и фитнес всех активных нейронных сетей увеличивается. Наконец, код удаляет трубы, которые вышли за пределы экрана. Этот процесс позволяет нейронным сетям обучаться и улучшать свою способность управлять птицами в игре, наказывая за столкновения и поощряя за успешное прохождение труб.

После уничтожения последней птицы на экране происходит завершение текущего поколения и подготовка к следующему поколению в процессе нейроэволюции. Когда последняя птица исчезает, программа делает паузу на 2 секунды, чтобы пользователь мог увидеть результаты текущего поколения. Затем программа выходит из основного игрового цикла, что означает завершение текущего поколения. После этого управление возвращается к функции ‘run_game’, которая была вызвана в контексте эволюционного процесса ‘NEAT’. В этот момент ‘NEAT’ анализирует фитнес всех нейронных сетей, которые управляли птицами в текущем поколении, и выбирает наиболее успешные сети для воспроизводства. Эти сети подвергаются мутациям и кроссинговеру, чтобы создать новую популяцию нейронных сетей для следующего поколения. Таким образом, процесс эволюции продолжается, и нейронные сети постепенно улучшают свою способность управлять птицами.

Результат выполнения нейронных сетей и нейроэволюции с использованием библиотеки ‘NEAT’ зависит от нескольких ключевых факторов. Параметры конфигурации, определенные в файле ‘config-feedforward.txt’, играют важную роль в процессе эволюции нейронных сетей. Эти параметры включают функции активации и агрегации, вероятности мутаций, параметры совместимости и воспроизводства. Структура нейронных сетей, включая количество узлов, соединений и их веса, также влияет на результат. Входные данные, такие как текущая высота птицы, расстояние до верхней и нижней трубы, а также расстояние до ближайшей трубы, позволяют нейронной сети принимать решения о прыжках. Логика оценки фитнеса включает время в игре, прохождение труб, столкновения и падения, а также частоту прыжков. Таким образом, результат выполнения нейронных сетей зависит от взаимодействия этих факторов, что влияет на процесс эволюции и улучшение способности нейронных сетей управлять птицами в игре.

Одним из важных параметров на результативность нейроэволюции является ‘pop_size’. Увеличение параметра ‘pop_size’ в конфигурационном файле ‘config-feedforward.txt’ приводит к улучшению управления птицами в игре, потому что это позволяет создать более разнообразную и большую популяцию нейронных сетей. Большая популяция обеспечивает большее разнообразие геномов, что увеличивает вероятность появления более успешных и адаптивных нейронных сетей. Это разнообразие позволяет

эволюционному процессу выбирать из большего числа вариантов, что способствует более быстрому и эффективному обучению.

С другой стороны, уменьшение параметра ‘pop_size’ приводит к ухудшению управления птицами, потому что меньшая популяция ограничивает разнообразие геномов. В маленькой популяции меньше шансов на появление успешных мутаций и комбинаций, что замедляет процесс эволюции и снижает общую эффективность нейронных сетей. Меньшее количество вариантов ограничивает возможности для выбора и воспроизводства наиболее успешных геномов, что приводит к менее адаптивным и менее успешным нейронным сетям.

Таким образом, увеличение параметра ‘pop_size’ способствует более разнообразной и адаптивной популяции нейронных сетей, что улучшает их способность управлять птицами в игре (рис. 6). На примере увеличения количества индивидуальных нейронных сетей до 100 единиц наблюдается результат, где на втором поколение птицы под управлением сетей преодолевают существенное расстояние.

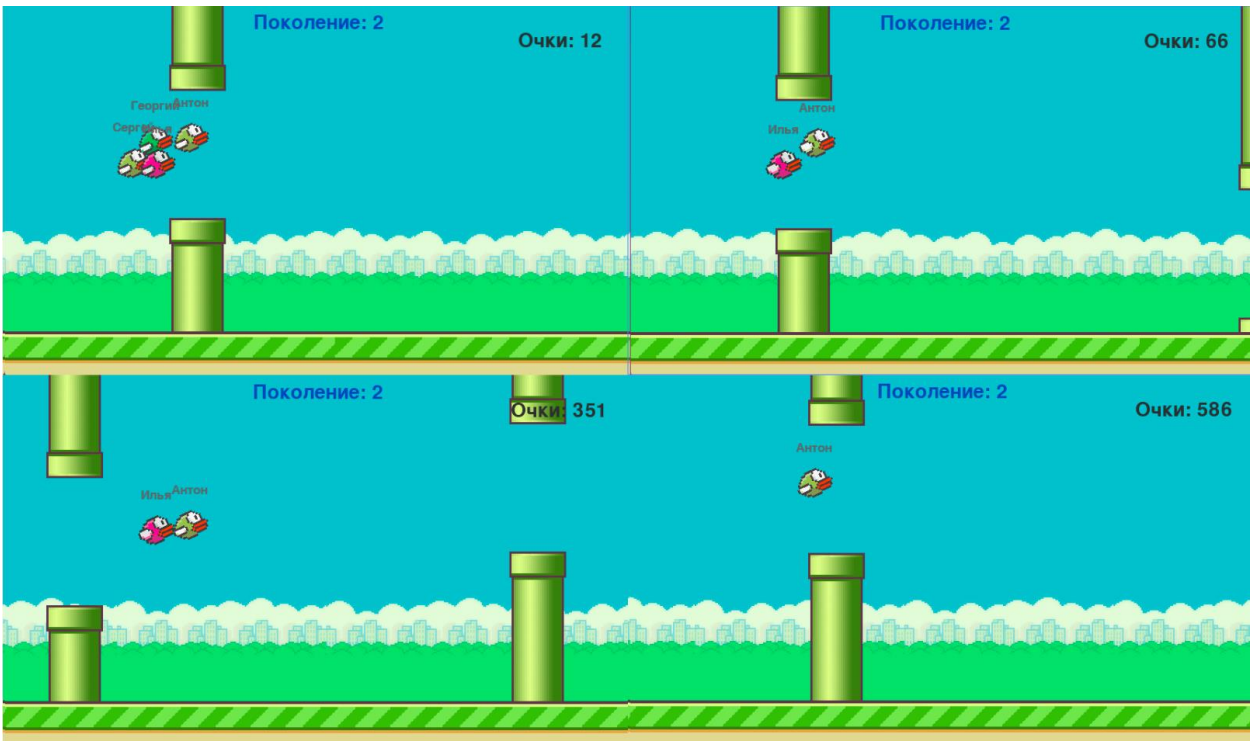


Рисунок 6. Результат нейроэволюции при значении ‘pop_size = 100’

Уменьшение этого параметра, наоборот, ограничивает разнообразие и снижает эффективность эволюционного процесса (рис. 7 и 8).

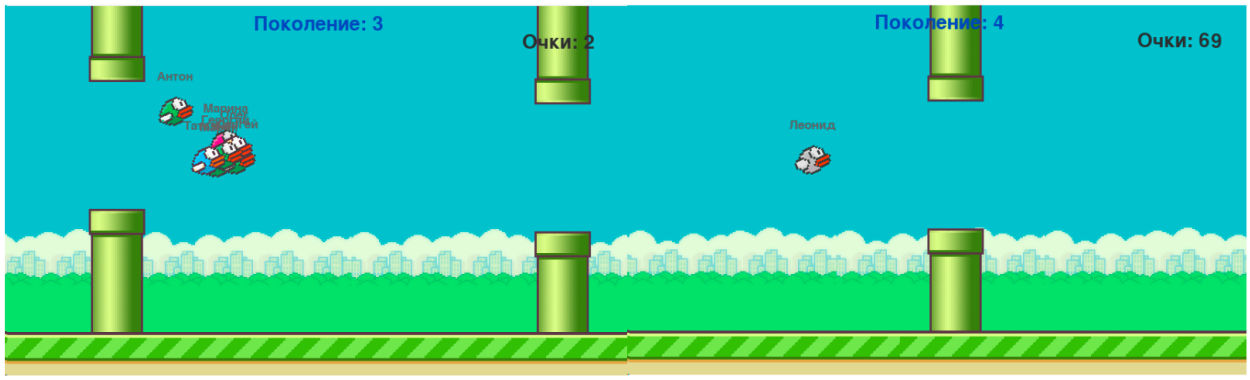


Рисунок 7. Результат нейроэволюции при значвении ‘pop_size = 50’

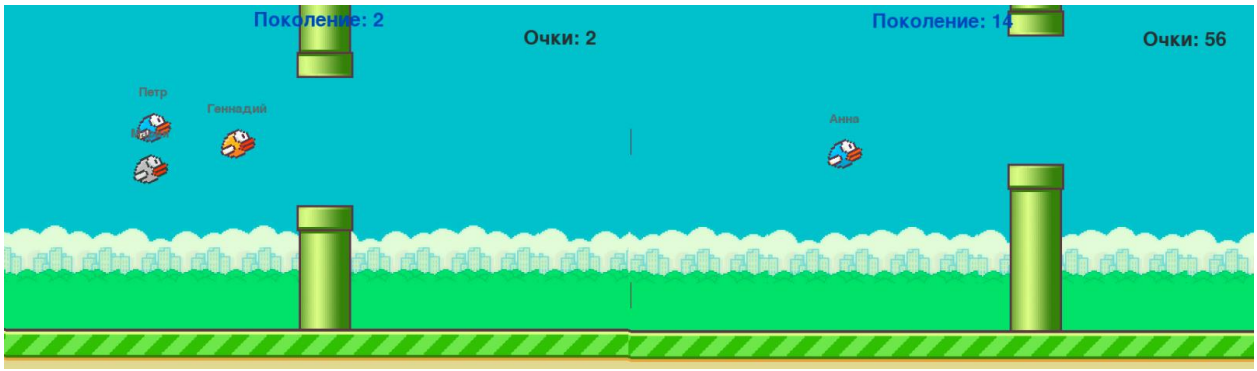


Рисунок 8. Результат нейроэволюции при значвении ‘pop_size = 10’

Вывод

В данной работе была исследована эффективность использования нейроэволюции для управления игровыми персонажами в игре с помощью библиотеки NEAT. Результаты показали, что увеличение размера популяции значительно улучшает управление персонажами, обеспечивая более разнообразные и адаптивные нейронные сети. Это разнообразие позволяет эволюционному процессу быстрее находить и улучшать успешные сети, что приводит к более высоким результатам в игре. Уменьшение размера популяции, наоборот, ограничивает разнообразие геномов и снижает эффективность эволюционного процесса, что приводит к менее адаптивным и менее успешным нейронным сетям. Таким образом, размер популяции является ключевым параметром, влияющим на эффективность нейроэволюции в контексте данной статьи.

Библиографический список

1. Рягузов А.В., Борисенков Д.В. Использование нейронной сети для игры в шахматы // В сборнике: Математика, информационные технологии, приложения. Сборник трудов Межвузовской научной конференции молодых ученых и студентов. Воронеж, 2024. С. 790-797.
2. Громова Г.М. Использование искусственной нейронной сети в игре жанра арканоид // В сборнике: Информационно-телекоммуникационные системы и технологии. Материалы Всероссийской научно-практической

конференции. 2018. С. 160-162.
3. Миндияров Н.И., Дороганов В.С. Программа обучения искусственного интеллекта при помощи нейронной сети на примере игры "гонки" // В сборнике: Материалы Всероссийской молодежной конференции "Информационно-телекоммуникационные системы и технологии (ИТСиТ-2012)". 2012. С. 132-133.
4. Гумерова Р.С. Построение нейросети средствами языка программирования Python на основе библиотек программных модулей // В сборнике: Математические методы и модели в исследовании современных проблем экономики и общества. Сборник статей Всероссийской молодежной научно-практической конференции. 2014. С. 19.
5. Пирматов А.З., Азимов Б.А., Камалов С.С. Искусственный интеллект с использованием Python: технологии и применение // Бюллетень науки и практики. 2023. Т. 9. № 11. С. 288-293.