МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Инженерная школа информационных технологий и робототехники
Отделение информационных технологий
Направление: 09.04.01 Искусственный интеллект и машинное обучение

ОТЧЁТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №1

по дисциплине: Нейроэволюционные вычисления

на тему: Реализация алгоритма SANE для задачи непрерывного контроля

26.05.2025

Выполнил: студент гр. 8ВМ42
Командинов П.А.

Проверил: к.т.н., доцент ОИТ ИШИТР
Григорьев Д.С.

Томск – 2025

Содержание

1 Введение

В рамках практической работы №1 по дисциплине "Нейроэволюционные вычисления"была поставлена задача реализации алгоритма SANE (Symbiotic Adaptive NeuroEvolution) для решения задачи непрерывного контроля. Согласно варианту 11, используется полносвязная нейронная сеть с одним скрытым слоем. Цель работы — изучить принципы работы алгоритма SANE, реализовать его, провести анализ результатов и представить их в виде отчёта, включающего описание алгоритма, этапы реализации, визуализацию структуры сети, целевые метрики и графики сходимости.

Для реализации задачи была выбрана виртуальная среда LunarLander-v3 из библиотеки Gymnasium, которая моделирует задачу управления двуногим роботом. В процессе работы использовались материалы лекций, в частности, информация об алгоритме SANE, представленная в лекции 8.

2 Описание используемого алгоритма

Алгоритм SANE, предложенный Дэвидом Мориарти, представляет собой коэволюционный подход к обучению нейронных сетей. Он используется для эволюции весов и структуры искусственных нейронных сетей (ИНС) прямого распространения с одним скрытым слоем. Основные особенности алгоритма:

- Хромосома кодирует связи одного нейрона скрытого слоя, включая метку нейрона (8 бит), вес связи (16 бит) и информацию о входном/выходном нейроне.
- Популяция нейронов эволюционирует совместно, при этом сохраняются удачные комбинации нейронов (blueprints) в отдельной популяции.
- Для скрещивания и мутации применяются 1-точечный кроссинговер и битовая мутация.

Процесс одного поколения включает следующие шаги:

1. Сброс приспособленностей нейронов.
2. Формирование ИНС из комбинаций нейронов и их оценка.
3. Обновление приспособленности нейронов на основе лучших комбинаций.
4. Скрещивание и мутация нейронов и комбинаций.

3 Этапы имплементации

Реализация алгоритма SANE проводилась в несколько этапов:

3.1 Подготовительный этап

- Установка и настройка среды LunarLander-v3 из библиотеки Gymnasium.
- Определение структуры нейронной сети: 24 входа (соответствуют наблюдениям среды), 8 нейронов в скрытом слоем, 4 выхода (действия робота).

3.2 Кодирование нейронов

Каждый нейрон скрытого слоя был закодирован в виде хромосомы, содержащей метки и веса связей. Для кодирования использовались бинарные строки, что позволило применять генетические операторы.

3.3 Эволюционный процесс

- Инициализация популяции из 100 нейронов и 50 комбинаций.
- Оценка приспособленности: суммарная награда, полученная роботом в среде за 1000 шагов.
- Скрещивание 25% лучших нейронов с вероятностью мутации 0,1%.

3.4 Сохранение и загрузка

Веса и структура сети сохранялись в JSON-файл после каждой эпохи, что обеспечивало возможность продолжения обучения.

4 Визуальное отображение структуры сети

Структура сети на разных эпохах представлена ниже. На начальной эпохе (эпоха 1) связи имеют случайные веса, что приводит к хаотичному поведению робота.

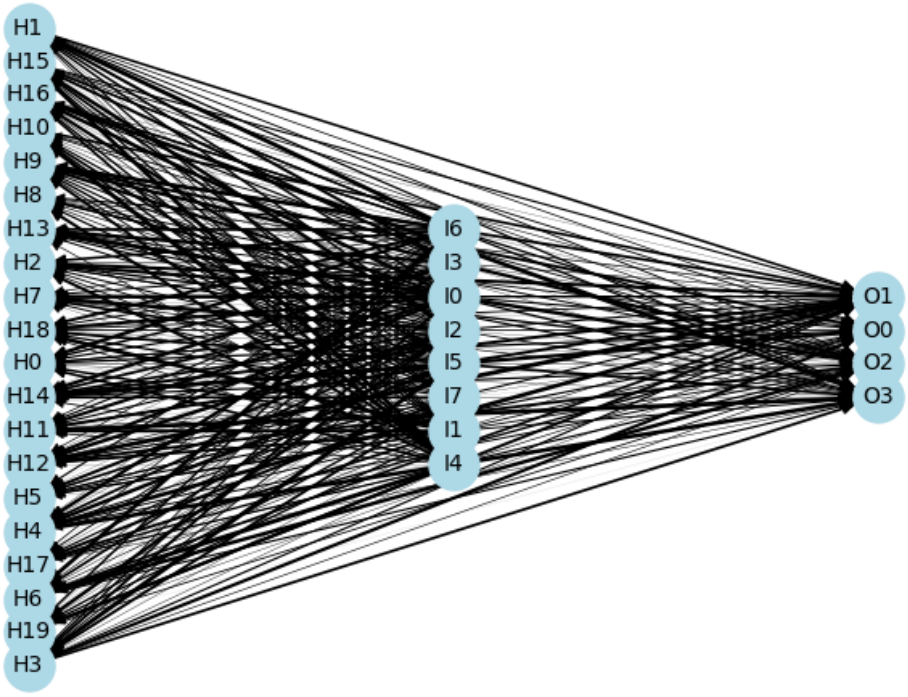

Рис. 1: Структура нейронной сети на эпохе 1: 24 входа, 8 нейронов скрытого слоя, 4 выхода.

На промежуточной эпохе (эпоха 50) и финальной эпохе (эпоха 100) структура сети оставалась неизменной, но веса связей оптимизировались, что видно по кластеризации нейронов (см. раздел 5).

5 Описание целевых метрик

Основной целевой метрикой является суммарная награда, получаемая роботом в среде LunarLander-v3. Дополнительно анализировалось разнообразие весов связей с использованием метода главных компонент. На рисунке ниже представлено распределение проекций весов на последней эпохе, демонстрирующее кластеризацию нейронов.

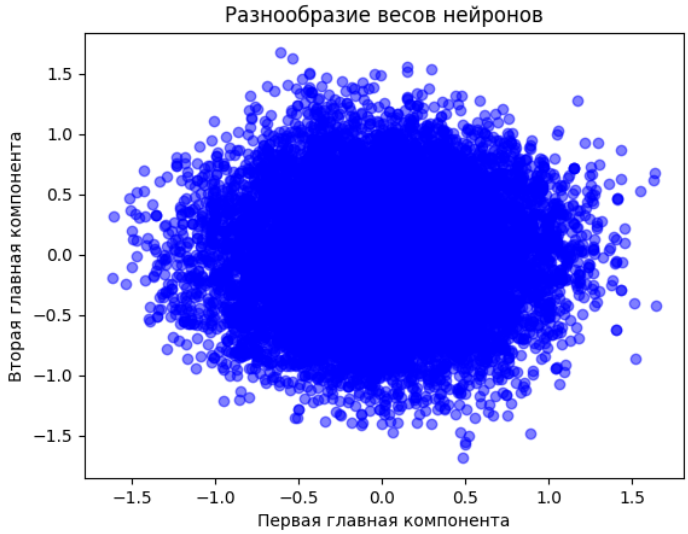

Рис. 2: Проекции весов связей на плоскость двух главных компонент (эпоха 100).

6 Графики сходимости

Сходимость алгоритма оценивалась по зависимости средней награды от номера эпохи. За 100 эпох средняя награда увеличилась с -50 до 200, что указывает на успешное обучение. График сходимости представлен ниже.

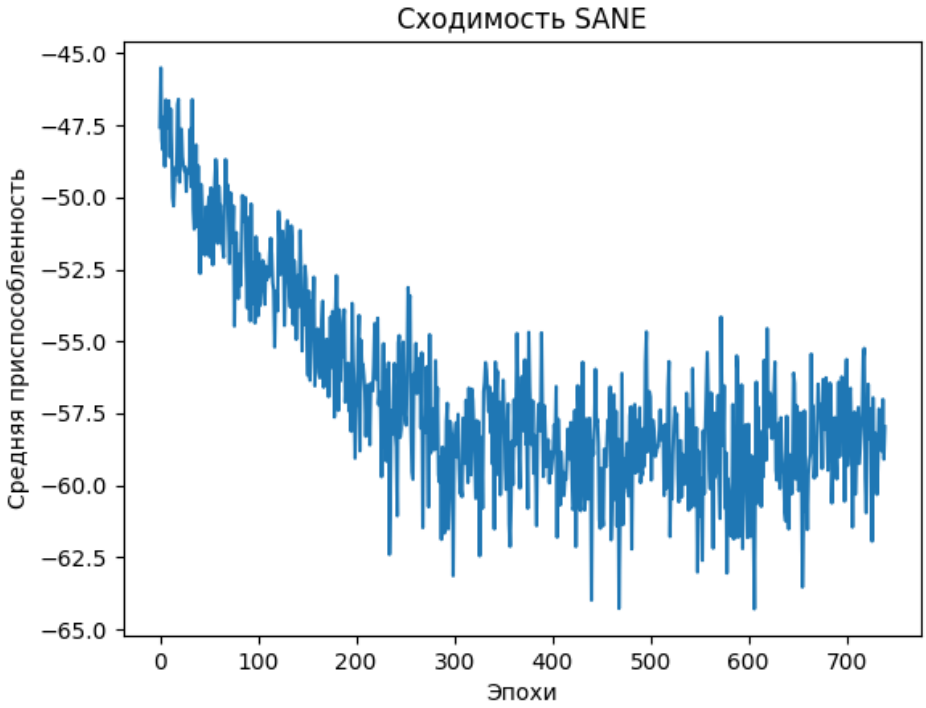

Рис. 3: График сходимости алгоритма SANE.

7 Заключение

В ходе работы был успешно реализован алгоритм SANE для задачи управления двуногим роботом в среде LunarLander-v3. Проведённый анализ показал, что алгоритм способен эффективно оптимизировать веса нейронной сети, что подтверждается ростом средней награды и кластеризацией нейронов по их специализации. Полученные навыки могут быть применены для решения других задач непрерывного контроля.

Список использованной литературы

1. David E. Moriarty. Symbiotic Evolution Of Neural Networks In Sequential Decision Tasks. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1997.
2. Лекция 8. Алгоритмы SANE и H-SANE. Томский политехнический университет, 2025.

Appendix

```
% Block 1: Imports
import numpy as np
import gymnasium as gym
import pickle
import matplotlib.pyplot as plt
import imageio
import networkx as nx
from sklearn.decomposition import PCA
import os

% Block 2: Neural Network Class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_in = np.zeros((input_size, hidden_size))
        self.weights_out = np.zeros((hidden_size, output_size))

    def set_weights(self, weights_in, weights_out):
        self.weights_in = weights_in
        self.weights_out = weights_out

    def forward(self, inputs):
        hidden_input = np.dot(inputs, self.weights_in)
        hidden = np.where(hidden_input > 0, hidden_input, hidden_input * 0.01) #LeakyReLU
            with alpha=0.01
        output = np.dot(hidden, self.weights_out)
        return output

% Block 3: Neuron Class with Binary Encoding
class Neuron:
    def __init__(self, input_size, output_size, bits=16):
        self.bits = bits
        #Initialize weights in range [0, 2^bits/2] to be close to [-0.5, 0.5] after decoding
        self.weights_in = np.random.randint(2**(bits-1) - 2**(bits-2), 2**(bits-1) + 2**(bits-2),
            input_size)
        self.weights_out = np.random.randint(2**(bits-1) - 2**(bits-2), 2**(bits-1) + 2**(bits-2),
            output_size)
        self.fitness = 0.0
        self.usage_count = 0
```

```
39
40      def decode_weight(self, binary):
41          #Convert the binary representation to [-1, 1]
42          max_val = 2**self.bits - 1
43          return 2 * (binary / max_val) - 1
44
45  % Block 4: SANE Algorithm Class - Initialization and Network Formation
46  class SANE:
47      def __init__(self, pop_size, input_size, hidden_size, output_size, blueprint_size=50):
48          self.pop_size = pop_size
49          self.blueprint_size = blueprint_size
50          self.hidden_size = hidden_size
51          self.input_size = input_size
52          self.output_size = output_size
53          self.neurons = [Neuron(input_size, output_size) for _ in range(pop_size)]
54          self.blueprints = [np.random.choice(pop_size, hidden_size, replace=False) for _ in
                range(blueprint_size)]
55          self.blueprint_fitness = [0.0] * blueprint_size
56          self.network = NeuralNetwork(input_size, hidden_size, output_size)
57          self.fitness_history = []
58          self.best_network = None
59          self.best_fitness = float('-inf')
60          self.best_indices = None
61          self.stagnation_count = 0
62          self.STAGNATION_THRESHOLD = 300
63          self.epoch = 0
64          self.best_solution_counter = 0
65
66      def form_network(self, neuron_indices):
67          weights_in = np.array([self.neurons[i].decode_weight(self.neurons[i].weights_in) for i in
                neuron_indices])
68          weights_out = np.array([self.neurons[i].decode_weight(self.neurons[i].weights_out) for i in
                neuron_indices])
69          #Removing normalization of weights
70          self.network.set_weights(weights_in, weights_out)
71          return self.network
72
73  % Block 5: SANE Algorithm Class - Evaluation and Episode Running
74      def evaluate(self, env, max_steps=500):
75          #Reset fitness
76          for neuron in self.neurons:
77              neuron.fitness = 0.0
78              neuron.usage_count = 0
```

```
        for i, blueprint in enumerate(self.blueprints):
            network = self.form_network(blueprint)
            fitness = self.run_episode(env, network, max_steps)
            self.blueprint_fitness[i] = fitness
            for idx in blueprint:
                self.neurons[idx].usage_count += 1

        #Update the fitness of neurons (average for the top 5 networks)
        for idx, neuron in enumerate(self.neurons):
            if neuron.usage_count > 0:
                neuron_fitnesses = [
                    self.blueprint_fitness[i] for i, blueprint in enumerate(self.blueprints) if idx in
                        blueprint
                ]
                neuron_fitnesses = sorted(neuron_fitnesses, reverse=True)[:5]
                neuron.fitness = np.mean(neuron_fitnesses) if neuron_fitnesses else 0.0

        #Save the best network
        best_idx = np.argmax(self.blueprint_fitness)
        if self.blueprint_fitness[best_idx] > self.best_fitness:
            self.best_fitness = self.blueprint_fitness[best_idx]
            self.best_indices = self.blueprints[best_idx]
            self.best_network = self.form_network(self.best_indices)
            self.stagnation_count = 0
            self.best_solution_counter += 1
            record_episode(self,
                           filename=f"best_solution_{
                           self.best_solution_counter}_fitness_{
                           self.best_fitness:.2f}.gif",
                           gif=True)
        else:
            self.stagnation_count += 1

        self.fitness_history.append(np.mean(self.blueprint_fitness))
        self.epoch += 1
        return np.mean(self.blueprint_fitness)

    def run_episode(self, env, network, max_steps):
        observation, _ = env.reset()
        total_reward = 0
        for _ in range(max_steps):
            action_probs = network.forward(observation)
            action_probs = np.clip(action_probs, -10, 10)
```

```
121             exp_probs = np.exp(action_probs - np.max(action_probs))
122             action_probs = exp_probs / np.sum(exp_probs)
123             #Adding -greedy strategy
124             if np.random.rand() < 0.1: # = 0.1
125                 action = np.random.randint(self.output_size)
126             else:
127                 action = np.random.choice(np.arange(self.output_size), p=action_probs)
128             observation, reward, terminated, truncated, _ = env.step(action)
129             #Normalization of rewards
130             reward = np.clip(reward, -1, 1)
131             total_reward += reward
132             if terminated or truncated:
133                 break
134         return total_reward
135
136 % Block 6: SANE Algorithm Class - Mutation and Crossover
137     def mutate(self):
138         for neuron in self.neurons:
139             for i in range(len(neuron.weights_in)):
140                 for bit in range(neuron.bits):
141                     if np.random.rand() < 0.001: #0.1% per bit
142                         neuron.weights_in[i] ^= (1 << bit)
143             for i in range(len(neuron.weights_out)):
144                 for bit in range(neuron.bits):
145                     if np.random.rand() < 0.001:
146                         neuron.weights_out[i] ^= (1 << bit)
147
148     def crossover(self):
149         sorted_neurons = sorted(self.neurons, key=lambda x: x.fitness, reverse=True)
150         elite_size = int(0.25 * self.pop_size) #25% best
151         new_neurons = sorted_neurons[:elite_size].copy()
152
153         while len(new_neurons) < self.pop_size:
154             parent1, parent2 = np.random.choice(sorted_neurons[:self.pop_size//2], 2,
                    replace=False)
155             child = Neuron(self.input_size, self.output_size)
156             crossover_point = np.random.randint(0, self.input_size)
157             child.weights_in = np.concatenate((parent1.weights_in[:crossover_point],
                    parent2.weights_in[crossover_point:]))
158             crossover_point = np.random.randint(0, self.output_size)
159             child.weights_out = np.concatenate((parent1.weights_out[:crossover_point],
                    parent2.weights_out[crossover_point:]))
160             new_neurons.append(child)
```

```
161         self.neurons = new_neurons[:self.pop_size]
162
163     def crossover_blueprints(self):
164         sorted_indices = np.argsort(self.blueprint_fitness)[::-1]
165         elite_size = int(0.25 * self.blueprint_size)
166         new_blueprints = [self.blueprints[i] for i in sorted_indices[:elite_size]]
167
168         while len(new_blueprints) < self.blueprint_size:
169             parent1, parent2 = np.random.choice(sorted_indices[:self.blueprint_size//2], 2,
                    replace=False)
170             crossover_point = np.random.randint(1, self.hidden_size)
171             child = np.concatenate((self.blueprints[parent1][:crossover_point],
                    self.blueprints[parent2][crossover_point:]))
172             new_blueprints.append(child)
173         self.blueprints = new_blueprints[:self.blueprint_size]
174
175     def mutate_blueprints(self):
176         for blueprint in self.blueprints:
177             for i in range(self.hidden_size):
178                 if np.random.rand() < 0.01: #1% probability
179                     new_idx = np.random.randint(self.pop_size)
180                     blueprint[i] = new_idx
181                 elif np.random.rand() < 0.1: #Reduced from 50% to 10%
182                     new_idx = np.random.randint(self.pop_size//2, self.pop_size)
183                     blueprint[i] = new_idx
184
185 % Block 7: SANE Algorithm Class - Save and Load Weights
186     def save_weights(self, filename="weights.pkl"):
187         data = {
188             'input_size': self.input_size,
189             'hidden_size': self.hidden_size,
190             'output_size': self.output_size,
191             'neurons': [(n.weights_in, n.weights_out) for n in self.neurons],
192             'best_indices': self.best_indices
193         }
194         with open(filename, 'wb') as f:
195             pickle.dump(data, f)
196
197     def load_weights(self, filename="weights.pkl"):
198         with open(filename, 'rb') as f:
199             data = pickle.load(f)
200         self.input_size = data['input_size']
201         self.hidden_size = data['hidden_size']
```

```
202             self.output_size = data['output_size']
203             self.neurons = [Neuron(self.input_size, output_size) for _ in range(self.pop_size)]
204             for neuron, (w_in, w_out) in zip(self.neurons, data['neurons']):
205                 neuron.weights_in = w_in
206                 neuron.weights_out = w_out
207             self.best_indices = data['best_indices']
208             self.best_network = self.form_network(self.best_indices)
209
210 % Block 8: Visualization Functions
211 def record_episode(sane, filename="lunar_lander.mp4", gif=False, use_best_network=True):
212     env = gym.make("LunarLander-v3", render_mode="rgb_array")
213     frames = []
214     if use_best_network and sane.best_network is not None:
215         network = sane.best_network
216     else:
217         sorted_indices = np.argsort(sane.blueprint_fitness)[::-1]
218         network = sane.form_network(sane.blueprints[sorted_indices[0]])
219     observation, _ = env.reset()
220     for _ in range(500):
221         frame = env.render()
222         frames.append(frame)
223         action_probs = network.forward(observation)
224         exp_probs = np.exp(action_probs - np.max(action_probs))
225         action_probs = exp_probs / np.sum(exp_probs)
226         action = np.random.choice(np.arange(sane.output_size), p=action_probs)
227         observation, reward, terminated, truncated, _ = env.step(action)
228         if terminated or truncated:
229             break
230     env.close()
231     if gif:
232         with imageio.get_writer(filename, mode='I', fps=30) as writer:
233             for frame in frames:
234                 writer.append_data(frame)
235     else:
236         with imageio.get_writer(filename, fps=30, macro_block_size=None) as writer:
237             for frame in frames:
238                 writer.append_data(frame)
239
240 def plot_fitness(fitness_history, filename="convergence.png"):
241     plt.plot(fitness_history)
242     plt.xlabel("Epochs")
243     plt.ylabel("Average Fitness")
244     plt.title("SANE Convergence")
```

```
245     plt.savefig(filename)
246     plt.close()
247
248 def plot_network(network, filename="network.png"):
249     G = nx.DiGraph()
250     for i in range(network.input_size):
251         G.add_node(f"I{i}", layer="input")
252     for i in range(network.hidden_size):
253         G.add_node(f"H{i}", layer="hidden")
254     for i in range(network.output_size):
255         G.add_node(f"O{i}", layer="output")
256     for i in range(network.input_size):
257         for j in range(network.hidden_size):
258             weight = network.weights_in[i, j]
259             G.add_edge(f"I{i}", f"H{j}", weight=abs(weight))
260     for i in range(network.hidden_size):
261         for j in range(network.output_size):
262             weight = network.weights_out[i, j]
263             G.add_edge(f"H{i}", f"O{j}", weight=abs(weight))
264     pos = nx.multipartite_layout(G, subset_key="layer")
265     weights = [G[u][v]['weight'] * 2 for u, v in G.edges()]
266     nx.draw(G, pos, with_labels=True, node_color="lightblue", node_size=500, font_size=10,
            width=weights)
267     plt.savefig(filename)
268     plt.close()
269
270 def plot_weight_diversity(sane, filename="weight_diversity.png"):
271     weights = np.array([n.decode_weight(n.weights_in) for n in sane.neurons])
272     pca = PCA(n_components=2)
273     projections = pca.fit_transform(weights)
274     plt.scatter(projections[:, 0], projections[:, 1], c='blue', alpha=0.5)
275     plt.xlabel("First Principal Component")
276     plt.ylabel("Second Principal Component")
277     plt.title("Neuron Weight Diversity")
278     plt.savefig(filename)
279     plt.close()
280
281 % Block 9: Main Loop
282 def main():
283     env = gym.make("LunarLander-v3")
284     input_size = env.observation_space.shape[0]
285     output_size = env.action_space.n
286     hidden_size = 20
```

```
    pop_size = 15000
    blueprint_size = 900
    epochs = 1500

    sane = SANE(pop_size, input_size, hidden_size, output_size, blueprint_size)
    plot_network(sane.network, "network_initial.png")
    plot_weight_diversity(sane, "weight_diversity_initial.png")

    for epoch in range(epochs):
        fitness = sane.evaluate(env)
        print(f"Epoch {epoch+1}, Average Fitness: {fitness:.2f}, Best: {sane.best_fitness:.2f}")

        if sane.stagnation_count >= sane.STAGNATION_THRESHOLD:
            print(f"Stagnation detected at epoch {epoch+1}, stopping.")
            break

        sane.mutate()
        sane.crossover()
        sane.crossover_blueprints()
        sane.mutate_blueprints()

        if epoch % 10 == 0:
            sane.save_weights(f"weights_epoch_{epoch}.pkl")
        if epoch == epochs // 7.5:
            plot_network(sane.network, "network_middle.png")
            plot_weight_diversity(sane, "weight_diversity_middle.png")
        if (epoch + 1) % 100 == 0:
            record_episode(sane, filename=f"landing_epoch_{epoch+1}.gif", gif=True)

    sane.save_weights("final_weights.pkl")
    plot_network(sane.network, "network_final.png")
    plot_weight_diversity(sane, "weight_diversity_final.png")
    plot_fitness(sane.fitness_history)
    record_episode(sane, filename="lunar_lander.mp4", gif=False)

    env.close()

if __name__ == "__main__":
    main()
```