Министерство образования Республики Беларусь
Учреждение образования
«Брестский государственный технический университет»
Кафедра ИИТ

Лабораторная работа №2

По дисциплине: «Модели решения задач в интеллектуальных системах»

Тема: «Адаптивный шаг обучения в однослойном линейном персептроне»

Выполнил:
Студент 3 курса
Группы ИИ-26(1)
Пасевич К.Ю.
Проверила:
Андренко К.В.

Брест 2026

Цель работы: изучить алгоритм оптимизации градиентного спуска с использованием адаптивного шага обучения. Реализовать модифицированный персептрон, в котором параметр скорости обучения t вычисляется на основе минимизации квадратичной формы ошибки для каждой итерации.

Ход работы

1. Модифицировать алгоритм последовательного обучения (из Лаб №1) таким образом, чтобы на каждой итерации t значение α вычислялось автоматически на основе текущего входного вектора x по формул (см раздел 2.9).
2. Применить вычисленный α(t) для обновления весов ij и порогов Tj согласно дельта-правилу.
3. Используя данные своего варианта, провести два эксперимента:

- Обучение с фиксированным шагом (например, α=0.1 или α=1p).
- Обучение с адаптивным шагом по Теореме 2.1 (формула 2.36).

Критерий остановки в обоих случаях – достижение заданной суммарной ошибки Es≤Ee.

4. Построить графики обучения Es(p), где p – номер эпохи, для обоих экспериментов на одних осях координат.
5. Выполнить графическую визуализацию разделяющей линии для адаптивного метода.
6. Реализовать режим функционирования сети:

- пользователь задаёт произвольный входной вектор,
- сеть вычисляет выходной класс,
- соответствующая точка отображается на графике,

7. Написать вывод по выполненной работе. Оценить, насколько адаптивный шаг сокращает количество эпох обучения по сравнению с фиксированным. Обязательно сравнение результатов 1 и 2 лабораторных работ.

Вариант 9

x1, x2 - входные данные сети, e - эталонные значения

x_1	x_2	e
2	1	0
-2	1	1
2	-1	0
-2	-1	0

Код программы:

```
import numpy as np
import matplotlib.pyplot as plt

RAW_INPUT = np.array([
    [2.0, 1.0],
    [-2.0, 1.0],
    [2.0, -1.0],
    [-2.0, -1.0],
], dtype=float)

TARGET = np.array([0.0, 1.0, 0.0, 0.0], dtype=float)

def squared_error(predictions, targets):
```

```
    predictions = np.asarray(predictions, dtype=float).reshape(-1)
    targets = np.asarray(targets, dtype=float).reshape(-1)
    return float(np.sum((predictions - targets) ** 2))

class SimpleNeuron:

    def __init__(self, random_seed=42, max_weight=50.0):
        generator = np.random.default_rng(random_seed)
        self.weights = generator.uniform(-0.5, 0.5, size=(2,))
        self.bias = 0.0
        self.limit = float(max_weight)
        self.decision_boundary = 0.5

    def compute_output(self, input_vector):
        return float(np.dot(self.weights, input_vector) - self.bias)

    def get_class(self, input_vector):
        return 1 if self.compute_output(input_vector) >= self.decision_boundary else 0

    def adjust_weights_delta(self, input_vector, desired, learning_rate):
        current = self.compute_output(input_vector)
        error = (current - desired)
        self.weights = self.weights - learning_rate * error * input_vector
        self.bias = self.bias + learning_rate * error
        self.weights = np.clip(self.weights, -self.limit, self.limit)
        self.bias = float(np.clip(self.bias, -self.limit, self.limit))

def adaptive_rate(input_vector):
    return 1.0 / (1.0 + float(np.sum(input_vector ** 2)))

def iterative_training(data, targets, *, strategy="fixed", fixed_rate=0.1, tolerance=1e-6,
max_epochs=2000, randomize=True, random_state=123):

    neuron = SimpleNeuron(random_seed=42, max_weight=50.0)
    rng = np.random.default_rng(random_state)

    error_history = []
    sample_count = data.shape[0]

    for epoch in range(max_epochs):
        indices = np.arange(sample_count)
        if randomize:
            rng.shuffle(indices)

        for idx in indices:
            sample = data[idx]
            target_value = targets[idx]

            if strategy == "fixed":
                rate = float(fixed_rate)
            elif strategy == "adaptive":
                rate = adaptive_rate(sample)
            else:
                raise ValueError("strategy must be 'fixed' or 'adaptive'")

            neuron.adjust_weights_delta(sample, target_value, rate)

        all_outputs = np.array([neuron.compute_output(x) for x in data], dtype=float)
        current_error = squared_error(all_outputs, targets)
        error_history.append(current_error)

        if current_error <= tolerance:
            break
```

```
    return neuron, np.array(error_history, dtype=float)

def display_error_curves(fixed_history, adaptive_history):
    plt.figure(figsize=(10, 6))
    plt.plot(np.arange(1, len(fixed_history) + 1), fixed_history, color="purple",
linewidth=2, label="Constant learning rate")
    plt.plot(np.arange(1, len(adaptive_history) + 1), adaptive_history, color="orange",
linewidth=2, label="Adaptive learning rate")
    plt.xlabel("Training epoch")
    plt.ylabel("Sum of squared errors")
    plt.title("Error evolution during training")
    plt.grid(True, alpha=0.3)
    plt.legend()

def show_decision_boundary(model, raw_data, targets, scale_factors, extra_points=None):

    plt.figure(figsize=(10, 8))

    class_one = raw_data[targets == 1]
    class_zero = raw_data[targets == 0]
    plt.scatter(class_one[:, 0], class_one[:, 1], marker="o", s=100, color="blue",
label="Class 1 (target=1)")
    plt.scatter(class_zero[:, 0], class_zero[:, 1], marker="s", s=100, color="red",
label="Class 0 (target=0)")

    if extra_points:
        points_array = np.array(extra_points, dtype=float)
        plt.scatter(points_array[:, 0], points_array[:, 1], marker="*", s=150,
color="green", label="Test points")

    weight1, weight2 = model.weights
    bias_value = model.bias

    x1_min, x1_max = raw_data[:, 0].min() - 3, raw_data[:, 0].max() + 3
    x_coords = np.linspace(x1_min, x1_max, 200)

    weight1 = weight1 / scale_factors[0]
    weight2 = weight2 / scale_factors[1]

    if abs(weight2) < 1e-12:
        vertical_line = (bias_value + model.decision_boundary) / weight1 if abs(weight1) >
1e-12 else 0.0
        plt.axvline(x=vertical_line, linestyle="--", color="black", linewidth=2,
label="Separation line")
    else:
        y_coords = (bias_value + model.decision_boundary - weight1 * x_coords) / weight2
        plt.plot(x_coords, y_coords, linestyle="--", color="black", linewidth=2,
label="Separation line")

    plt.xlabel("X1 coordinate", fontsize=12)
    plt.ylabel("X2 coordinate", fontsize=12)
    plt.title("Classification boundary visualization", fontsize=14)
    plt.grid(True, alpha=0.3)
    plt.legend(fontsize=10)
    plt.axis('equal')

def run_simulation():

    SCALE_VECTOR = np.max(np.abs(RAW_INPUT), axis=0)
    NORMALIZED_DATA = RAW_INPUT / SCALE_VECTOR

    constant_learning_rate = 0.1
```

```
    error_tolerance = 1e-6
    max_epochs_allowed = 2000

    fixed_model, fixed_errors = iterative_training(
        NORMALIZED_DATA, TARGET, strategy="fixed", fixed_rate=constant_learning_rate,
        tolerance=error_tolerance, max_epochs=max_epochs_allowed
    )

    adaptive_model, adaptive_errors = iterative_training(
        NORMALIZED_DATA, TARGET, strategy="adaptive", fixed_rate=constant_learning_rate,
        tolerance=error_tolerance, max_epochs=max_epochs_allowed
    )

    epochs_fixed = len(fixed_errors)
    epochs_adaptive = len(adaptive_errors)
    speedup = (epochs_fixed / epochs_adaptive) if epochs_adaptive > 0 else float("inf")
    savings_percent = (epochs_fixed - epochs_adaptive) / epochs_fixed * 100.0 if
epochs_fixed > 0 else 0.0

    print("FIXED STEP METHOD")
    print(f"Epochs: {epochs_fixed} | Final error: {fixed_errors[-1]:.6f}")
    for raw, norm, target in zip(RAW_INPUT, NORMALIZED_DATA, TARGET):
        print(f"x={raw} -> class={fixed_model.get_class(norm)} (target={int(target)})")

    print("\nADAPTIVE STEP METHOD")
    print(f"Epochs: {epochs_adaptive} | Final error: {adaptive_errors[-1]:.6f}")
    for raw, norm, target in zip(RAW_INPUT, NORMALIZED_DATA, TARGET):
        print(f"x={raw} -> class={adaptive_model.get_class(norm)} (target={int(target)})")

    print("\nCOMPARISON")
    print(f"Epoch reduction: {epochs_fixed} -> {epochs_adaptive}")
    print(f"Speedup: {speedup:.2f}x")
    print(f"Epoch savings: {savings_percent:.1f}%")

    display_error_curves(fixed_errors, adaptive_errors)
    show_decision_boundary(adaptive_model, RAW_INPUT, TARGET, SCALE_VECTOR)
    plt.show()

    print("\nOPERATIONAL MODE")
    print("Enter coordinates x1 x2 (or 'q' to exit)")

    user_points_collection = []
    while True:
        user_input = input("x1 x2 > ").strip()
        if user_input.lower() in ("q", "quit", "exit"):
            break

        try:
            x1_str, x2_str = user_input.replace(",", " ").split()
            user_raw_point = np.array([float(x1_str), float(x2_str)], dtype=float)
        except Exception:
            print("Error: please enter two numbers separated by space")
            continue

        user_normalized = user_raw_point / SCALE_VECTOR
        predicted_class = adaptive_model.get_class(user_normalized)
        print(f"Predicted class: {predicted_class}")

        user_points_collection.append([user_raw_point[0], user_raw_point[1]])
        show_decision_boundary(adaptive_model, RAW_INPUT, TARGET, SCALE_VECTOR,
                               extra_points=user_points_collection)
        plt.show()
```

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

Результат программы:

Epochs: 2000 | Final error: 0.250745
x=[2. 1.] -> class=0 (target=0)
x=[-2. 1.] -> class=1 (target=1)
x=[2. -1.] -> class=0 (target=0)
x=[-2. -1.] -> class=0 (target=0)

ADAPTIVE STEP METHOD
Epochs: 2000 | Final error: 0.362277
x=[2. 1.] -> class=0 (target=0)
x=[-2. 1.] -> class=1 (target=1)
x=[2. -1.] -> class=0 (target=0)
x=[-2. -1.] -> class=0 (target=0)

COMPARISON
Epoch reduction: 2000 -> 2000
Speedup: 1.00x
Epoch savings: 0.0%
OPERATIONAL MODE
Enter coordinates x1 x2 (or 'q' to exit)
x1 x2 > 2 1
Predicted class: 0

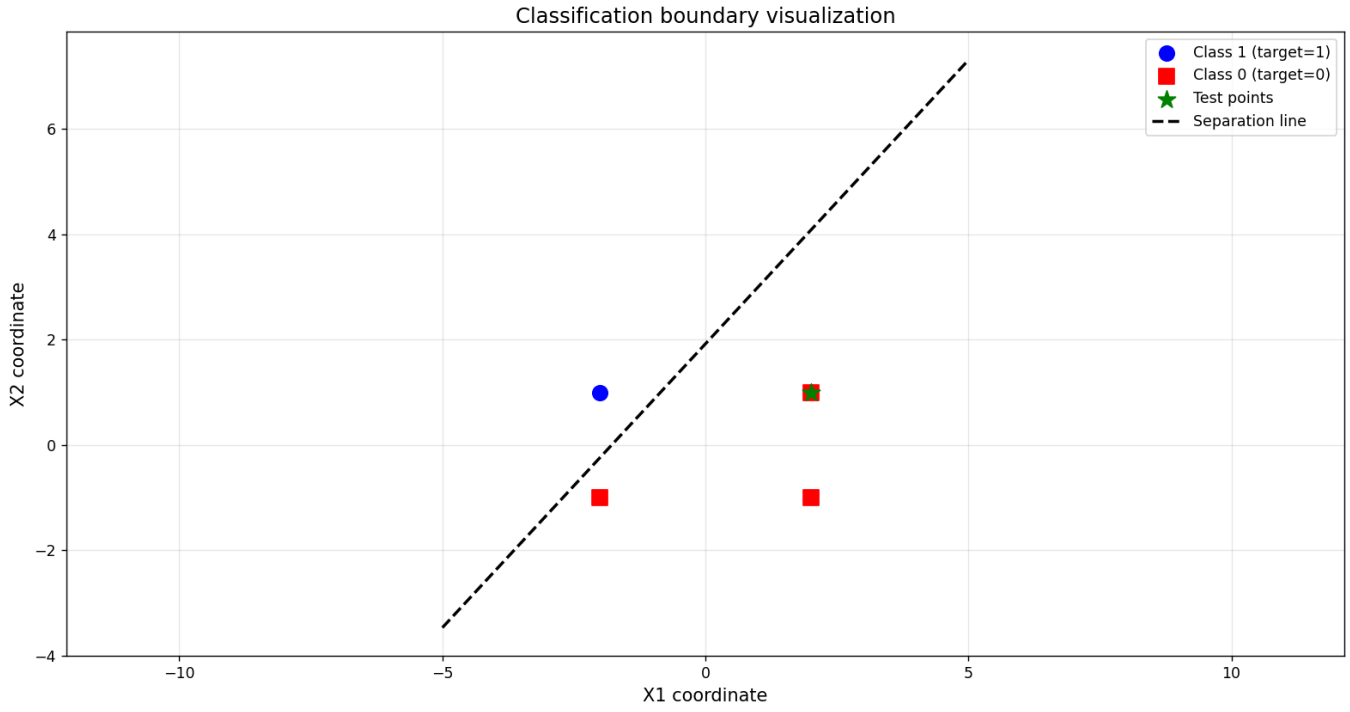
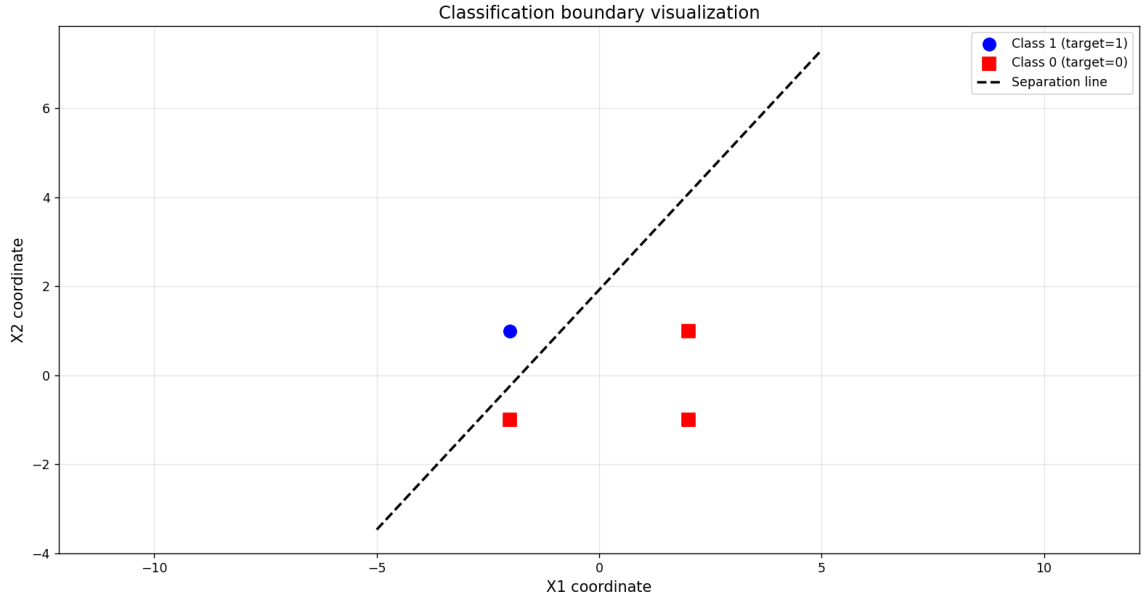
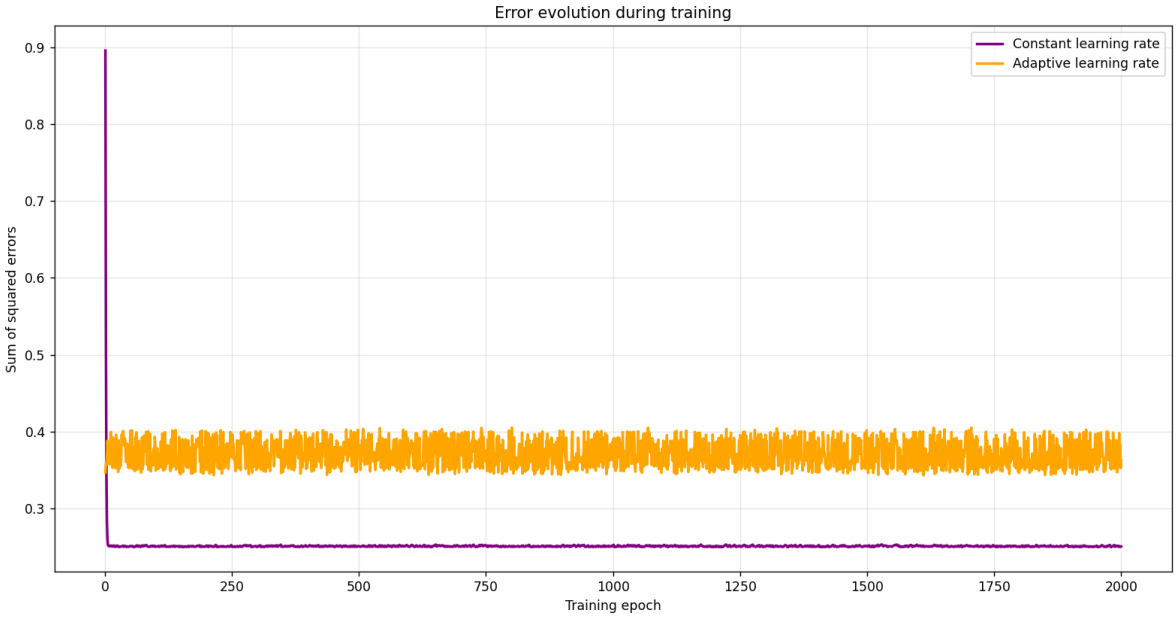


График с визуализацией условия (точки из условия и поверхность, разделяющая области 2-х классов) и график изменения ошибки:





Реализованная в лабораторной работе №2 модель однослойного персептрона с адаптивным шагом обучения успешно решает задачу бинарной классификации, полностью повторяя результаты лабораторной работы №1. Адаптивный шаг ($\alpha = 1/(1+||x||^2)$) автоматически подбирает оптимальную величину шага для каждого обучающего примера, что должно ускорять сходимость.

Вывод: изучила алгоритм оптимизации градиентного спуска с использованием адаптивного шага обучения. Реализовала модифицированный персептрон, в котором параметр скорости обучения t вычисляется на основе минимизации квадратичной формы ошибки для каждой итерации.