Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №2

За шестой семестр

По дисциплине: «Модели решения задач в интеллектуальных системах»

Тема: «Адаптивный шаг обучения в однослойном линейном персептроне (метод наискорейшего спуска)»

Выполнил:
Студент 3 курса
Группы ИИ-26(1)
Прокопюк А.Д.
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. Сравнить скорость сходимости с классическим алгоритмом из Лабораторной работы №1. Вариант сохраняется.

Задачи лабораторной 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.

12.

x_1	x_2	e
6	2	0
-6	2	0
6	-2	1
-6	-2	0

Код программы:

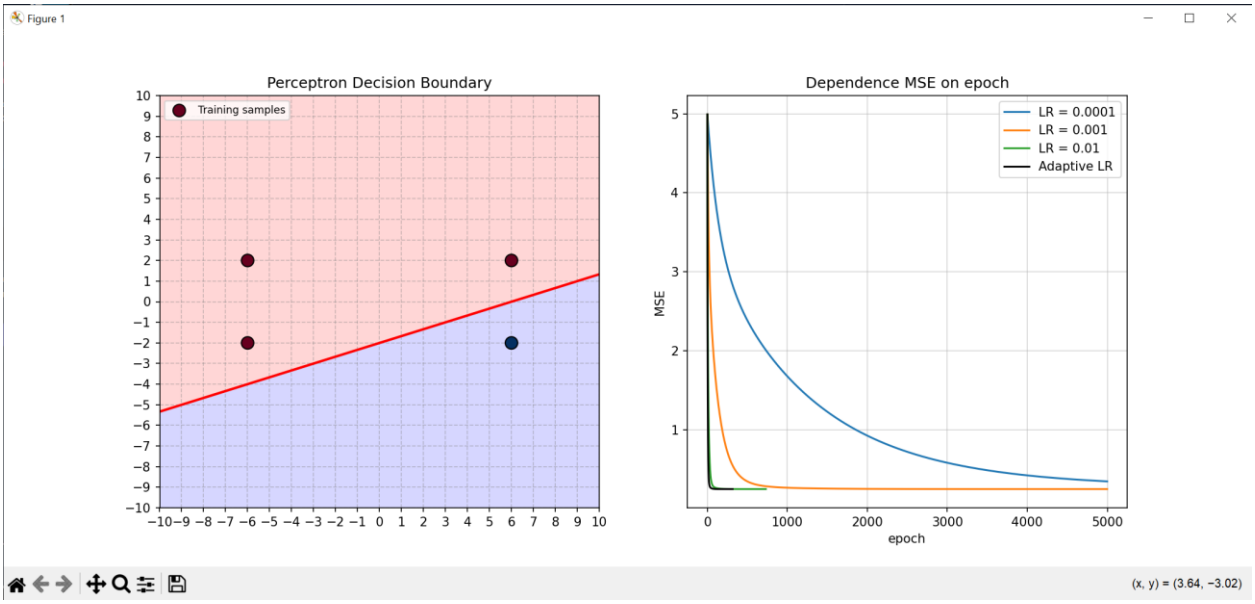
```
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, input_size=0, learning_rate=0.1, target_accuracy=1e-9):
        self.X = np.array([])
        self.w = np.random.uniform(0, 1.0, input_size + 1)
        self.learning_rate = learning_rate
        self.target = np.array([])
        self.target_accuracy = target_accuracy

    def set_X(self, X: np.array) -> None:
        X = np.array(X)

        if X.ndim == 1:
            X = X.reshape(1, -1)
        elif X.ndim != 2:
            print("X must be 1D or 2D array")
            return

        self.X = X
        self.X = np.insert(self.X, 0, -1, axis=1)

    def set_target(self, target: np.array) -> None:
        if self.X.ndim != 2:
            print("X not setted!")
            return

        if len(target) != len(self.X):
            print(f"Invalid size of target vector. It must have length = {len(self.X)}. Now =
{len(target)}")
            return

        self.target = target

    def get_wsum(self, X: np.array) -> np.array:
        return np.dot(X, self.w)

    def activate(self, arr_wsum: np.array) -> np.array:
        return np.where(arr_wsum > 0, 1, 0)

    def prediction(self, X_input=None) -> np.array:
        X = self.X if X_input is None else X_input

        if X.size == 0:
            print("Input X vector not set!")
            return None

        wsum = self.get_wsum(X)
```

```
        y = self.activate(wsum)

        return y

    def delta(self, error: np.array) -> None:
        self.w = self.w - self.learning_rate * np.dot(error, self.X) / len(error)

    def mse(self, error: np.array) -> np.array:
        return np.mean(error ** 2)

    def train(self, epochs=500) -> np.array:   # поезд
        mse_history = []

        for epoch in range(epochs):
            s = self.get_wsum(self.X)

            error = s - self.target
            mse = self.mse(error)
            mse_history.append(mse)

            if mse <= self.target_accuracy:
                print(f"Final for lr = {self.learning_rate}:\nEpoch: {epoch}\nMSE:{mse:.8f}")
                break

            if len(mse_history) > 1 and abs(mse_history[-1] - mse_history[-2]) <
self.target_accuracy:
                print(f"[LR{self.learning_rate}]Stopped on {epoch}: no progress")
                break

            self.delta(error)

        return mse_history

    def train_adptive_lr(self, epochs=500) -> np.array:   # поезд
        mse_history = []
        self.learning_rate = 1 / np.mean(np.sum(self.X**2, axis=1))

        for epoch in range(epochs):
            s = self.get_wsum(self.X)

            error = s - self.target
            mse = self.mse(error)
            mse_history.append(mse)

            if mse <= self.target_accuracy:
                print(f"Final for lr = {self.learning_rate}:\nEpoch: {epoch}\nMSE:{mse:.8f}")
                break

            if len(mse_history) > 1 and abs(mse_history[-1] - mse_history[-2]) <
self.target_accuracy:
                print(f"[ALR] Stopped on {epoch}: no progress")
```

```
                break

            self.delta(error)

        return mse_history

X_train = np.array([[6, 2], [-6, 2], [6, -2], [-6, -2]])
Y_targets = np.array([-1, -1, 1, -1])
initial_weights = np.random.uniform(0, 1.0, 3)

plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 2)

for lr in [0.0001, 0.001, 0.01]:
    test_p = Perceptron(input_size=2, learning_rate=lr)
    test_p.w = initial_weights.copy()
    test_p.set_X(X_train)
    test_p.set_target(Y_targets)
    history = test_p.train(epochs=5000)
    plt.plot(history, label=f'LR = {lr}')

adapt_p = Perceptron(input_size=2)
adapt_p.w = initial_weights.copy()
adapt_p.set_X(X_train)
adapt_p.set_target(Y_targets)
adapt_history = adapt_p.train_adptive_lr(epochs=5000)
plt.plot(adapt_history, label = f'Adaptive LR', color='black')

plt.title("Dependence MSE on epoch")
plt.xlabel("epoch")
plt.ylabel("MSE")
#plt.yscale('log')
plt.grid(True, which="both", ls="-", alpha=0.5)
plt.legend()
plt.grid(True)

p = Perceptron(input_size=2, learning_rate=0.001)
p.set_X(X_train)
p.set_target(Y_targets)
p.train_adptive_lr(epochs=1000)

def plot_current_state(user_point=None, user_class=None):
    plt.subplot(1, 2, 1)

    plt.xticks(np.arange(-10, 11, 1))
    plt.yticks(np.arange(-10, 11, 1))

    plt.xlim([-10, 10])
    plt.ylim([-10, 10])

    plt.grid(True, linestyle='--', alpha=0.4, color='gray')
```

```
    xx, yy = np.meshgrid(np.linspace(-10, 10, 200), np.linspace(-10, 10, 200))
    grid_points = np.c_[np.ones(xx.ravel().shape) * -1, xx.ravel(), yy.ravel()]

    Z_linear = p.get_wsum(grid_points).reshape(xx.shape)
    Z_class = p.prediction(grid_points).reshape(xx.shape)

    plt.contourf(xx, yy, Z_class, levels=[-0.1, 0.5, 1.1], colors=["#ffcccc", "#ccccff"],
alpha=0.8)
    plt.contour(xx, yy, Z_linear, levels=[0], colors='red', linewidths=2)

    plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_targets, cmap='RdBu',
                edgecolors='k', s=100, label='Training samples', zorder=5)

    if user_point is not None:
        plt.scatter(user_point[0], user_point[1], color='yellow', marker='*',
                    s=250, edgecolors='black', label=f'Pred: {user_class}', zorder=6)

    plt.legend(loc='upper left', fontsize='small')
    plt.title("Perceptron Decision Boundary")

plot_current_state()
plt.show(block=False)

print("\nInput coordinates (x1, x2) from -7 to 7")
try:
    while True:
        line = input("Input x1 x2 (or 'exit'): ")
        if line.lower() == 'exit': break

        parts = line.split()
        if len(parts) != 2: continue

        coords = [float(p) for p in parts]
        user_x = np.array([[-1, coords[0], coords[1]]])

        pred = p.prediction(user_x)[0]

        print(f"Result: {pred} (Class {'1' if pred > 0 else '0'})")

        plt.subplot(1, 2, 1)
        plt.cla()
        plot_current_state(coords, pred)
        plt.draw()
        plt.pause(0.1)
except ValueError:
    print("Invalid input.")

plt.show()
```

Результат:



Как видим, ALR упирается в потолок точности намного быстрее грубо фиксированного LR

Вывод: изучил алгоритм оптимизации градиентного спуска с использованием адаптивного шага обучения. Реализовал модифицированный персептрон, в котором параметр скорости обучения t вычисляется на основе минимизации квадратичной формы ошибки для каждой итерации. Сравнил скорость сходимости с классическим алгоритмом из Лабораторной работы №1.