Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

Лабораторная работа №2

По дисциплине «Методы решения задач в И С»

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

Выполнил:

Студент 3 курса

Группы ИИ-26

Ковальчук А. И.

Проверил:

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.

Постановка задачи:

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 7

x_1	x_2	e
4	6	0
-4	6	1
4	-6	1
-4	-6	1

Код программы:

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


class DenseLayer:
    def __init__(self, units=1, activation='relu'):
        self.units = units
        self.activation = activation.lower()
        self.w = None
        self.b = None
        self.input = None
        self.z = None

    def forward(self, x):
        self.input = x

        if self.w is None:
            fan_in = x.shape[-1]
            if self.activation in ['relu', 'leaky_relu']:
                std = np.sqrt(2.0 / fan_in)
            else:
                std = np.sqrt(1.0 / fan_in)
            self.w = np.random.normal(0.0, std, (fan_in, self.units))
            self.b = np.zeros(self.units)

        self.z = x @ self.w + self.b

        if self.activation == 'relu':
```

```
            return np.maximum(0, self.z)
        elif self.activation == 'leaky_relu':
            return np.maximum(0.01 * self.z, self.z)
        elif self.activation == 'sigmoid':
            return 1 / (1 + np.exp(-self.z))
        elif self.activation == 'tanh':
            return np.tanh(self.z)
        elif self.activation == 'softmax':
            exp_z = np.exp(self.z - np.max(self.z, axis=1, keepdims=True))
            return exp_z / np.sum(exp_z, axis=1, keepdims=True)
        elif self.activation == 'linear':
            return self.z
        elif self.activation == 'step':
            return (self.z >= 0).astype(float)
        else:
            raise ValueError(f"Неизвестная активация: {self.activation}")

    def derivative(self, a):
        if self.activation == 'relu':
            return (self.z > 0).astype(float)
        elif self.activation == 'leaky_relu':
            return (self.z > 0).astype(float) + 0.01 * (self.z <= 0).astype(float)
        elif self.activation == 'sigmoid':
            return a * (1 - a)
        elif self.activation == 'tanh':
            return 1 - a**2
        elif self.activation in ('linear', 'step'):
            return np.ones_like(a)
```

```
        elif self.activation == 'softmax':
            return np.ones_like(a)
        return np.ones_like(a)

class Input:
    def __init__(self, shape=None):
        self.shape = shape

    def forward(self, x):
        if self.shape is not None:
            expected = self.shape if isinstance(self.shape, tuple) else (self.shape,)
            if x.shape[1:] != expected:
                x = x.reshape((x.shape[0],) + expected)
        return x

class Sequential:
    def __init__(self, layers):
        self.layers = layers
        self.history_mse = []

    def forward(self, x):
        out = x
        for layer in self.layers:
            out = layer.forward(out)
        return out

    def fit(self, x_input, y_input, epochs=100, alpha=0.001,
            clip_value=5.0, adaptive_alpha=False, Ee=1e-5):
```

```
x_input = np.asarray(x_input, dtype=np.float32)
y_input = np.asarray(y_input, dtype=np.float32).reshape(-1, 1)

n_samples = x_input.shape[0]
self.history_mse = []

for epoch in range(epochs):
    mse_sum = 0.0

    indices = np.random.permutation(n_samples)
    x_shuffled = x_input[indices]
    y_shuffled = y_input[indices]

    for i in range(n_samples):
        x = x_shuffled[i:i+1]
        y = y_shuffled[i:i+1]

        if adaptive_alpha:
          alpha_t = 1.0 / (1 + np.sum(x ** 2))
        else:
          alpha_t = alpha

        activations = [x]
        for layer in self.layers:
            a = layer.forward(activations[-1])
            activations.append(a)

        pred = activations[-1]
```

```
mse_sum += np.mean((pred - y) ** 2)

delta = pred - y

if self.layers[-1].activation == 'step':
    delta = y - pred

for l in range(len(self.layers) - 1, -1, -1):
    layer = self.layers[l]

    if not hasattr(layer, 'activation'):
        continue

    a_prev = activations[l]

    if layer.activation == 'step':
        grad_w = a_prev.T @ delta
        grad_b = np.sum(delta, axis=0)
    else:
        da = layer.derivative(activations[l+1])
        delta = delta * da
        grad_w = a_prev.T @ delta
        grad_b = np.sum(delta, axis=0)

    grad_w = np.clip(grad_w, -clip_value, clip_value)
    grad_b = np.clip(grad_b, -clip_value, clip_value)

    layer.w += alpha_t * grad_w
```

```
                layer.b += alpha_t * grad_b

                delta = delta @ layer.w.T

        avg_mse = mse_sum / n_samples
        self.history_mse.append(avg_mse)

        if epoch % 3 == 0 or epoch == epochs - 1:
            print(f"Epoch {epoch:4d} | MSE = {avg_mse:.6f}")

        if avg_mse <= Ee:
          print(f'Достигнут критерий остановки: {avg_mse} <= {Ee}")
          break

def predict(self, x):
    x = np.asarray(x, dtype=np.float32)
    if x.ndim == 1:
        x = x.reshape(1, -1)
    return self.forward(x)
```

Графики обучения Es(p), где p – номер эпохи, для обоих экспериментов на одних осях координат:

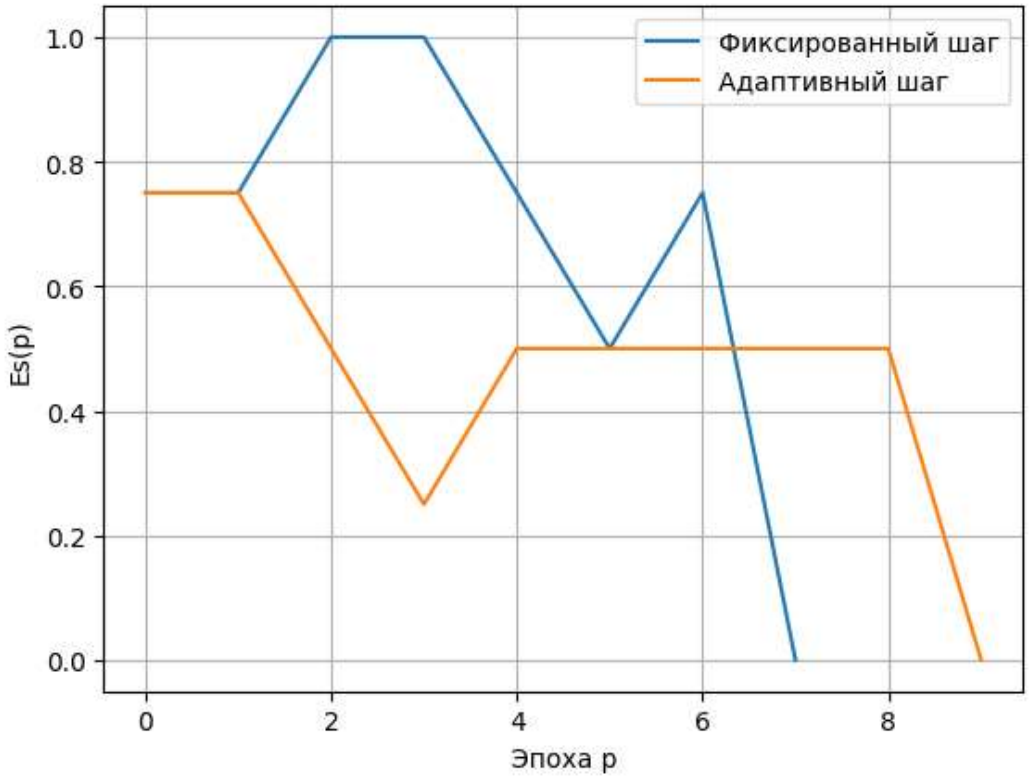


График разделяющей прямой для персептрона с адаптивным шагом:

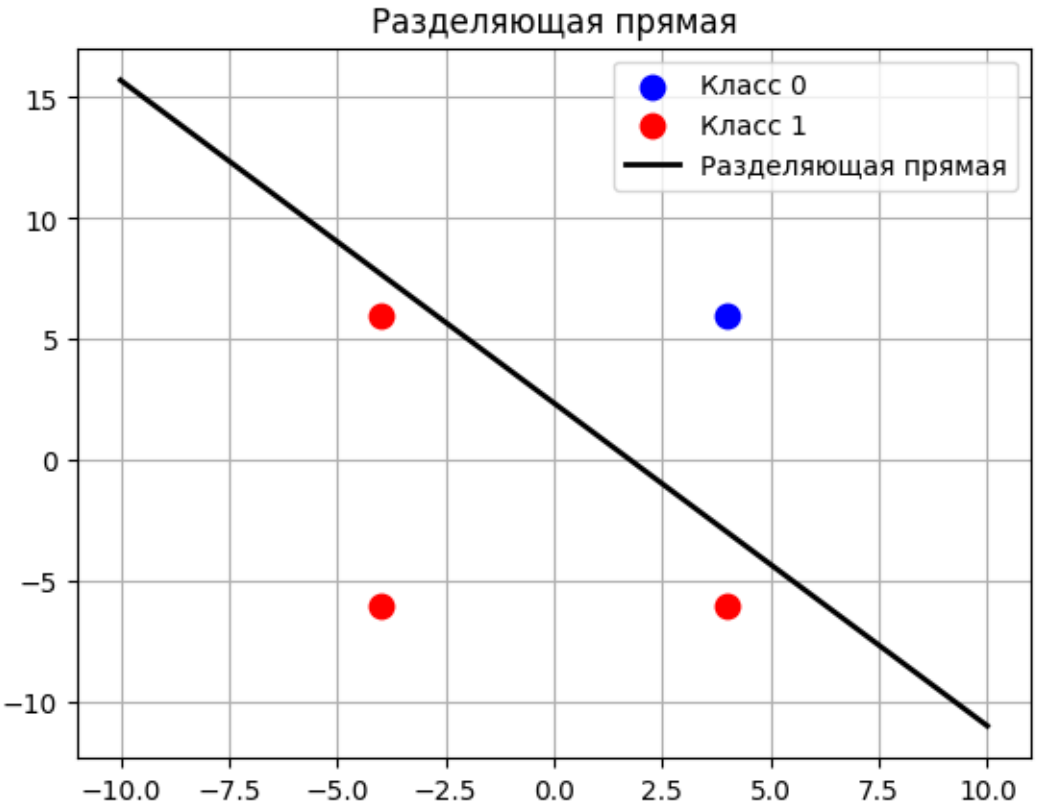
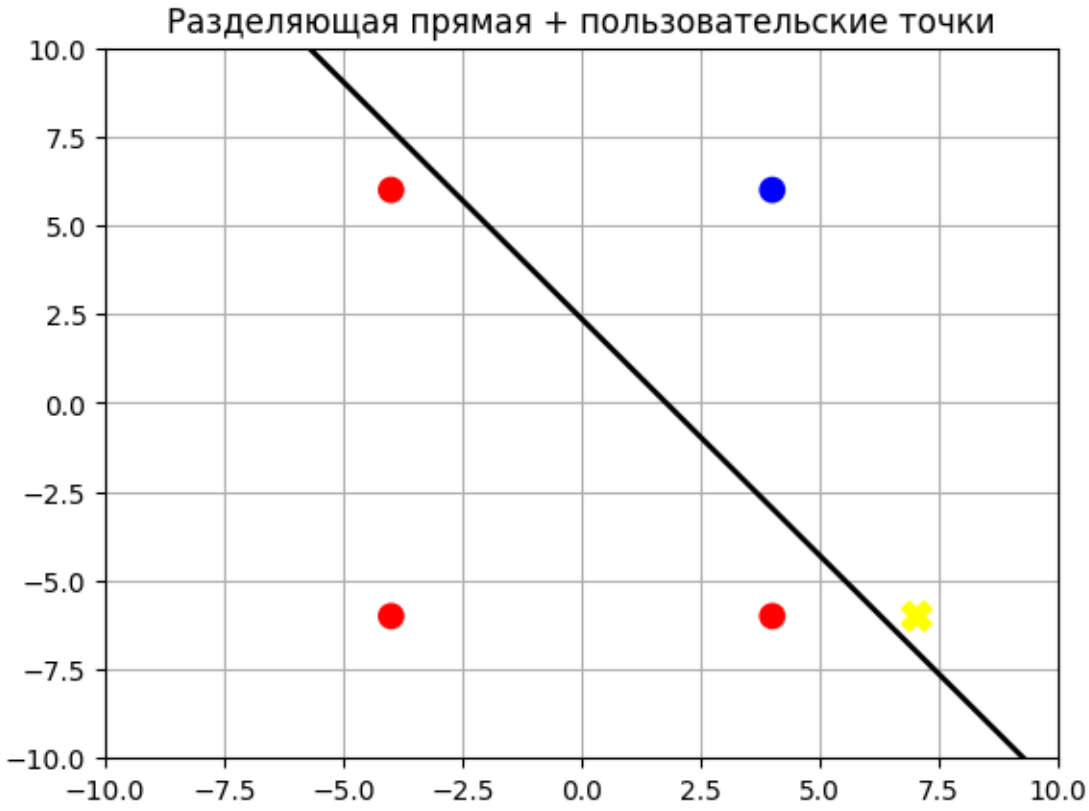


График с введёнными пользователем точками:



Вывод: Для ускорения процедуры обучения градиентного спуска вместо постоянного шага обучения можно использовать адаптивный шаг. Адаптивным называется шаг обучения, который выбирается на каждом этапе алгоритма таким образом, чтобы минимизировать квадратичную ошибку сети.