Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

Лабораторная работа №6

По дисциплине «МРЗИС»

Тема: “Использование адаптивного шага и batch-learning. Метод градиентного бустнига”

Выполнил:

Студент 3 курса

Группы ИИ-21

Романко Н.А.

Проверил:

Туз И.С.

Брест 2023

Цель:реализовать использование адаптивного шага в нейронной сети и использовать batch-learning и сравнить с работой градиентного бустинга

Код персептрона:

import numpy as np

from sklearn.datasets import load\_wine

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

class Perceptron:

    def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate="adapt"):

        self.input\_size = input\_size

        self.hidden\_size = hidden\_size

        self.output\_size = output\_size

        self.learning\_rate = learning\_rate

        self.weights\_input\_hidden = np.random.randn(self.input\_size, self.hidden\_size)

        self.bias\_hidden = np.zeros((1, self.hidden\_size))

        self.weights\_hidden\_output = np.random.randn(self.hidden\_size, self.output\_size)

        self.bias\_output = np.zeros((1, self.output\_size))

    def sigmoid(self, x):

        return 1 / (1 + np.exp(-x))

    def sigmoid\_derivative(self, x):

        return x \* (1 - x)

    def forward(self, inputs):

        self.hidden\_input = np.dot(inputs, self.weights\_input\_hidden) + self.bias\_hidden

        self.hidden\_output = self.sigmoid(self.hidden\_input)

        self.output = np.dot(self.hidden\_output, self.weights\_hidden\_output) + self.bias\_output

        return self.sigmoid(self.output)

    def backward(self, inputs, targets, outputs):

        if self.learning\_rate == "adapt":

            error = targets - outputs

            delta\_output = error \* self.sigmoid\_derivative(outputs)

            self.learning\_rate\_output = (4 \* np.sum((error \*\* 2 \* outputs \* (1 - outputs)), axis=0)) / (1 + np.sum(self.hidden\_output \*\* 2)) \* (np.sum((error \* outputs \* (1 - outputs)) \*\* 2, axis=0))

            h\_error = delta\_output.dot(self.weights\_hidden\_output.T)

            self.learning\_rate\_hidden = (4 \* np.sum((h\_error \*\* 2 \* self.hidden\_output \* (1 - self.hidden\_output)), axis=0)) / (1 + np.sum(self.hidden\_input \*\* 2)) \* (np.sum((h\_error \* self.hidden\_output \* (1 - self.hidden\_output)) \*\* 2, axis=0))

            delta\_hidden = delta\_output.dot(self.weights\_hidden\_output.T) \* self.sigmoid\_derivative(self.hidden\_output)

            self.weights\_hidden\_output += self.hidden\_output.T.dot(delta\_output) \* self.learning\_rate\_output

            self.bias\_output += np.sum(delta\_output, axis=0, keepdims=True) \* self.learning\_rate\_output

            self.weights\_input\_hidden += inputs.T.dot(delta\_hidden) \* self.learning\_rate\_hidden

            self.bias\_hidden += np.sum(delta\_hidden, axis=0, keepdims=True) \* self.learning\_rate\_hidden

    def train(self, inputs, targets, epochs, batch\_size):

        for epoch in range(epochs):

            for i in range(0, len(inputs), batch\_size):

                input\_data = inputs[i:i+batch\_size]

                target\_data = targets[i:i+batch\_size]

                output = self.forward(input\_data)

                self.backward(input\_data, target\_data, output)

                loss = np.mean(np.square(target\_data - output))

            if not epoch % 100:

                print(f'Epoch {epoch + 1}, Learning rate {self.learning\_rate\_output}, Loss: {loss}')

    def predict(self, inputs):

        output = self.forward(inputs)

        return output

data = load\_wine()

X = data.data

y = data.target.reshape(-1, 1)

scaler = StandardScaler()

X = scaler.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)

input\_size = X\_train.shape[1]

hidden\_size = 5

output\_size = y\_train.shape[1]

perceptron = Perceptron(input\_size, hidden\_size, output\_size)

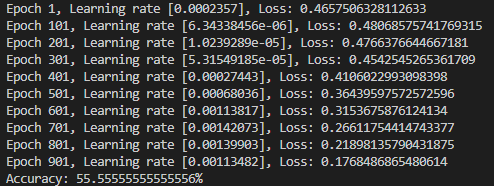
perceptron.train(X\_train, y\_train, epochs=1000, batch\_size=5)

predictions = perceptron.predict(X\_test)

accuracy = np.mean(np.round(predictions) == y\_test)

print(f'Accuracy: {accuracy \* 100}%')

Вывод программы:



Код градиентного бустинга:

import numpy as np

from sklearn.datasets import load\_wine

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

import xgboost as xgb

class GradientBoostingClassifier:

    def \_\_init\_\_(self, n\_estimators=100, learning\_rate=0.1, max\_depth=3):

        self.n\_estimators = n\_estimators

        self.learning\_rate = learning\_rate

        self.max\_depth = max\_depth

        self.model = None

    def train(self, X\_train, y\_train):

        self.model = xgb.XGBClassifier(n\_estimators=self.n\_estimators, learning\_rate=self.learning\_rate, max\_depth=self.max\_depth)

        self.model.fit(X\_train, y\_train)

    def predict(self, X\_test):

        return self.model.predict(X\_test)

data = load\_wine()

X = data.data

y = data.target

scaler = StandardScaler()

X = scaler.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)

gradient\_boosting = GradientBoostingClassifier()

gradient\_boosting.train(X\_train, y\_train)

predictions = gradient\_boosting.predict(X\_test)

accuracy = np.mean(predictions == y\_test)

print(f'Accuracy: {accuracy \* 100}%')

Вывод программы:



Вывод: научился реализовывать и применять на практике адаптивный шаг обучения и batch-learning. Сравнил скорость выполнения и точность классификации с методом градиентного бустинга.