Министерство образования Республики Беларусь

Учреждение образования

“Брестский государственный технический университет”

Кафедра интеллектуально-информационных технологий

Лабораторная работа №5

“ MLP. Прогнозирование”

Выполнил:

студент 3 курса

группы ИИ-22

Копанчук Е. Р.

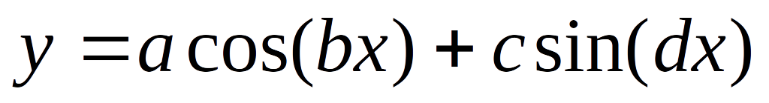
Проверил:

Туз И. С.

Брест 2023

Написать нейронную сеть(multilayer perceptron c одним скрытым слоем) для

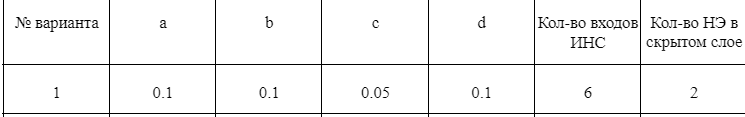
решения задачи прогнозирования функции:



Обучить сеть с использованием константного и адаптивного шага обучения, online-

learning и batch-learning. Результаты для каждого варианта сети занести в таблицу(

test error, количество эпох, время обучения и тд)



from matplotlib import pyplot as plt

import numpy as np

from scipy.special import expit

E\_arr\_arr = []

E\_arr = []

f = lambda x : 0.1 \* np.cos(0.5 \* x) + 0.09 \* np.sin(0.5 \* x)

def normalize\_data(data, min\_val, max\_val) :

min\_data = np.min(data)

max\_data = np.max(data)

normalized\_data = (data - min\_data) / (max\_data - min\_data) \* (max\_val - min\_val) + min\_val

return normalized\_data

class Network :

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate = 0.01) :

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.learning\_rate = learning\_rate

self.start\_learn\_rate = learning\_rate

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

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

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

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

def sigmoid(self, x) :

return expit(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.output

def predict(self, inputs) :

output = self.forward(inputs)

result = [el[0] for el in output]

return normalize\_data(result, -0.13446, 0.13446)

def backward(self, inputs, target, output) :

error = target - output

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

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

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

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

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

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

error = targets - outputs

mse\_batch = np.sum(error) / len(error)

inputs\_cut = 0

for j in range(len(error)) : inputs\_cut += error[j] \* inputs[j]

mse\_batch = np.array(mse\_batch)

inputs\_cut = np.array(inputs\_cut).reshape(1, -1)

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

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

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

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

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

def train(self, inputs, targets, epochs: int, isAdapt : bool = False) :

global E\_arr

for epoch in range(epochs) :

e\_arr = []

for i in range(len(inputs)) :

input\_data = np.array([inputs[i]])

target\_data = np.array([targets[i]])

output = self.forward(input\_data)

e\_arr.append(targets[i] - output)

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

E2 = np.sum(np.array(e\_arr) \* \*2) / 2

E\_arr.append(E2)

self.learning\_rate = min(0.2, max(self.learning\_rate \* E2, 0.0046)) if isAdapt else self.learning\_rate

#print(f"Online: Epoch: {epoch} MSE: {E2}")

def trainBatch(self, inputs, targets, epochs: int, batchsize : int, isAdapt : bool = False) :

global E\_arr

if (len(inputs) % batchsize != 0) : return ValueError

inputspack = [inputs[i - batchsize:i] for i in range(batchsize, len(inputs), batchsize)]

targetspack = [targets[i - batchsize:i] for i in range(batchsize, len(targets), batchsize)]

for epoch in range(epochs) :

e\_arr = []

for i in range(len(inputspack)) :

outputs = [self.forward(batchElem).item() for batchElem in inputspack[i]]

e\_arr.append(np.mean(targetspack[i] - outputs))

self.backwardBatch(inputspack[i], targetspack[i], outputs)

E2 = np.sum(np.array(e\_arr) \* \*2) / 2

E\_arr.append(E2)

self.learning\_rate = min(0.2, max(self.learning\_rate \* E2, 0.0046)) if isAdapt else self.learning\_rate

#print(f"Batch: Epoch: {epoch} MSE: {E2}")

def get\_train\_data(all\_points, input\_size) :

result\_X = [all\_points[i:i + input\_size] for i in range(len(all\_points) - input\_size)]

result\_Y = [all\_points[i] for i in range(input\_size, len(all\_points), 1)]

return np.array(result\_X), np.array(result\_Y)

input\_size = 8

hidden\_size = 3

output\_size = 1

epochs = 100

all\_train\_points = f(np.arange(100, 200, 0.2))

all\_test\_points = f(np.arange(180, 280, 0.2))

X\_train, Y\_train = get\_train\_data(all\_train\_points, input\_size)

X\_test, Y\_test = get\_train\_data(all\_test\_points, input\_size)

def cutVer(E\_arr, epochs, all\_train\_points, predicted, start, end) :

print(f"Time taken: {(end-start)\*10\*\*3:.03f}ms")

print(min(E\_arr))

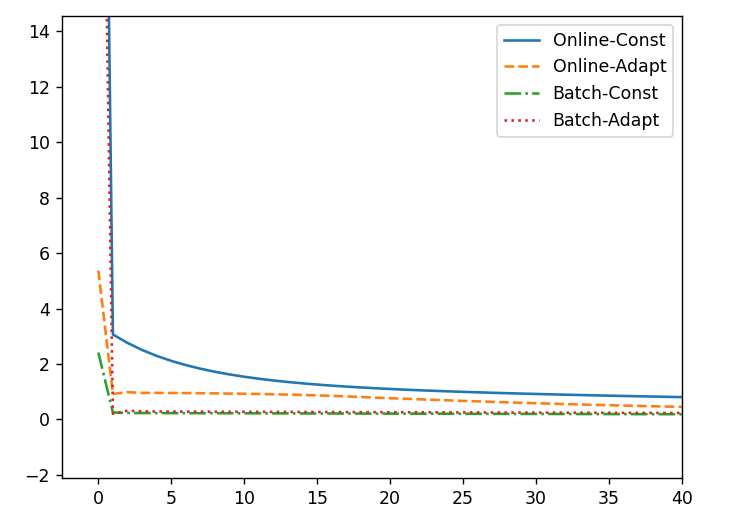
plt.plot(range(0, epochs), E\_arr)

plt.show()

plt.plot(np.arange(100, 200, 0.2), all\_train\_points)

plt.plot(np.arange(181.6, 280, 0.2), predicted, ":")

plt.show()



Вывод: изучил обучение и функционирование ИНС при решении задач прогнозирование.