Міністерство освіти і науки України

Національний технічний університет "ХПІ"

кафедра "Інформатика та інтелектуальна власність"

**ЗВІТ**

до лабораторної роботи № 5

Тема : «Багатошаровий перцептрон для розпізнавання графічних символів бібліотеки Fashion MNIST»

Варіант номер 18

з дисципліни "Основи штучного інтелекту"

Виконав:

студент групи КН-321В

Хома Д.М.

Перевірив:

Паржин Ю. В.

Харків 2023

**1 Побудова нейромережі Network1 для бібліотеки MNIST**

Вивчено алгоритм та розглянуто код програми Network1. Отримані результати класифікації рукописних цифр наведені на рис. 1.1.

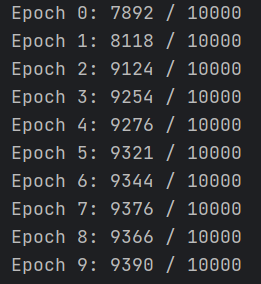


Рисунок 1.1 – Результати класифікації Network1

Отже цифри класифікуються з точністю в 94%.

**2 Завантаження бібліотеки Fashion MNIST**

Я обрав варіант завантаження зображень з бібліотеки шляхом імпортування її з пакету keras. На рис. 1.2 показано імпортування зображень та як зображення та мітки з набору даних Fashion MNIST перетворюються в формат, придатний для навчання мережі. Зображення змінюються, щоб бути векторами розмірністю 784, і далі значення пікселів нормалізуються до діапазону [0, 1] шляхом ділення на 255.

import numpy as np  
from tensorflow import keras  
  
  
def load\_data\_wrapper():  
 (train\_images, train\_labels), (test\_images, test\_labels) = keras.datasets.fashion\_mnist.load\_data()  
  
 #Змінення розмірності зображень у вхідні дані для навчання  
 training\_inputs = [np.reshape(x, (784, 1)) / 255.0 for x in train\_images]  
 #Нормалізація значень пікселів до діапазону [0, 1]  
 training\_results = [vectorized\_result(y) for y in train\_labels]  
 training\_data = list(zip(training\_inputs, training\_results))  
  
 # Перетворення набору даних тестування  
 test\_inputs = [np.reshape(x, (784, 1)) / 255.0 for x in test\_images]  
 test\_data = list(zip(test\_inputs, test\_labels))  
  
 return training\_data, test\_data  
  
def vectorized\_result(j):  
 e = np.zeros((10, 1))  
 e[j] = 1.0  
 return e

Рисунок 1.2 – Імпортування та нормалізація вхідних та тестових даних

**3 Код програми Network2**

Створено аналогічну програму **Network2** для розпізнавання набору даних **Fashion MNIST** з використанням активаційної функції **ReLu** на рис. 3.1.

****

Рисунок 3.1 - Активаційна функція **ReLu**

Гладке наближення ReLU є гладкою функцією яка називається **softplus**-функцією. Похідна softplus наведена нижче на рис. 3.2.

****

Рисунок 3.2 - Похідна softplus

Також у коді ваги ініціалізуються випадковими значеннями з нормального розподілу, але тут ваги множаться на *np.sqrt(2. / x)*, де x - кількість входів у шар, а 2 / x - це коефіцієнт, який допомагає зменшити дисперсію вагів. Цей метод ініціалізації відомий як Kaiming initialization, це допомагає підвищити стійкість нейронної мережі до надмірної дисперсії і швидкість навчання.

import random

import numpy as np  
from matplotlib import pyplot as plt  
  
  
def relu(z): # визначення RELU функції активації  
 return np.maximum(0, z)  
  
def relu\_prime(z): # Похідна гладкого наближення RELU  
 return 1.0/(1.0+np.exp(-z))  
  
  
class Network(object): # використовується для опису нейронної мережі  
  
 def \_\_init\_\_(self, sizes): # Конструктор класу  
 self.num\_layers = len(sizes) # Визначаємо кількість шарів в мережі  
 self.sizes = sizes # Зберігаємо розміри кожного шару  
 self.biases = [np.random.randn(y, 1) for y in  
 sizes[1:]] # Ініціалізація випадкових зміщень для кожного нейрона, окрім вхідного шару  
 self.weights = [np.random.randn(y, x) \* np.sqrt(2. / x) for x, y in zip(sizes[:-1], sizes[1:])]  
  
 def feedforward(self, a):  
 for b, w in zip(self.biases, self.weights):  
 a = relu(np.dot(w, a) + b)  
 return a  
  
 def SGD(self,training\_data,epochs,mini\_batch\_size,eta,test\_data):  
 test\_data = list(test\_data) # створюємо список об'єктів тестової вибірки  
 n\_test = len(test\_data) # обчислюємо довжину тестової вибірки  
 training\_data = list(training\_data) # створюємо список об'єктів навчальної вибірки  
 n = len(training\_data) # обчислюємо розмір навчальної вибірки  
 for j in range(epochs): # цикл по эпохам  
 random.shuffle(training\_data) # змішуємо елементи навчальної вибірки  
 mini\_batches = [training\_data[k:k + mini\_batch\_size] for k in range(0, n, mini\_batch\_size)] # створюємо підвибірки  
 for mini\_batch in mini\_batches: # цикл по подвибіркам  
 self.update\_mini\_batch(mini\_batch, eta) # один крок градієнтного спуску  
 print("Epoch {0}: {1} / {2}".format(j, self.evaluate(test\_data), n\_test)) # дивимось прогрес у навчанні  
  
 def update\_mini\_batch(self , mini\_batch , eta ):  
 nabla\_b = [np.zeros(b.shape) for b in self.biases] # список градієнтів dC/db для кожного шару (спочатку заповнюється нулями)  
 nabla\_w = [np.zeros(w.shape) for w in self.weights] # список градієнтів dC/dw для кожного шару (спочатку заповнюється нулями)  
 for x, y in mini\_batch:  
 delta\_nabla\_b, delta\_nabla\_w = self.backprop(x, y) # пошарово обчислюємо градієнти dC/db та dC/dw для поточного прикладу (x, y)  
 nabla\_b = [nb + dnb for nb, dnb in zip(nabla\_b, delta\_nabla\_b)] #підсумовуємо градієнти dC / db для різних прикладів поточної підвиборки  
 nabla\_w = [nw + dnw for nw, dnw in zip(nabla\_w, delta\_nabla\_w)] #підсумовуємо градієнти dC / dw для різних прикладів поточної підвиборки  
  
 self.weights = [w - (eta / len(mini\_batch)) \* nw for w, nw in zip(self.weights, nabla\_w)] #оновлюємо всі ваги w нейронної мережі  
 self.biases = [b - (eta / len(mini\_batch)) \* nb for b, nb in zip(self.biases, nabla\_b)] # оновлюємо всі зміщення b нейронної мережі  
  
  
  
 def backprop(self, x , y ):  
 nabla\_b = [np.zeros(b.shape) for b in  
 self.biases] # список градієнтів dC/db для кожного шару (спочатку заповнюється нулями)  
  
 nabla\_w = [np.zeros(w.shape) for w in  
 self.weights] # список градієнтів dC/dw для кожного шару (спочатку заповнюється нулями)  
  
 # визначення змінних  
 activation = x # вихідні сигнали шару (спочатку відповідають вихідним сигналам 1-го шару або вхідним сигналам мережі)  
 activations = [x] # список вихідних сигналів по всім шарам (спочатку містить тільки вихідні сигнали 1-го шару)  
 zs = [] # список активаційних потенціалів по всім шарам (спочатку пустий)  
  
 # пряме розповсюдження  
 for b, w in zip(self.biases, self.weights):  
 z = np.dot(w, activation) + b # зчитуємо активаційні потенціали поточного шару  
 zs.append(z) # додаємо елемент (активаційні потенціали шару) в кінець списку  
 activation = relu(z) # зчитуємо вихідні сигнали поточного шару, застосовуючи сигмоїдальну функцію активації до активаційних потенціалів шару  
 activations.append(activation) # додаємо елемент (вихідні сигнали шару) в кінець списку  
  
 # зворотне розповсюдження  
 delta = self.cost\_derivative(activations[-1], y) \* relu\_prime(zs[-1]) # зчитуємо міру впливу нейронів вихідного шару L на величину помилки (BP1)  
 nabla\_b[-1] = delta # градієнт dC/db для шару L (BP3)  
 nabla\_w[-1] = np.dot(delta, activations[-2].transpose()) # градієнт dC / dw для шару L(BP4)  
  
 for l in range(2, self.num\_layers):  
 z = zs[-l] # активаційні потенціали l-го шару (рухаємось по списку справа наліво)  
 sp = relu\_prime(z) # зчитуємо сигмоїдальну функцію від активаційних потенціалів l-го шару  
 delta = np.dot(self.weights[-l + 1].transpose(), delta) \* sp #зчитуємо міру впливу нейронів l - го шару на величину помилки(BP2)  
 nabla\_b[-l] = delta # градієнт dC/db для l-го шару (BP3)  
 nabla\_w[-l] = np.dot(delta, activations[-l-1].transpose()) #градієнт dC / dw для l - го шару(BP4)  
 return (nabla\_b, nabla\_w)  
  
 def evaluate(self, test\_data): # Оцінка прогресу в навчанні  
 test\_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test\_data]  
  
 return sum(int(x == y) for (x, y) in test\_results)  
  
 def cost\_derivative(self, output\_activations,y): # Обчислення приватних похідних функції вартості по вихідним сигналам останнього шару  
 return (output\_activations - y)  
  
  
def image\_selection(net):  
 index1 = random.randint(0, len(test\_data) - 1)  
 index2 = random.randint(0, len(test\_data) - 1)  
  
 # Отримаємо вхідні дані та мітки за вибраними індексами  
 image1, label1 = test\_data[index1]  
 image2, label2 = test\_data[index2]  
  
 # Класифікуємо вхідні дані з допомогою навченої мережі  
 prediction1 = net.feedforward(image1)  
 prediction2 = net.feedforward(image2)  
  
 # Візуалізуємо результати  
 visualize\_classification\_result(image1, label1, prediction1)  
 visualize\_classification\_result(image2, label2, prediction2)  
  
def visualize\_classification\_result(image, label, prediction):  
 # Розміщення зображення  
 plt.subplot(1, 2, 1)  
 plt.imshow(image.reshape(28, 28), cmap='gray')  
 plt.title(f"True Label: {label}")  
  
 # Виведемо результат класифікації  
 plt.subplot(1, 2, 2)  
  
 # Розділимо значення prediction на окремі класи  
 classes = list(range(10))  
 plt.bar(classes, prediction.flatten()) # Використовуємо .flatten() для перетворення на одновимірний масив  
 plt.xticks(range(10), [str(i) for i in range(10)]) # Встановлюємо значення на горизонтальній осі як цілі числа  
 plt.title("Network's Prediction")  
 plt.show()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 from Network2.fashion\_mnist\_loader import load\_data\_wrapper  
 training\_data, test\_data = load\_data\_wrapper()  
  
 net = Network([784, 30, 10])  
 net.SGD(training\_data, 10, 16, 0.065, test\_data=test\_data)  
 image\_selection(net)

**4 Навчання нейромережі**

Створено мережу, що складається з трьох шарів: вхідний шар мережі складається з 784 нейронів; внутрішній шар з 30 нейронів та вихідний шар з 10 нейронів.

Для навчання були обрані такі гіперпараметри:

Кількість епох навчання : 10,

Розмір міні пакета : 16,

Швидкість навчання: 0.065.

Результати навчання на рис. 4.1.

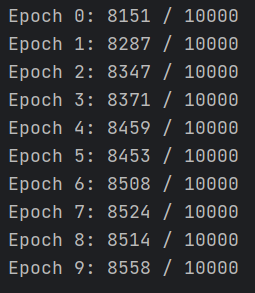


Рисунок 4.1 – Результати навчання

Отже точність класифікаціє дорівнює 85.6%, зі збільшенням кількості епох збільшується час очікування навчання та нейромережа не виходить за межі 85-86% точності.

Створена функція *image\_selection* для рандомного вибору прикладів з бібліотеки та подано їх на вхід мережі після чого отримуємо результат класифікації. Також зроблена процедура візуалізації цих двох обраних зображень у функції *visualize\_classification\_result.*

На рис. 4.2, 4.3 продемонстровано успішну роботу нейромережі на двох будь яких довільно вибраних прикладах зображень бібліотеки Fashion MNIST та виведено дійсний клас зображення для порівняння з передбачуваним результатом нейромережі.

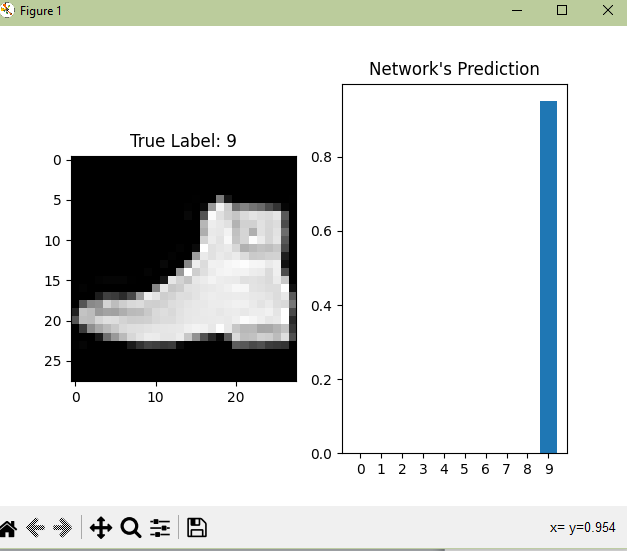


Рисунок 4.2 – Перший приклад

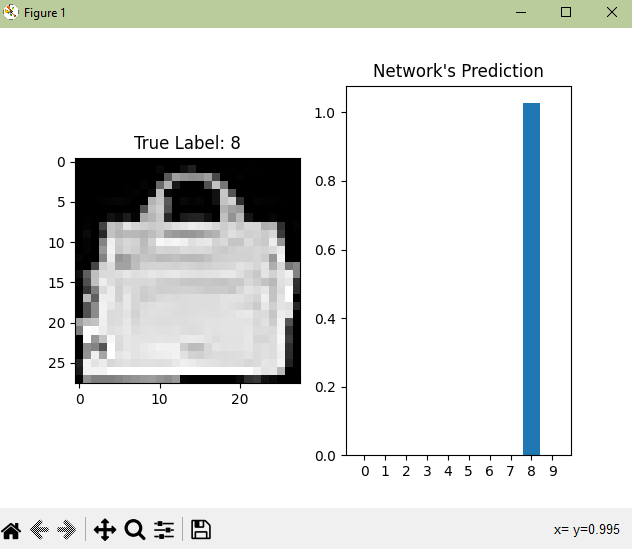


Рисунок 4.3 – Другий приклад

**5 Порівняння двох мереж на різних активаційних функцій**

Для початку визначимося, що ми маємо результати роботи Network1 для класифікації зображень MNIST з сигмоїдальною функцією див. рис. 1.1. – 94%.

Та результати роботи Network2 для зображень Fashion MNIST на рис. 4.1. – 85%.

Перевіримо Network1 на бібліотеці Fashion MNIST, для цього змінимо вхідні данні на рис. 5.1.

from Network2.fashion\_mnist\_loader import load\_data\_wrapper  
training\_data, test\_data = load\_data\_wrapper()

Рисунок 5.1 – Завантажуємо дані з Fashion MNIST

Результати роботи нейромережі наведені на рис. 5.2.

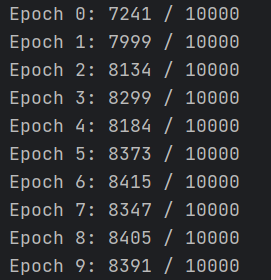


Рисунок 5.2 – Network1 та Fashion MNIST

Отже точність класифікаціє дорівнює 84%, що менше приблизно на 1.5% ніж класифікує Network2.

Далі протестуємо Network2 з бібліотекою MNIST. Для цього також змінимо вхідні данні на рис. 5.3.

from Network1.mnist\_loader import load\_data\_wrapper  
training\_data, validation\_data, test\_data = load\_data\_wrapper()

Рисунок 5.3 - Завантажуємо дані з MNIST

Результати роботи нейромережі наведені на рис. 5.4.

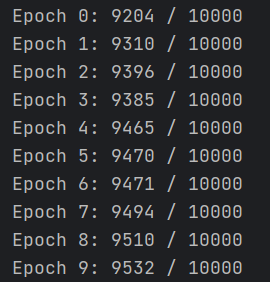


Рисунок 5.4 – Network2 та MNIST

Тут ми бачимо, що активаційна функція ReLu краще класифікує зображення з MNIST ніж сигмоїдна. У цьому випадку Network2 збільшила точність приблизно на 1.5% та класифікувала на 142 зображення більше ніж Network1.

**6. Висновки**

У міні-пакетному SGD під час навчання моделі я беру групу прикладів - 16. Такий підхід працює краще, тому що потрібний єдиний цикл для міні-пакетів, а не для кожного прикладу.

Швидкість навчання - це параметр налаштування в алгоритмі оптимізації, що визначає розмір кроку на кожній ітерації під час руху до мінімуму функції втрат. Я обрав маленьку швидкість саме тому, що:

1 Іноді ReLU може призводити до більш агресивних змін вагів, оскільки вона допускає прохід значень через себе без обмежень. Тому може бути корисною менша швидкість навчання для стабільності навчання.

2 Деякі архітектури з ReLU можуть стикатися з проблемами, такими як "зниклі градієнти" або "вибухаючі градієнти," що може призвести до труднощів у навчанні.

Були порівняні дві нейронні мережі, Network1 і Network2, які використовують різні активаційні функції (сигмоїдальну та ReLU) для класифікації зображень з наборів даних MNIST і Fashion MNIST.

Для MNIST (зображення рукописних цифр) Network2 з ReLU показала кращі результати, ніж Network1 з сигмоїдальною активацією. Network2 досягла більшої точності класифікації (приблизно на 1.5% вище).

Для Fashion MNIST, Network2 показала кращі результати, ніж Network1. Network2 досягла більшої точності класифікації (приблизно на 1.5% вище).

Важливо зазначити, що вибір активаційної функції залежить від конкретної задачі та даних. В даному випадку, ReLU виявилася кращою для MNIST та Fashion MNIST.

Отже, висновок полягає в тому, що вибір активаційної функції має велике значення для досягнення оптимальної точності у задачі класифікації. Різні функції можуть вести себе по-різному на різних наборах даних.