ПРАКТИЧНА РОБОТА №8. СТВОРЕННЯ ПРОСТОЇ НЕЙРОННОЇ МЕРЕЖІ ДЛЯ КЛАСИФІКАЦІЇ ЗОБРАЖЕНЬ

Мета роботи: Ознайомитися з основами нейронних мереж та їх застосуванням для класифікації зображень. Навчитися створювати просту нейронну мережу з використанням модуля Numpy. Зрозуміти принципи роботи нейронних мереж та їх ключові компоненти.

Основні теоретичні відомості

Нейронна мережа – це математична модель, натхненна будовою людського мозку. Вона складається з з'єднаних між собою нейронів, які обробляють та передають інформацію. Нейронні мережі можуть навчатися на даних, знаходити закономірності та робити прогнози.

Типи нейронних мереж варіюються від тих, що мають лише один або два рівні односпрямованої логіки, до складних з кількома входами, багатьма спрямованими петлями та шарами зворотного зв’язку. Загалом ці системи використовують алгоритми у своєму програмуванні для визначення контролю та організації своєї функції. Більшість систем використовують «ваги» для зміни параметрів пропускної здатності та різноманітних з’єднань із нейронами. Нейронні мережі можуть бути автономними та навчатися за допомогою зовнішнього «вчителя» або навіть самонавчання. Навчання нейронної мережі досягається шляхом багаторазового оновлення вагових значень під час зворотного розповсюдження, доки не буде знайдено поєднання значень, яке найкраще відображає вхідні дані на виході.

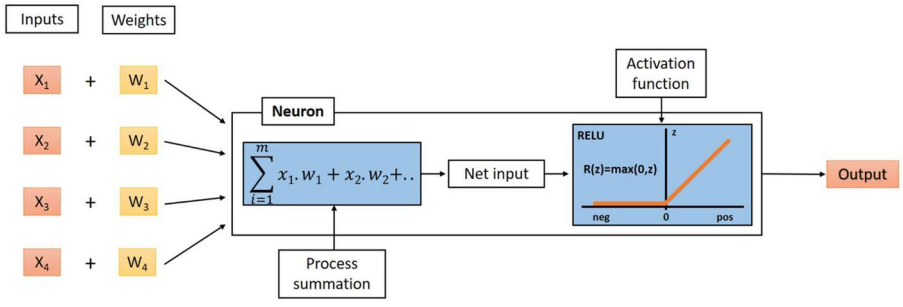

Рисунок 8.1 – Графічна схема нейронної мережі

Вхідний шар (input layer) – шар, на який передаються вхідні дані. Це перший рівень нашої нейронної мережі.

Вихідний шар (output layer) – шар, з якого ми будемо отримувати результати. Щойно дані пройдуть через усі інші рівні, вони надійдуть сюди.

Прихований шар (hidden layer) – усі інші шари нашої нейронної мережі називаються «прихованими шарами». Це тому, що вони приховані для нас, ми не можемо їх спостерігати. Більшість нейронних мереж складається принаймні з одного прихованого шару, але може мати необмежену кількість. Як правило, чим складніша модель, тим більше прихованих шарів.

Нейрони (neurons, units) – кожен шар складається з так званих нейронів, якщо коротко, то кожен нейрон містить в собі одне числове значення. Це означає, що у випадку нашого вхідного рівня він матиме стільки нейронів, скільки у нас є вхідної інформації. Наприклад, скажімо, ми хочемо передати зображення розміром 28x28 пікселів, тобто 784 пікселя. Нам знадобиться 784 нейрони на нашому вхідному рівні, щоб захопити кожен із цих пікселів.

Вагові значення (weights) – "Вага" пов’язана з кожним з’єднанням у нашій нейронній мережі. Кожна пара з’єднаних вузлів матиме одну вагу, яка позначає силу зв’язку між ними. Вони є життєво важливими для внутрішньої роботи нейронної мережі та будуть налаштовані під час навчання нейронної мережі. Модель намагатиметься визначити, якими мають бути ці ваги, щоб досягти найкращого результату. Ваги починаються з постійного або випадкового значення та змінюватимуться, коли мережа переглядатиме дані навчання.

Зсув (bias) – Зсув є ще однією важливою частиною нейронних мереж, і вони також будуть налаштовані під час навчання моделі. Зсув – це просто постійне значення, пов’язане з кожним шаром. Його можна розглядати як додатковий нейрон, який не має зв’язків. Метою зсуву є зсув всієї функції активації на постійне значення. Це забезпечує набагато більшу гнучкість під час вибору активації та навчання мережі. Для кожного шару є один зсув.

Функції активації (activation functions) — це просто функція, яка застосовується до зваженої суми нейрона. Вони можуть бути будь-якими, але зазвичай це функції вищого порядку/ступеня, які мають на меті додати вищий

вимір до наших даних. Ми хотіли б зробити це, щоб додати більше спільності до нашої моделі. Трансформуючи наші дані у вищий вимір, ми зазвичай можемо робити кращі та складніші прогнози.

Найпоширеніші функції активації:

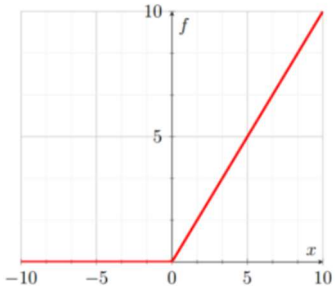

Рисунок 8.2 – Функція активації ReLU (rectified linear unit)

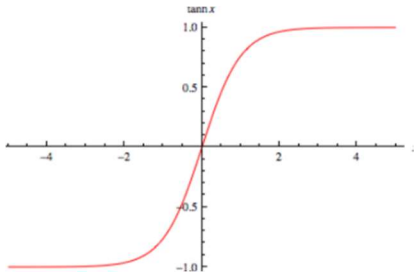

Рисунок 8.3 – Функція активації tanh (гіперболічний тангенс)

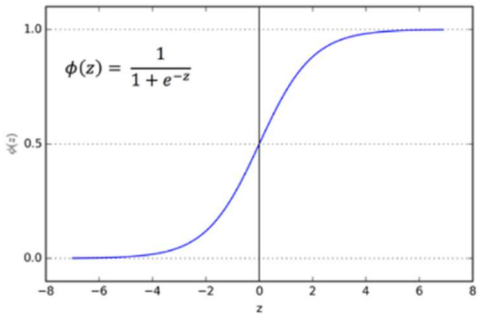

Рисунок 8.4 – Функція активації softmax

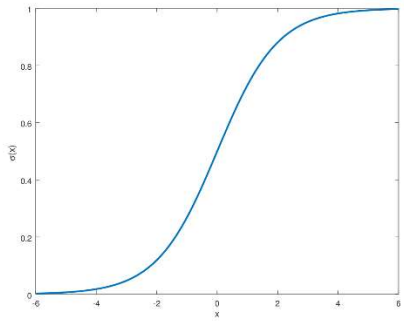

Рисунок 8.5 – Функція активації sigmoid

Зворотне поширення (backward propagation) – це фундаментальний алгоритм навчання нейронних мереж. Це те, що змінює ваги та упередження нейронної мережі. Щоб повністю пояснити цей процес, нам потрібно почати з обговорення так званої функції витрат/збитків.

Функція витрат/збитків (loss function) – на етапі навчання мережа, ймовірно, буде робити багато помилок і поганих прогнозів. Фактично, на початку навчання мережа нічого не знає (вона має випадкові ваги та зміщення)!

Градієнтний спуск (gradient descent) – градієнтний спуск та зворотне поширення тісно пов'язані. Градієнтний спуск — це алгоритм, який використовується для пошуку оптимальних параметрів (ваги та зміщення) для мережі, тоді як зворотне поширення — це процес обчислення градієнта, який використовується на етапі градієнтного спуску.

«Градієнтний спуск — це алгоритм оптимізації, який використовується для мінімізації певної функції шляхом ітеративного переміщення в напрямку найкрутішого спуску, визначеного мінусом градієнта. У машинному навчанні ми використовуємо градієнтний спуск для оновлення параметрів нашої моделі».

Потрібен певний спосіб оцінити наскільки добре вона працює. Для навчальних даних у нас є функції (вхід) і мітки (очікуваний вихід), через це ми можемо порівняти вихід нашої мережі з очікуваним виходом. На основі різниці між цими значеннями ми можемо визначити, добре чи погано виконала свою роботу наша мережа. Якщо мережа виконала хорошу роботу, ми внесемо незначні зміни у ваги та зміщення. Якщо вона виконала свою роботу погано, наші зміни можуть бути більш радикальними.

Отже, тут з’являється функція витрати/втрати. Ця функція відповідає за визначення того, наскільки добре працює мережа. Ми передаємо їй результат і очікуваний вихід, і він повертає нам деяке значення, що представляє вартість/втрати мережі. Це фактично змушує мережі оптимізувати цю функцію витрат, намагаючись зробити її якомога нижчою.

Деякі загальні функції втрат/витрат включають:

1) Середньоквадратична похибка;

2) Середня квадратична похибка;

3) Середня абсолютна похибка;

4) Перехресна ентропія.

Класифікація зображень – це завдання, яке полягає у визначенні категорії, до якої належить зображення. Нейронні мережі можуть успішно використовуватися для класифікації зображень, досягаючи високої точності.

Для наочного прикладу реалізуємо просту 3-шарову нейронну мережу з нуля за допомогою Numpy і навчимо її на наборі даних MNIST для розпізнавання рукописних цифр. Набір даних MNIST – це колекція зображень 28x28 у відтінках сірого, що містять рукописні цифри від 0 до 9. Його зазвичай використовують для навчання моделей класифікації зображень. Вхідний шар матиме 784 одиниці, що відповідають 784 пікселям у кожному вхідному зображенні 28x28. Прихований шар матиме 10 одиниць з активацією ReLU, і, нарешті, наш вихідний шар матиме 10 одиниць, що відповідають десяти розрядним класам з активацією softmax. Це навчальний приклад, за допомогою якого ви зможете краще зрозуміти математику, що лежить в основі нейронних мереж:

Імпортування бібліотек: numpy, pandas, random: Для числових операцій, маніпуляцій з даними та генерації випадкових чисел. matplotlib.pyplot: Для візуалізації даних. tensorflow: Для завантаження набору даних MNIST.

```
import numpy as np
import pandas as pd
import random
from matplotlib import pyplot as plt
import tensorflow as tf
```

Завантаження та попередня обробка даних: Набір даних MNIST завантажується за допомогою вбудованої функції TensorFlow. Навчальні та тестові зображення перетворюються на сплющений масив форми (784, 1) для кожного зразка. Цей процес згладжування необхідний для того, щоб подати дані в нейронну мережу. Значення пікселів нормалізуються до діапазону [0, 1].

```python
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

X_train_flatten = X_train.reshape(X_train.shape[0],-1).T
X_test_flatten = X_test.reshape(X_test.shape[0],-1).T

m=X_train_flatten.shape[1]

def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

X_train, X_test = prep_pixels(X_train_flatten, X_test_flatten)
```

Клас нейронних мереж: Клас NeuralNetwork визначає архітектуру нейронної мережі та методи навчання. Він містить методи ініціалізації параметрів, прямого і зворотного поширення, оновлення параметрів і прогнозування.

```python
class NeuralNetwork:
    def __init__(self):
        pass
```

Ваги (W1, W2, W3) та **зсуви** (b1, b2, b3) нейронної мережі ініціалізуються випадковим чином за допомогою np.random.rand(). Ваги ініціалізуються невеликими випадковими значеннями з центром навколо нуля, щоб порушити симетрію і запобігти застряганню мережі під час навчання.

```python
def init_params(self):
    W1 = np.random.rand(64, 784) - 0.5  # Ваги для першого шару (64 нейрони)
    b1 = np.random.rand(64, 1) - 0.5    # Зсуви для першого шару
    W2 = np.random.rand(32, 64) - 0.5   # Ваги для другого шару (32 нейрони)
    b2 = np.random.rand(32, 1) - 0.5    # Зсуви для другого шару
    W3 = np.random.rand(10, 32) - 0.5   # Ваги для третього шару (10 нейронів)
    b3 = np.random.rand(10, 1) - 0.5    # Зсуви для третього шару
    return W1, b1, W2, b2, W3, b3
```

Функції активації: Функція активації ReLU (Rectified Linear Unit) використовується для нейронів прихованого шару. ReLU вносить нелінійність в модель, дозволяючи їй вивчати складні закономірності в даних. Функція активації Softmax використовується для нейронів вихідного шару. Вона

перетворює необроблені оцінки (логіти) в ймовірності, що робить її придатною для багатокласових задач класифікації, таких як розпізнавання цифр MNIST.

```
def ReLU(self, Z):
    return np.maximum(Z, 0)
def softmax(self, Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A
```

Пряме поширення (forward propagation): Під час прямого поширення вхідні дані проходять через шари мережі для генерації прогнозів. Вихід кожного шару обчислюється за допомогою матричного множення вхідних даних на ваги з подальшим додаванням зсувів і застосуванням функцій активації.

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$
$$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$$
$$A^{[3]} = \sigma(Z^{[3]})$$

```
def forward_prop(self, W1, b1, W2, b2, W3, b3, X):
    Z1 = W1.dot(X) + b1
    A1 = self.ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = self.ReLU(Z2)
    Z3 = W3.dot(A2) + b3
    A3 = self.softmax(Z3)
    return Z1, A1, Z2, A2, Z3, A3
```

Зворотне поширення: Зворотне поширення використовується для обчислення градієнтів функції втрат по відношенню до параметрів. Ці градієнти обчислюються за допомогою ланцюгового правила, починаючи з вихідного шару і поширюючись у зворотному напрямку через мережу.

$$dZ^{[3]} = A^{[3]} - Y$$
$$dW^{[3]} = \frac{1}{m} dZ^{[3]} A^{[2]T}$$
$$db^{[3]} = \frac{1}{m} \sum dZ^{[3]}$$
$$dZ^{[2]} = W^{[3]T} dZ^{[3]} * \sigma^{[2]\prime}(Z^{[2]})$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \sum dZ^{[2]}$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \sigma^{[1]\prime}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$
$$db^{[1]} = \frac{1}{m} \sum dZ^{[1]}$$

```
def backward_prop(self, Z1, A1, Z2, A2, Z3, A3, W1, W2, W3, X, Y):
    one_hot_Y = self.one_hot(Y)
    dZ3 = A3 - one_hot_Y
    dW3 = 1 / m * dZ3.dot(A2.T)
    db3 = 1 / m * np.sum(dZ3)
    dZ2 = W3.T.dot(dZ3) * self.ReLU_deriv(Z2)
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * self.ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2, dW3, db3
```

```
def ReLU_deriv(self, Z):
    return Z > 0
```

Оновити параметри: Після обчислення градієнтів параметри (ваги та зміщення) оновлюються за допомогою градієнтного спуску. Швидкість навчання (альфа) контролює розмір кроків під час оптимізації.

$$W^{[3]} := W^{[3]} - \alpha dW^{[3]}$$
$$b^{[3]} := b^{[3]} - \alpha db^{[3]}$$
$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$
$$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$
$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$
$$b^{[2]} := b^{[1]} - \alpha db^{[1]}$$

```
def update_params(self, W1, b1, W2, b2, W3, b3, dW1, db1, dW2, db2, dW3, db3, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    W3 = W3 - alpha * dW3
    b3 = b3 - alpha * db3
    return W1, b1, W2, b2, W3, b3
```

Утиліти. one_hot: Перетворює цілочисельні мітки у вектори з однократним кодуванням (one-hot encoding). "One-hot encoding" - це спосіб кодування категоріальних змінних, коли кожна категорія представлена вектором, де всі значення рівні нулю, окрім одного, яке відповідає індексу

категорії, і воно рівне одиниці. Таке кодування необхідне для навчання нейронної мережі. get_predictions та get_accuracy: Ці функції використовуються для оцінки продуктивності моделі шляхом обчислення точності прогнозів. make_predictions і test_prediction: Ці функції генерують передбачення на тестовому наборі та візуалізують деякі вибіркові передбачення відповідно.

```
def get_predictions(self, A3):
    return np.argmax(A3, 0)

def get_accuracy(self, predictions, Y):
    return np.sum(predictions == Y) / Y.size
```

```
def one_hot(self, Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y
```

```
def make_predictions(self, X, W1, b1, W2, b2, W3, b3):
    _, _, _, _, _, A3 = self.forward_prop(W1, b1, W2, b2, W3, b3, X)
    predictions = self.get_predictions(A3)
    return predictions

def test_prediction(self, X, Y, W1, b1, W2, b2, W3, b3):
    plt.figure(figsize=(10, 10))
    for i in range(4):
        ax = plt.subplot(2, 2, i + 1)
        index = random.randint(0, X.shape[1]-1)
        current_image = X_train[:, index, None]
        prediction = self.make_predictions(X_train[:, index, None], W1, b1, W2, b2, W3, b3)
        label = Y[index]
        if label == prediction[0]:
          color = "green"
        else:
          color = "red"
        current_image = current_image.reshape((28, 28)) * 255
        plt.gray()
        plt.title("Prediction: {} (True: {})".format(prediction[0],label), color=color)
        plt.imshow(current_image, interpolation='nearest')
```

Навчання та тестування: Метод gradient_descent навчає нейронну мережу за допомогою градієнтного спуску. Він ітеративно оновлює параметри для мінімізації функції втрат. Метод test_prediction візуалізує вибіркові прогнози на тестовому наборі, показуючи передбачену цифру поряд з істинною цифрою для порівняння.

```
def gradient_descent(self, X, Y, alpha, iterations):
    W1, b1, W2, b2, W3, b3 = self.init_params()
    for i in range(iterations):
        Z1, A1, Z2, A2, Z3, A3 = self.forward_prop(W1, b1, W2, b2, W3, b3, X)
        dW1, db1, dW2, db2, dW3, db3 = self.backward_prop(Z1, A1, Z2, A2, Z3, A3, W1, W2, W3, X, Y)
        W1, b1, W2, b2, W3, b3 = self.update_params(W1, b1, W2, b2, W3, b3, dW1, db1, dW2, db2, dW3, db3, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = self.get_predictions(A3)
            print("Accuracy: ", self.get_accuracy(predictions, Y))
    return W1, b1, W2, b2, W3, b3
```

```
model=NeuralNetwork()
W1, b1, W2, b2, W3, b3 = model.gradient_descent(X_train, y_train, 0.10, 200)
model.test_prediction(X_test, y_test, W1, b1, W2, b2)
```

```
Accuracy:  0.8939833333333334
Iteration:  450
Accuracy:  0.8947833333333334
Iteration:  460
Accuracy:  0.89585
Iteration:  470
Accuracy:  0.89695
Iteration:  480
Accuracy:  0.89775
Iteration:  490
Accuracy:  0.8985666666666666
```

Вивід результату за допомогою функції test_prediction наведено нижче, зверніть увагу, що нейронна мережа показує неоднозначний результат, це означає що на тестовому наборі даних вона працює гірше на відміну від тренувального набору даних, отже можливо корегувати кількість ітерацій, крок альфа, кількість прихованих шарів та нейронів для покращення роботи цієї нейронної мережі.

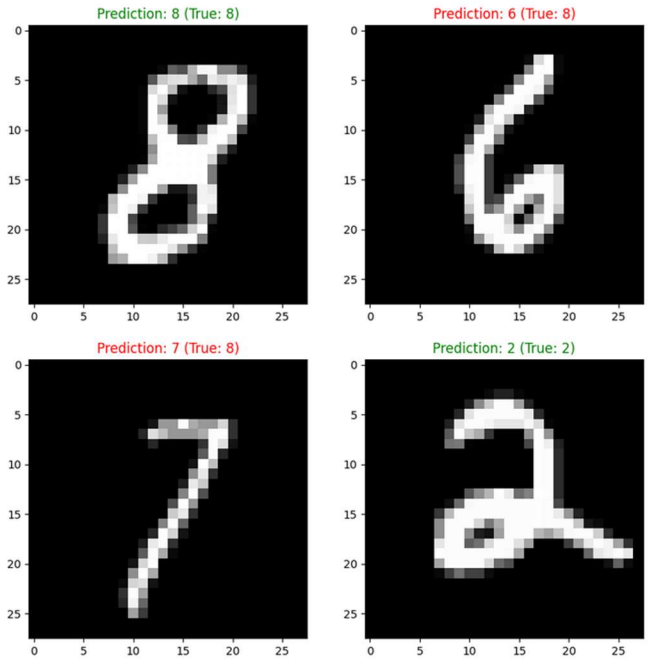

Рисунок 8.6 – Результат виконання функції test_prediction

У наданому коді використовується метод зворотнього поширення помилки (backpropagation) для навчання нейронної мережі. Метод зворотнього поширення є основним алгоритмом для навчання нейронних мереж і використовується для обчислення градієнтів функції втрати за вагами мережі. Він базується на принципі ланцюгового правила диференціювання і дозволяє ефективно розповсюджувати помилку від виходу мережі до входу, визначаючи, як параметри мережі повинні бути змінені для зменшення помилки прогнозування.

Функція помилок в даному випадку не явно визначена в коді, але вона присутня у вигляді методу backward_prop. У цьому методі обчислюються градієнти відносно параметрів мережі, щоб вони могли бути оновлені під час оптимізації. В якості функції втрати зазвичай використовується категоріальна перехресна ентропія (Categorical Cross-Entropy), але в коді вона не вказана явно. Однак, метод backward_prop обчислює градієнти для оновлення параметрів на основі помилок між передбаченими значеннями та справжніми мітками.

Завдання до виконання

Використовуючи засоби Numpy виконати індивідуальне завдання відповідно до таблиці варіантів (таб. 8.1). Для виконання завдання дозволено використовувати **допоміжні засоби**, але на захисті роботи потрібно детально пояснити як працює реалізована нейронна мережа. Якщо при завантаженні набору даних він одразу не поділений на тренувальний та тестові набори, то поділити з відношенням 80% на 20%. Точність нейронної мережі повинна бути не менше 80% на тестовому наборі даних. Протестуйте нейронну мережу з різними гіперпараметрами, та наведіть графіки, в яких показано як змінюється точність класифікації від заданих параметрів.

Додаткове завдання №1 (опціонально): імплементуйте нейронну мережу за допомогою бібліотек tensorflow або pytorch. Порівняйте реалізацію нейронних мереж між собою.

Додаткове завдання №2 (опціонально): реалізуйте «confusion matrix» для розробленої нейронної мережі. Поясніть її результати.

Таблиця 8.1 – Варіанти завдань

№	Завдання
1	**Класифікація тварин та транспортних засобів:** Використовуйте набір даних CIFAR-10 для класифікації зображень транспортних засобів та тварин. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh
2	**Визначення різних типів одягу:** Використовуйте набір даних Fashion-MNIST для класифікації різних типів одягу, таких як футболки, штани, плаття тощо. Нейронна мережа повинна складатись з 3-х шарів. Функції активації для прихованих шарів: relu, tanh
3	**Розпізнавання видів птахів:** Використовуйте набір даних CUB-200 для класифікації різних видів птахів на зображеннях. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh
4	**Класифікація категорій кухонних предметів:** Використовуйте набір даних Food-101 для класифікації кухонних предметів на зображеннях, таких як фрукти, овочі, страви тощо. Нейронна мережа повинна складатись з 5-х шарів. Функції активації для прихованих шарів: relu, tanh
5	**Класифікація різних видів листя:** Використовуйте набір даних plant_village для класифікації різних видів листя на зображеннях. Нейронна мережа повинна складатись з 3-х шарів. Функції активації для прихованих шарів: relu, tanh
6	**Класифікація різних сортів кави:** Використовуйте набір даних Coffee Dataset для класифікації різних сортів кави за зображеннями їх зерен. Нейронна мережа повинна складатись з 5-х шарів. Функції активації для прихованих шарів: relu, tanh
7	**Визначення типів квітів:** Використовуйте набір даних Tf_flowers для класифікації різних типів квітів на зображеннях. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh
8	**Класифікація різних видів хлібобулочних виробів:** Використовуйте набір даних Bakery Products для класифікації різних видів хлібобулочних виробів на зображеннях. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh
9	**Визначення порід собак:** Використовуйте набір даних stanford_dogs для класифікації різних порід собак на зображеннях. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh

10	**Розпізнавання різних видів тварин у дикій природі:** Використовуйте набір даних Caltech-101 для класифікації різних видів тварин у дикій природі на зображеннях. Нейронна мережа повинна складатись з 3-х шарів. Функції активації для прихованих шарів: relu, tanh
11	**Класифікація різних сортів винограду:** Використовуйте набір даних Grape Vine Dataset для класифікації різних сортів винограду за зображеннями їх грон. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh
12	**Визначення різних типів спортивного спорядження:** Використовуйте набір даних Sports Equipment для класифікації різних типів спортивного спорядження на зображеннях. Нейронна мережа повинна складатись з 3-х шарів. Функції активації для прихованих шарів: relu, tanh
13	**Класифікація різних видів риб:** Використовуйте набір даних Fish Recognition для класифікації різних видів риб на зображеннях. Нейронна мережа повинна складатись з 4-х шарів. Функції активації для прихованих шарів: relu, tanh
14	**Визначення різних типів фарб для волосся:** Використовуйте набір даних Hair Dye Dataset для класифікації різних типів фарб для волосся на зображеннях. Нейронна мережа повинна складатись з 6-х шарів. Функції активації для прихованих шарів: relu, tanh
15	**Класифікація різних видів хвороб рослин:** Використовуйте набір даних Plant Disease Dataset для класифікації різних видів хвороб рослин за зображеннями їх листя. Нейронна мережа повинна складатись з 5-х шарів. Функції активації для прихованих шарів: relu, tanh

Контрольні запитання

1) Як ви можете схарактеризувати поняття «нейронна мережа»?
2) Як влаштована нейронна мережа?
3) Як навчаються нейронні мережі?
4) Що таке зворотне поширення помилки?
5) У яких сферах використовуються нейронні мережі?
6) Які фактори впливають на час навчання нейронної мережі?
7) Які параметри нейронної мережі можна налаштувати для покращення її точності?
8) Що таке функція активації і які її типи?
9) Що таке градієнтний спуск?