НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики
Кафедра прикладної математики

Звіт
із лабораторної роботи
із дисципліни «СИСТЕМИ ГЛИБИННОГО НАВЧАННЯ»
на тему:

Розробка програмного забезпечення для реалізації двошарового персептрону з сигмоїдальною функцією активації

Виконав:
студент групи КМ-13
Онищенко В.С.

Перевірив:
Терейковський І. А.

Київ — 2024

ЗМІСТ

1 Теорія

1.1 Персептрон та багатошарові нейронні мережі

Персептрон є базовою моделлю штучного нейрона, яка була розроблена для розв'язання задач класифікації. Він приймає на вхід один або кілька сигналів (вхідних значень), зважує їх відповідно до навчальних параметрів (ваг), та обчислює значення виходу за допомогою функції активації. У разі використання багатошарового підходу, персептрон містить один або більше прихованих шарів нейронів, які допомагають мережі вивчати складні нелінійні залежності.

У даній роботі розглядаються наступні структури нейронних мереж:

- Класичний нейрон із одним входом та одним виходом;
- Двошаровий персептрон із одним прихованим нейроном та одним вихідним (структура 1-1-1);
- Двошаровий персептрон із структурою 2-3-1, де є два вхідних нейрони, три нейрони в прихованому шарі та один вихідний нейрон.

1.2 Сигмоїдна функція активації

Сигмоїдна функція активації є однією з найбільш популярних нелінійних функцій для активації нейронів у штучних нейронних мережах. Вона визначається як:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Ця функція переводить значення вхідного сигналу у діапазон (0, 1), що робить її зручною для роботи із значеннями ймовірностей або бінарними класифікаціями. Похідна сигмоїдної функції використовується під час навчання і обчислюється за

формулою:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)).$$

Похідна сигмоїдної функції необхідна для корекції ваг у процесі навчання за допомогою методу градієнтного спуску.

1.3 Градієнтний спуск

Градієнтний спуск — це ітеративний оптимізаційний метод, що використовується для мінімізації функції похибки (втрат) у нейронній мережі. У контексті нейронних мереж функція похибки визначає, наскільки добре модель прогнозує виходи для заданих вхідних даних. Основна ідея градієнтного спуску полягає в тому, щоб зменшувати похибку, коректуючи значення ваг у напрямку негативного градієнта функції похибки.

Алгоритм градієнтного спуску включає наступні кроки:

1. Ініціалізація ваг нейронів випадковими значеннями.

2. Визначення значення функції похибки на основі різниці між передбаченими та фактичними виходами.

3. Розрахунок градієнта функції похибки щодо кожної ваги. Для цього використовується правило ланцюга, яке пов'язує похибку із значенням ваг через сигмоїдну функцію активації.

4. Корекція ваг за формулою:

$$w = w - \eta \cdot \frac{\partial E}{\partial w},$$

де w — вага, η — швидкість навчання, а $\frac{\partial E}{\partial w}$ — частинна похідна функції похибки по вазі w.

Процес навчання триває до досягнення заданої кількості ітерацій (епох) або до зменшення похибки нижче визначеного порогу.

1.4 Навчання нейронних мереж за допомогою градієнтного спуску

Під час навчання персептронів, таких як описані в цій лабораторній роботі, ваги та зсуви (bias) коригуються відповідно до градієнта похибки. Ваги прихованого шару оновлюються для мінімізації загальної похибки вихідного нейрону, а похибки обчислюються як похідні функцій активації. У двошаровій мережі процес коригування ваг проходить через прихований шар до вхідного, використовуючи зворотне поширення (backpropagation) градієнта для коригування кожного шару.

Градієнтний спуск у поєднанні з сигмоїдною активацією дозволяє нейронним мережам вивчати та адаптуватися до різних патернів у даних, що є ключовим принципом машинного навчання.

2 Програма

Ініціалізація та функція активації

Для цієї програми використовуються основні бібліотеки Python, зокрема **numpy** для роботи з векторами та матрицями.

Спершу визначаємо сигмоїдну функцію активації та її похідну, оскільки вона буде використовуватись у всіх нейронах:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

Сигмоїдна функція обмежує значення виходу в діапазоні $(0, 1)$, що дозволяє моделювати ймовірності. Похідна функції потрібна для корекції ваг під час навчання з використанням градієнтного спуску.

Класичний нейрон

Класичний нейрон є базовим компонентом нейронної мережі. Він містить методи `train` для навчання та `predict` для передбачення вихідного значення. Вага і зсув ініціалізуються випадковими значеннями.

```
class Neuron:
    def __init__(self):
        self.weight = np.random.rand()
        self.bias = np.random.rand()
```

Тут ваги і зсуви (bias) нейрону ініціалізуються випадковими числами, що допомагає уникнути однакових результатів під час обчислень.

Метод **train** реалізує процес навчання за допомогою градієнтного спуску:

```
    def train(self, x, y, learning_rate=0.1, epochs=1000):
        for _ in range(epochs):
            pred = self.predict(x)
            error = y - pred
            self.weight += learning_rate * error * x *
                           sigmoid_derivative(pred)
            self.bias += learning_rate * error *
                         sigmoid_derivative(pred)
```

Функція **train** розраховує похибку як різницю між очікуваним значенням y та передбаченим значенням **pred**. Потім вага і зсув коригуються, використовуючи похідну сигмоїдної функції, щоб мінімізувати похибку.

Метод **predict** розраховує вихід нейрону на основі вхідного значення та поточних ваг:

```
    def predict(self, x):
        return sigmoid(x * self.weight + self.bias)
```

Елементарний двошаровий персептрон (1-1-1)

Ця частина реалізує персептрон із одним нейроном у прихованому шарі та одним на виході. Клас **SimplePerceptron** містить один нейрон для прихованого шару і один нейрон для вихідного шару.

```
class SimplePerceptron:
    def __init__(self):
        self.hidden_neuron = Neuron()
        self.output_neuron = Neuron()
```

Конструктор класу створює два екземпляри **Neuron**, один для прихованого шару (**hidden_neuron**) і один для вихідного шару (**output_neuron**).

Метод **train** відповідає за навчання на одному прикладі. Навчання спершу проходить прихований шар, а потім вихідний шар:

```
    def train(self, x, y, learning_rate=0.1, epochs=1000):
        for _ in range(epochs):
            hidden_output = self.hidden_neuron.predict(x)
            pred = self.output_neuron.predict(hidden_output)
            error = y - pred
            self.output_neuron.weight += learning_rate * error *
                        hidden_output * sigmoid_derivative(pred)
            self.output_neuron.bias += learning_rate * error *
                                       sigmoid_derivative(pred)
```

Спершу обчислюється вихід для прихованого нейрону, який передається на вхід вихідному нейрону. Вихідний нейрон коригує свої ваги за допомогою градієнтного спуску, після чого виконується оновлення ваг прихованого нейрону на основі отриманого результату.

Метод **predict** обчислює вихід двошарового персептрону, послідовно обробляючи вхід через прихований і вихідний шари:

```
    def predict(self, x):
        hidden_output = self.hidden_neuron.predict(x)
        return self.output_neuron.predict(hidden_output)
```

Двошаровий персептрон із структурою 2-3-1

У цій частині реалізується двошаровий персептрон із двома вхідними нейронами, трьома нейронами у прихованому шарі та одним на виході. Така структура дозволяє працювати із задачами, що вимагають складніших патернів.

```
class TwoLayerPerceptron:
    def __init__(self):
        self.hidden_layer = [Neuron() for _ in range(3)]
```

```
        self.output_neuron = Neuron()
```

Конструктор створює три нейрони для прихованого шару та один нейрон для вихідного шару.

Метод **train** реалізує режим навчання «ON-LINE», що полягає у корекції ваг для кожного прикладу з тренувального набору даних. У прихованому шарі ваги оновлюються відповідно до похибки вихідного нейрону.

```
    def train(self, x, y, learning_rate=0.1):
        for epoch in range(1000):
            for i in range(len(x)):
                hidden_outputs = np.array([
                    self.hidden_layer[j].predict(x[i][j % len(x[i])])
                    for j in range(3)])

                pred = self.output_neuron.predict(np.sum(hidden_outputs))
                error = y[i] - pred
                self.output_neuron.weight += learning_rate * error *
                    np.sum(hidden_outputs) * sigmoid_derivative(pred)
                self.output_neuron.bias += learning_rate * error *
                                           sigmoid_derivative(pred)
```

У кожній ітерації для кожного прикладу обчислюється вихід прихованого шару та похибка вихідного нейрону. Вага і зсув вихідного нейрону оновлюються на основі похибки, після чого ваги прихованого шару коригуються.

```
                for j, neuron in enumerate(self.hidden_layer):
                    hidden_error = error * self.output_neuron.weight *
                                   sigmoid_derivative(hidden_outputs[j])
                    neuron.weight += learning_rate * hidden_error *
                                   x[i][j % len(x[i])] *
                                   sigmoid_derivative(hidden_outputs[j])
                    neuron.bias += learning_rate * hidden_error *
```

```
                                    sigmoid_derivative(hidden_outputs[j])
```

Кожен нейрон прихованого шару оновлює свою вагу і зсув на основі похибки, розрахованої для вихідного нейрону. Це дозволяє зменшити загальну похибку мережі.

Метод **predict** обчислює вихідну відповідь персептрону, передаючи вхід через прихований і вихідний шари:

```
    def predict(self, x):
        hidden_outputs = np.array([
                         self.hidden_layer[j].predict(x[j % len(x)])
                         for j in range(3)])
        return self.output_neuron.predict(np.sum(hidden_outputs))
```

Таким чином, модель двошарового персептрону забезпечує передбачення на основі обробки вхідних даних через прихований та вихідний шари.

Ця структура програмного забезпечення дозволяє виконувати навчання та передбачення для задач різної складності, від класичного нейрону до двошарового персептрону зі складнішою архітектурою.

3 Результати роботи програми

Результати запуску програми наведені нижче:

Пункт 1: Класичний нейрон

```
Вхід: 0.0, Вихід: 0.096655983652895 67
Вхід: 0.1, Вихід: 0.10079679317337406
```

Пункт 2: Елементарний двошаровий персептрон

```
Вхід: 0.0, Вихід: 0.10039114630197979
Вхід: 0.1, Вихід: 0.10011309213589634
```

Пункт 3: Двошаровий персептрон (2-3-1)

```
Вхід: [0. 0.], Вихід: 0.17646590879110607
Вхід: [0.  0.1], Вихід: 0.17706798669346935
Вхід: [0.1 0. ], Вихід: 0.17889546089243516
Вхід: [0.2 0.3], Вихід: 0.18323159924259028
```

4 ВИСНОВКИ

У ході виконання лабораторної роботи було розроблено програмне забезпечення для реалізації різних варіантів штучних нейронних мереж: класичного нейрону, елементарного двошарового персептрону зі структурою 1-1-1 та двошарового персептрону із структурою 2-3-1. Всі нейрони у мережах використовували сигмоїдальну функцію активації, яка дозволяє моделювати нелінійні залежності.

Виконані завдання

Основні задачі, що були виконані в ході роботи, включали:

- Розробку коду для класичного нейрону з одним входом, що здатен навчатися на одному прикладі та виконувати передбачення;
- Реалізацію елементарного двошарового персептрону з одним нейроном у прихованому шарі, здатного навчатися на одному прикладі та виконувати передбачення;
- Реалізацію двошарового персептрону із структурою 2-3-1, здатного виконувати навчання «ON-LINE» для кожного прикладу та виконувати передбачення.

Отримані результати та їх аналіз

Отримані результати демонструють, що всі реалізовані моделі здатні коректно навчатися та робити передбачення:

- Для класичного нейрону отримані результати показали здатність моделі коректно відображати лінійні залежності, однак її обмеженість у задачах з нелінійними зв’язками.

- Елементарний двошаровий персептрон (1-1-1) показав базову здатність моделювати прості залежності через нелінійне поєднання вхідного сигналу і прихованого шару. Однак результати показали, що така модель все ще обмежена для більш складних задач.

- Двошаровий персептрон із структурою 2-3-1 показав здатність вивчати більш складні залежності між вхідними сигналами, що робить його здатним до моделювання задач, які включають нелінійні взаємозв'язки (наприклад, логічну функцію XOR).

Аналіз результатів показує, що збільшення кількості шарів та нейронів у мережі покращує її здатність до вивчення складніших залежностей, проте це також збільшує обчислювальну складність і час навчання. Двошаровий персептрон із структурою 2-3-1 продемонстрував здатність до більш якісного навчання на задачах з нелінійними залежностями, що підтверджує важливість багатошарових архітектур у нейронних мережах.

Таким чином, виконана робота продемонструвала базові можливості двошарових нейронних мереж для задач класифікації та передбачень, а також підкреслила важливість структури та методів навчання для досягнення кращої точності та ефективності. Подальше вдосконалення моделей дозволить отримати більш точні передбачення та застосовувати програму для більш складних прикладних задач.

5 Додаток А (лістинг програми)

```
    import numpy as np

# Сигмоїдна функція активації та її похідна
def sigmoid(x):
```

```
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Клас для класичного нейрону
class Neuron:
    def __init__(self):
        self.weight = np.random.rand()
        self.bias = np.random.rand()

    def train(self, x, y, learning_rate=0.5, epochs=1000):
        for _ in range(epochs):
            pred = self.predict(x)
            error = y - pred
            self.weight += learning_rate * error * x * sigmoid_derivative(
            self.bias += learning_rate * error * sigmoid_derivative(pred)

    def predict(self, x):
        return sigmoid(x * self.weight + self.bias)

# Клас для елементарного двошарового персептрону (1-1-1)
class SimplePerceptron:
    def __init__(self):
        self.hidden_neuron = Neuron()
        self.output_neuron = Neuron()

    def train(self, x, y, learning_rate=0.5, epochs=1000):
        for _ in range(epochs):
            hidden_output = self.hidden_neuron.predict(x)
            pred = self.output_neuron.predict(hidden_output)
            error = y - pred
```

```
            self.output_neuron.weight += learning_rate * error * hidden_ou
            sigmoid_derivative(pred)
            self.output_neuron.bias += learning_rate * error *
            sigmoid_derivative(pred)

            hidden_error = error * self.output_neuron.weight *
            sigmoid_derivative(hidden_output)
            self.hidden_neuron.weight += learning_rate * hidden_error * x
            sigmoid_derivative(hidden_output)
            self.hidden_neuron.bias += learning_rate * hidden_error *
            sigmoid_derivative(hidden_output)

    def predict(self, x):
        hidden_output = self.hidden_neuron.predict(x)
        return self.output_neuron.predict(hidden_output)

# Клас для двошарового персептрону із структурою 2-3-1
class TwoLayerPerceptron:
    def __init__(self):
        self.hidden_layer = [Neuron() for _ in range(3)]
        self.output_neuron = Neuron()

    def train(self, x, y, learning_rate=0.3):
        for epoch in range(1000):
            for i in range(len(x)):
                hidden_outputs = np.array([self.hidden_layer[j].predict(x[
                len(x[i])]) for j in range(3)])

                pred = self.output_neuron.predict(np.sum(hidden_outputs))
                error = y[i] - pred

                self.output_neuron.weight += learning_rate * error *
```

```
                np.sum(hidden_outputs) * sigmoid_derivative(pred)
                self.output_neuron.bias += learning_rate * error *
                sigmoid_derivative(pred)

                for j, neuron in enumerate(self.hidden_layer):
                    hidden_error = error * self.output_neuron.weight *
                    sigmoid_derivative(hidden_outputs[j])
                    neuron.weight += learning_rate * hidden_error * x[i][j
                    len(x[i])] * sigmoid_derivative(hidden_outputs[j])
                    neuron.bias += learning_rate * hidden_error *
                    sigmoid_derivative(hidden_outputs[j])

    def predict(self, x):
        hidden_outputs = np.array([self.hidden_layer[j].predict(x[j % len(
        for j in range(3)])
        return self.output_neuron.predict(np.sum(hidden_outputs))

# Приклади роботи для пунктів 1 та 2

# Пункт 1: Тестування класичного нейрону
print("Пункт 1: Класичний нейрон")
neuron = Neuron()
x_train = np.array([0, 0.1])  # Вхідні значення
y_train = np.array([0, 0.1])  # Очікуваний результат наприклад(, реалізаці

# Навчання нейрону
neuron.train(x_train[1], y_train[1])
for x in x_train:
    print(f"Вхід: {x}, Вихід: {neuron.predict(x)}")

# Пункт 2: Тестування елементарного двошарового персептрону (1-1-1)
print("\Пунктn 2: Елементарний двошаровий персептрон")
```

```
simple_perceptron = SimplePerceptron()
simple_perceptron.train(x_train[1], y_train[1])
for x in x_train:
    print(f"Вхід: {x}, Вихід: {simple_perceptron.predict(x)}")

# Пункт 3: Тестування двошарового персептрону (2-3-1)
print("\Пунктn 3: Двошаровий персептрон (2-3-1)")
perceptron = TwoLayerPerceptron()
x_train = np.array([[0, 0], [0, 0.1], [0.1, 0], [0.2, 0.3]])  # Вхідні зна
 двох змінних
y_train = np.array([0, 0.1, 0.1, 0.5])  # Очікуваний результат наприклад(
perceptron.train(x_train, y_train)

# Перевірка результатів для двошарового персептрону
for x in x_train:
    print(f"Вхід: {x}, Вихід: {perceptron.predict(x)}")
```