КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Кафедра інтелектуальних та інформаційних систем

Лабораторна робота № 5

з дисципліни

“Нейромережні технології та їх застосування”

Виконав студент
групи КН-31
Пашковський Павло Володимирович

Київ-2021

Контрольні питання

1. Класи завдань, які вирішуються нейронними мережами?

Мережа може бути навчена рішенням різних прикладних задач - апроксимації функцій, ідентифікації та управління об'єктами, розпізнавання образів, класифікації об'єктів і т. п.

2. Що таке активаційна функція? Типи активаційних функцій.

Функція активації (activation function) fa(•) описує правило переходу нейрона, що перебуває в момент часу k у стані z(k), у новий стан z(k + 1) при надходженні вхідних сигналів x

$$z(k+1) = f_a\,(z(k),\, f_{вх}\,(x,w)).$$

Лінійні:

- Лінійна
- Лінійна біполярна з насиченням
- Лінійна уніполярна з насиченням

Нелінійні:

- Уніполярна порогова функція
- Біполярна порогова функція

Сигмоїдальні функції:

- Логістична (уніполярна)
- Гіперболічного тангенса (біполярна)
- Синусоїдальна з насиченням (біполярна)
- Косинусоїдальна з насиченням (уніполярна)

3. Топології штучних нейронних мереж.

ШНМ поділяються на ті, що містять зворотні зв’язки, та ті, що не містять.

ШНМ без зворотних зв'язків (прямого поширення, Feed forward):

- першого порядку
- другого порядку (з «shortcut connections»)

ШНМ зі зворотними зв'язками (зворотного поширення, рекурентні, Feedback):

- з прямими зворотними зв'язками (direct feedback)
- з непрямими зворотними зв'язками (indirect feedback)
- з латеральними зв'язками (lateral feedback)
- повнозв'язні.

4. Типи алгоритмів навчання.

Перший тип навчання припускає, що є «учитель», що задає пари, які навчають — для кожного вхідного вектора, що навчає, необхідний вихід мережі. Для кожного вхідного вектора, що навчає, обчислюється вихід мережі, порівнюється з відповідно необхідним, визначається похибка виходу, на основі якої й коректуються ваги.

Пари, що навчають, подаються мережі послідовно й ваги уточнюються доти, поки похибка за такими парами не досягне необхідного рівня. Цей вид навчання неправдоподібний з біологічної точки зору. Дійсно, важко уявити зовнішнього «учителя» мозку, що порівнює реальні й необхідні реакції того, кого навчають, і коригує його поведінку за допомогою негативного зворотного зв’язку. Більш природним є навчання без учителя, коли мережі подаються тільки вектори вхідних сигналів, і мережа сама, використовуючи деякий алгоритм навчання, підстроювала б ваги так, щоб при поданні їй досить близьких вхідних векторів вихідні сигнали були б однаковими. У цьому випадку в процесі навчання виділяються статистичні властивості множини вхідних векторів, що навчають, і відбувається об’єднання близьких (подібних) векторів у класи. Подання мережі

вектора з даного класу викликає її певну реакцію, яка до навчання є непередбаченою.

5. Градієнтні алгоритми навчання.

1) Алгоритм GD, або алгоритм градієнтного спуску, використовується для такого коригування ваг і зміщень, щоб мінімізувати функціонал помилки, тобто забезпечити рух по поверхні функціоналу в напрямку, протилежному градієнту функціоналу по параметрам.

2) Алгоритм GDM, або алгоритм градієнтного спуску з обуренням, призначений для настройки і навчання мереж прямої передачі. Цей алгоритм дозволяє долати локальні нерівності поверхні помилки і не зупинятися в локальних мінімумах. З урахуванням обурення метод зворотного поширення помилки реалізує наступне співвідношення для збільшення вектора параметрів, що настроюються:

$$\Delta w_k = mc\Delta w_{k-1} + (1 - mc)lrg_k, \text{де}$$

Δw_k- приріст вектора ваг; mc - параметр обурення; lr - параметр швидкості навчання; gk - вектор градієнта функціоналу помилки на k -й ітерації.

3) Алгоритм GDA, або алгоритм градієнтного спуску з вибором параметра швидкості настройки, використовує евристичну стратегію зміни цього параметра в процесі навчання.

4) Алгоритм Rprop, або пороговий алгоритм зворотного поширення помилки, реалізує наступну евристичну стратегію зміни кроку збільшення параметрів для багатошарових нейронних мереж.

6. Алгоритми методу сполучених градієнтів

В алгоритмах сполученого градієнта пошук мінімуму виконується вздовж пов'язаних напрямків, що забезпечує зазвичай

більш швидку збіжність, ніж при якнайшвидшому спуску. Всі алгоритми сполучених градієнтів на першій ітерації починають рух в напрямку антіградієнта.

Основний алгоритм зворотного поширення помилки коригує настроюються параметри в напрямку найшвидшого зменшення функціоналу помилки. Але такий напрямок далеко не завжди є найсприятливішим напрямком, щоб за можливе мале число кроків забезпечити збіжність до мінімуму функціоналу. Існують напрямки руху, рухаючись по яких можна визначити шуканий мінімум набагато швидше. Зокрема, це можуть бути так звані зв'язані напрямки, а відповідний метод оптимізації – це методу сполучених градієнтів.

Якщо в навчальних алгоритмах градієнтного спуску, управління сходимостью здійснюється за допомогою параметра швидкості настройки, то в алгоритмах методу сполучених градієнтів розмір кроку коригується на кожній ітерації. Для визначення розміру кроку уздовж сполученого напрямку виконуються спеціальні одномірні процедури пошуку мінімуму.

7. Квазіньютонови алгоритми.

1) Альтернативою методу сполучених градієнтів для прискореного навчання нейронних мереж служить метод Ньютона. Основний крок цього методу визначається співвідношенням

$$x_{k+1} = x_k - H_k^{-1} g_k$$

де x_k - вектор параметрів, що настроюються; H_k - матриця Гессе друге приватних похідних функціоналу помилки по параметрам; g_k - вектор градієнта функціонала помилки. Процедури мінімізації на основі методу Ньютона, як правило, сходяться швидше, ніж ті ж процедури на основі методу сполучених градієнтів. Однак обчислення матриці Гессе - це дуже складна і дорога в обчислювальному відношенні процедура. Тому розроблений клас алгоритмів, які засновані на методі Ньютона, але не вимагають

обчислення других похідних. Це класс квазіньютонових алгоритмів, які використовують на кожній ітерації деяку наближену оцінку матриці Гессе.

2) Алгоритм OSS (One Step Secant), або однокроковий алгоритм методу січних площин, описаний в роботі Баттіті (Battiti). У ньому зроблена спроба об'єднати ідеї методу сполучених градієнтів і схеми Ньютона, тому в бібліотек Neurolab він названий Newton Conjugate Gradient Method. Алгоритм не запам'ятовує матрицю Гессе, вважаючи її на кожній ітерації рівній одиничною. Це дозволяє визначати новий напрямок пошуку не обчисляючи зворотну матрицю.

Індивідуальне завдання:

Варіант 6. y = cos(2x) –sin(7x) – tanh(2x);

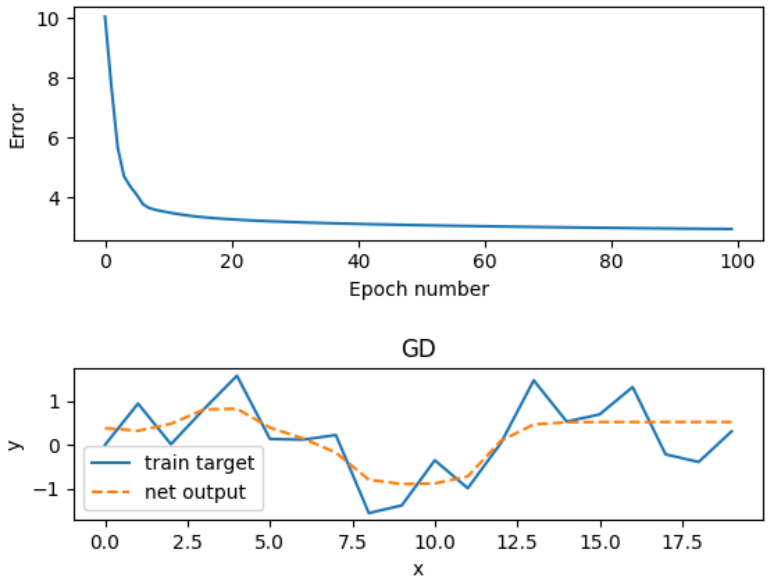

Рисунок 1 – Алгоритм GD

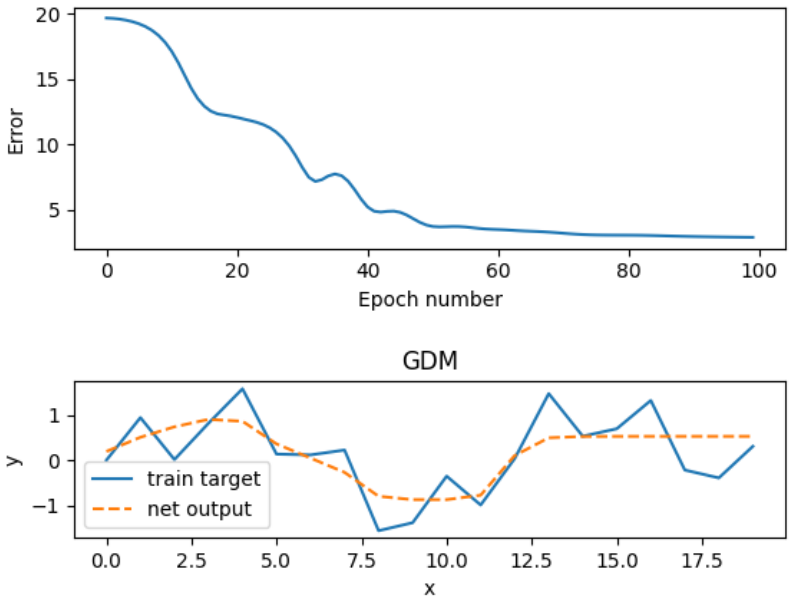

Рисунок 2 – Алгоритм GDM

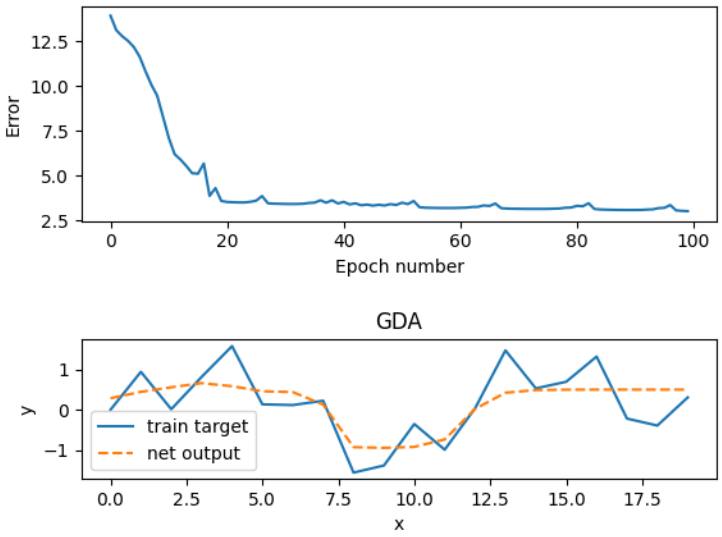

Рисунок 3 – Алгоритм GDA

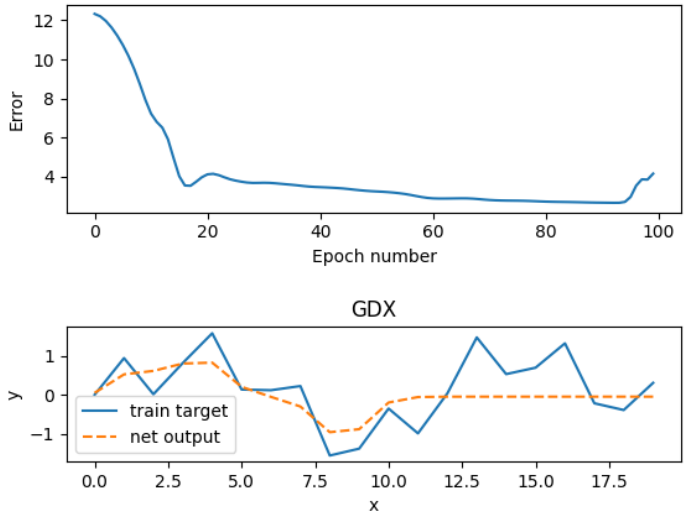

Рисунок 4 – Алгоритм GDX

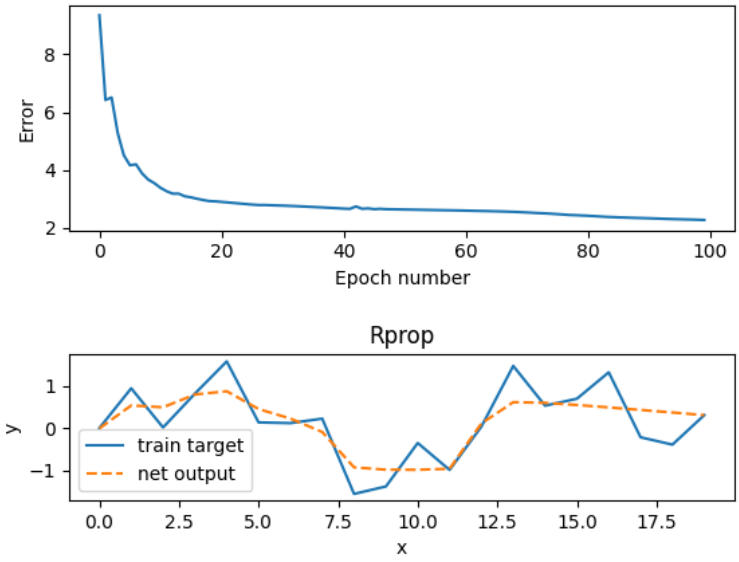

Рисунок 5 – Алгоритм Rprop

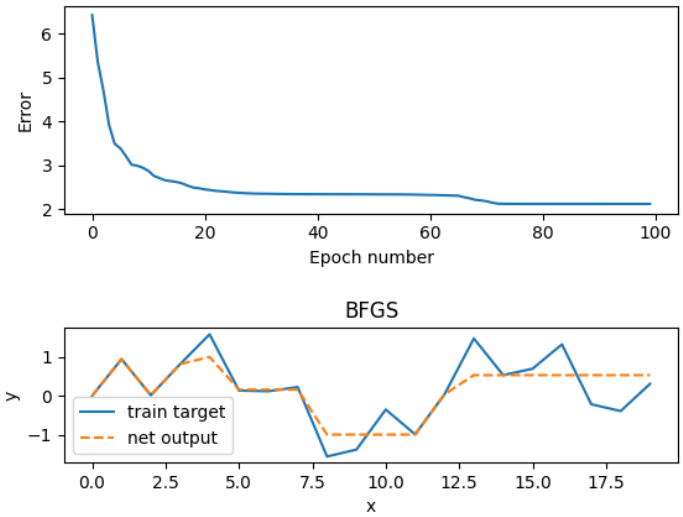

Рисунок 6 – Алгоритм BFGS

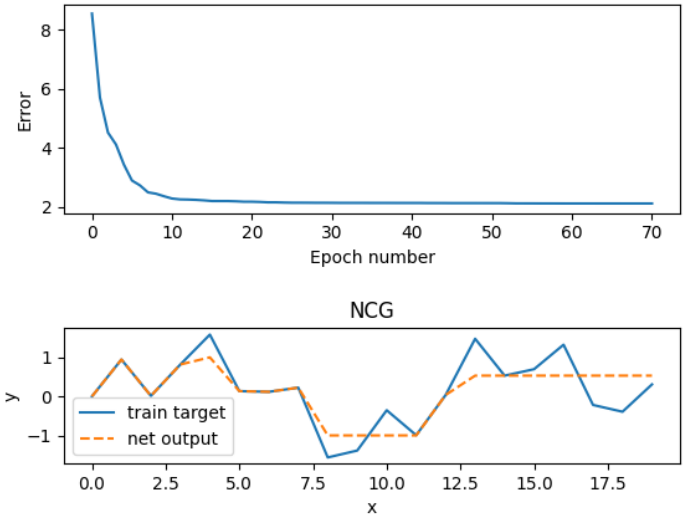

Рисунок 7 – Алгоритм NCG

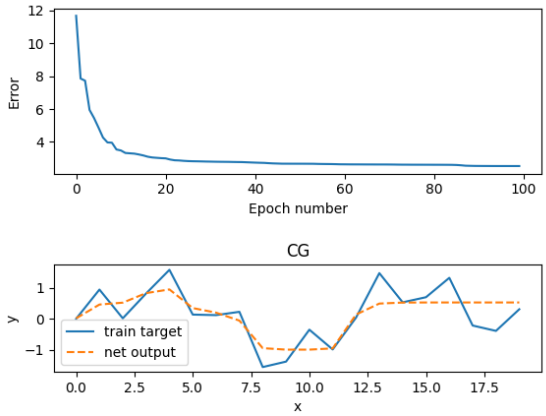

Рисунок 8 – Алгоритм CG

Висновок:

У даній роботі було створено програму для апроксимації функції. Навчання відбувалося за допомогою бібліотеки Neuralab.

Результати програми можна практично застосовувати для розділення класів.

Отримані навички щодо навчання мережі можна використовувати в майбутньому для аналізу та класифікації даних.

Код програми:

```
import neurolab as nl
import numpy as np
import os
import pylab as pl


def getTrainName(net_trainf):
    net_name = str(net.trainf.__dict__['_train_class']).split('.')[-1]
    net_name = net_name.replace("'>","")
    net_name = net_name.split('Train')[-1]

    return net_name


methods_dict = {1:"GD",2:"GDM", 3:"GDA", 4:"GDX", 5:"RPROP", 6:"BFGS",
7:"NCG", 8:"CG"}

x = np.linspace(0,20,20)
y = np.cos(2 * x) - np.sin(7 * x) + (np.tanh(2 * x))
max_value = np.max(x)
min_value = np.min(x)
size = len(x)
x_train = x.reshape(size, 1)
y_train = y.reshape(size,1)

if  os.path.exists('saved_weights'):
    print("1 - GD")
    print("2 - GDM")
    print("3 - GDA")
    print("4 - GDX")
    print("5 - RPROP")
    print("6 - BFGS")
    print("7 - NCG")
    print("8 - CG")

    number = int(input("Input one of netword: "))
    name_method = methods_dict[number]
    net = nl.load(f'saved_weights/{name_method}.net')
    out = net.sim(x_train)
    print(f"Out: {out}")
    print(f"Target: {y_train}")

    pl.plot(out, '-', y_train, '--o')
    pl.legend(['train target', 'net output'])
    pl.show()
else:
    trainings =
['nl.train.train_gd','nl.train.train_gdm','nl.train.train_gda',

'nl.train.train_gdx','nl.train.train_rprop','nl.train.train_bfgs','nl.train
.train_ncg','nl.train.train_cg']
    outputs = []
    outputs_dict = {}
    os.mkdir('saved_weights')

    for train in trainings:
        net = nl.net.newff([[min_value,max_value]],[10,1])
        exec(f"net.trainf = {train}")

        net_name = getTrainName(net.trainf)

        print(net_name)
        print("------------------")
```

```
        error = net.train(x_train, y_train, epochs=100, show=15,goal=
0.00001)

        net.save(f'saved_weights/{net_name}.net')

        out = net.sim(x_train)
        outputs.append(out)
        outputs_dict[f'{net_name}'] = out
        print(f"Errors: {error}")
        print(y_train)
        print(out)

        pl.subplot(211)
        pl.plot(error)
        pl.xlabel('Epoch number')
        pl.ylabel('Error')
        pl.subplot(313)
        pl.plot(y_train, '-', out, '--')
        pl.xlabel('x')
        pl.ylabel('y')
        pl.title(f"{net_name}")
        pl.legend(['train target', 'net output'])
        pl.show()

    for out in outputs_dict.values():
        pl.plot(out)
    pl.plot(y_train,'--')

    outputs = list(outputs_dict.keys())
    outputs.append('target')
    print(outputs)
    pl.legend(outputs)
    pl.show()
```