

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СІПЕНКО

«__» _____ 2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Нейронна мережа з розпізнавання чисел та арифметичних операцій»

Виконав (-ла):

студент (-ка) IV курсу, групи ІО-62

Смирнов Михайло Анатолійович _____

Керівник:

Старший викладач

Луцький Георгій Михайлович _____

Консультант з нормоконтролю:

Професор кафедри ОТ, д.т.н.,

Сімоненко Валерій Павлович _____

Рецензент:

к.т.н, доцент кафедри АУТС

Олег Євгенович Сокульський _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент (-ка) _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій СТИПЕНКО

«__» _____ 2020 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Смирнова Михайла Анатолійовича

1. Тема проєкту «Нейронна мережа з розпізнання чисел та арифметичних операцій», керівник проєкту Луцький Георгій Михайлович, старший викладач, затверджені наказом по університету від «07» травня 2020 р. № 1081-с
2. Термін подання студентом проєкту 26 травня 2020р.
3. Вихідні дані до проєкту: теоретичні дані, технічна документація.
4. Зміст пояснювальної записки огляд існуючих технології для вирішення питання, теоретичні відомості щодо тематики проєкту, математична реалізація алгоритмів, програмна реалізація.
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) структурна схема, блок схема.

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
нормоконтроль	Сімоненко В. П., професор, д.т.н.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	<i>Затвердження теми роботи</i>	<i>01.09.2019</i>	
2.	<i>Вивчення та аналіз завдання</i>	<i>15.12.2019-15.03.2020</i>	
3.	<i>Розробка архітектури додатку</i>	<i>15.03.2020-25.03.2020</i>	
4.	<i>Написання програмної частини</i>	<i>25.03.2020-05.04.2020</i>	
5.	<i>Тестування та виправлення помилок</i>	<i>05.04.2020-15.04.2020</i>	
6.	<i>Оформлення пояснювальної записки</i>	<i>15.04.2020-20.05.2020</i>	
7.	<i>Захист програмного продукту</i>	<i>25.04.2020</i>	
8.	<i>Передзахист</i>	<i>26.05.2020</i>	
9.	<i>Захист</i>	<i>25.06.2020</i>	

Студент

Керівник

Михайло СМІРНОВ

Георгій ЛУЦЬКИЙ

Анотація

У даній роботі проводиться розробка нейронної мережі з розпізнавання чисел і арифметичних операцій над ними. У зв'язку із зростанням затребуваності застосування технологій нейронних мереж, для вирішення ряду завдань. Які знаходять своє застосування в розпізнаванні образів, прогнозуванні, а також аналізу даних, автоматизації і тощо. Постійно виникають нові завдання які вимагають високоточної рішення за допомогою тих чи інших математичних алгоритмів, одним з яких можна виділити нейронну мережу.

Оскільки нейронна мережа, дуже близька по моделі роботи нейронної мережі людського мозку, тобто до тренувань, а точніше навчання і в подальшому коригування своїх помилок. То в даній роботі запропоновано варіант алгоритму розпізнавання чисел. Модель в даній роботі буде розпізнавати числа з картинок запропонованих їй з великого набору даних і відображення результату.

Аннотация

В данной работе проводится разработка нейронной сети по распознаванию чисел и арифметических операций над ними. В связи с ростом востребованности применения технологий нейронных сетей, для решения ряда задач. Которые находят свое применение в распознавании образов, прогнозировании, а также анализа данных, автоматизации и т.д. Постоянно возникают новые задачи которые требуют высокоточного решения с помощью тех или иных математических алгоритмов, одним из которых можно выделить нейронную сеть.

Поскольку нейронная сеть, очень близка по модели работы нейронной сети человеческого мозга, то есть к тренировкам, а точнее обучению и в дальнейшем корректирования своих ошибок. То в данной работе предложен вариант алгоритма распознавания чисел. Модель в данной работе будет распознавать числа с картинок предложенных ей с большого набора данных и отображения результата.

Abstract

In this paper, we develop a neural network for the recognition of numerical and arithmetic operations on them. Due to the growing demand for the use of neural network technologies, to solve a number of problems. Data analysis, automation, etc. Constantly existing new tasks that require high-precision solutions using certain mathematical algorithms.

The neural network has everything you need for training, for accurate training and further correction of your mistakes. Number recognition algorithm. In this work, recognition of the number of results will be established.

[illegible]

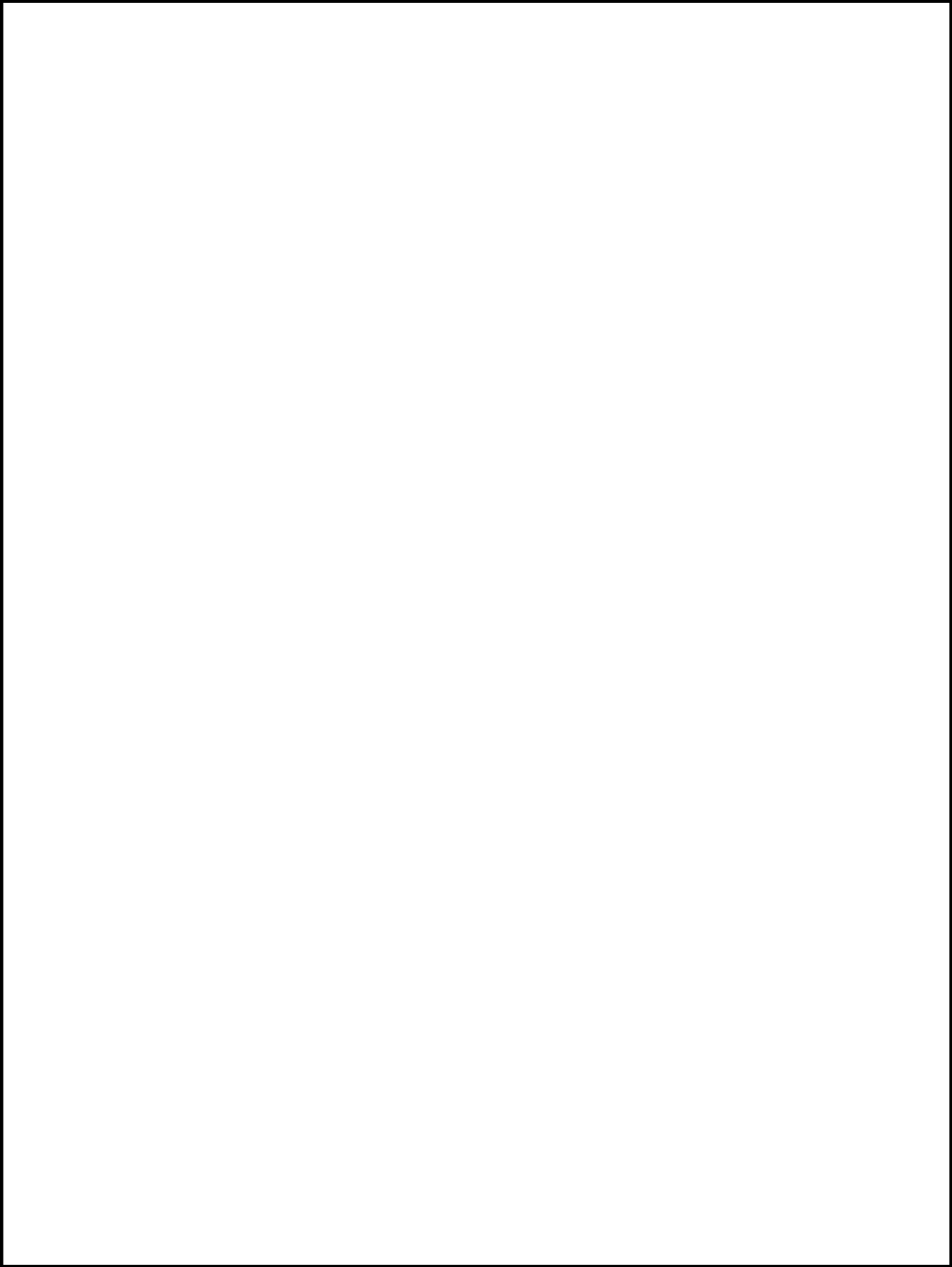
					ДП.6224.01.000 ВП								
Зм.		№ документа	Підп.	Дата	Відомість Дипломного проекту					Літ.	Аркуш		
Розробив		Смирнов М.А.								Т		1	1
Перевір.		Луцький Г.М.								НТУУ «КПІ імені Ігоря Сікорського», ФІОТ			
										Група ІО - 62			
Н.конт		Симоненко В.П.											
Затв.		Стіренко С.Г											

Технічне завдання
до дипломного проєкту
на тему «Нейронна мережа з розпізнання чисел та арифметичних
операції»

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ.....	2
2. ПІДСТАВИ ДЛЯ РОЗРОБКИ	2
3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ.....	2
4. ДЖЕРЕЛА РОЗРОБКИ	2
5. ТЕХНІЧНІ ВИМОГИ	2
5.1. Вимоги до програмного продукту що розробляється	2
6. ЕТАПИ РОЗРОБКИ	3

					<i>ДП.6224.02.000 ТЗ</i>			
Зм.		№ документа	Підп.	Дата				
Розробив	Смирнов М.А				Нейронна мережа з розпізнання чисел та арифметичних операцій Технічне завдання	Літ.	Аркуш	Аркушів
Перевірів	Луцький					Т	1	3
Н.конт	Симоненко В.П.					НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 62		
Затв.	Стіренко С.Г.							



					<i>ДП.6224.02.000 ТЗ</i>			
Зм.		№ документа	Підп.	Дата				
Розробив	Смирнов М.А				<i>Нейронна мережа з розпізнання чисел та арифметичних операцій</i> <i>Технічне завдання</i>	Літ.	Аркуш	Аркушів
Перевірів	Луцький					Т	1	3
Н.конт	Симоненко В.П.					<i>НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 62</i>		
Затв.	Стіренко С.Г.							

1.НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: «Нейронна мережа з розпізнання чисел та арифметичних операції».

Область застосування: дана програма може бути використана для розпізнання рукописного тексту, зокрема в таких сферах як сфера обслуговування, банківська сфера.

2.ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського дипломного проекту, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського».

3.МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є програмний продукт, який здатен буде розпізнавати рукописні числа.

4.ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література та публікації в спеціалізовані публікації в Інтернеті з даних питань.

5.ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Програмний продукт, що розробляється повинен: вміти розпізнавати числа, також необхідним є те щоб продукт міг навчатися та на кожному етапі навчання підвищувати точність розпізнання.

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення необхідної літератури	19.02.2019
Складання і узгодження технічного завдання	06.03.2019
Написання вступної частини та огляд рішень	19.03.2019
Розробка структури та визначення можливостей сайту	03.04.2019
Написання програмної частини	09.04.2019
Тестування та виправлення помилок	07.05.2019
Оформлення документації дипломного проекту	16.05.2019
Попередній захист та проходження нормативного контролю	30.05.2019
Захист дипломного проекту	15.06.2019

Пояснювальна записка
до дипломного проєкту
на тему: «Нейронна мережа з розпізнавання чисел та арифметичних
операції»

Київ – 2020 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	4
ВСТУП.....	5
РОЗДІЛ 1	8
ОГЛЯД РОБОТИ ТА АРХІТЕКТУРИ НЕЙРОННОЇ МЕРЕЖІ	8
1.1. Персептрон	8
1.2. Сигмовидний нейрон.....	14
1.3. Архітектура нейронних мереж	19
1.4. Проста мережа для класифікації рукописних цифр	22
ВИСНОВКИ ДО РОЗДІЛУ 1	29
РОЗДІЛ 2	30
НАВЧАННЯ І ТРЕНУВАННЯ НЕЙРОННОЇ МЕРЕЖІ	30
2.1 Набір даних MNIST.....	30
2.2 Функція витрат	31
2.3 Градієнтний спуск	32
2.4 Стохастичний градієнтний спуск	38
2.5 Алгоритм зворотного розповсюдження.....	40
ВИСНОВКИ ДО РОЗДІЛУ 2.....	42
РОЗДІЛ 3	43
РЕАЛІЗАЦІЯ НЕЙРОННОЇ МЕРЕЖІ ДЛЯ КЛАСИФІКАЦІЇ ЦИФР ТА АРИФМЕТИЧНИХ ДІЙ НАД НИМИ.....	43
3.1. Реалізація класу нейронної мережі	43
3.1.1 Тренування нейронної мережі	46
3.1.2 Реалізація алгоритму зворотного розповсюдження	48

					ДП 6224. 03.000 ПЗ		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив	Смирнов М.А				Літ.	Аркуш	Аркушів
Перевір.	Луцький Г.М					2	61
Н. контр.	Симоненко В.П.				Пояснювальна записка НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ гр. ІО-62		
Затверд.	Стіренко С.Г						

3.2. Реалізація файлу з функціями для підключення набору даних	51
ВИСНОВКИ ДО РОЗДІЛУ 3	55
РОЗДІЛ 4	56
АНАЛІЗ ГОТОВОЇ МЕРЕЖІ ТА ЇЇ РОБОТИ.....	56
4.1 Огляд роботи реалізованого проекту	56
ВИСНОВКИ ДО РОЗДІЛУ 4.....	58
ВИСНОВКИ	59
ПЕРЕЛІК ПОСИЛАНЬ	60
ДОДАТОК А. Структурна схема	63
ДОДАТОК Б. Функціональна схема	64
ДОДАТОК В. Блок-схема алгоритму	65

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

MNIST - Modified National Institute of Standards and Technology

ООП - Об'єктно-орієнтоване програмування

					<i>ДП 6224. 03.000 ПЗ</i>	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

В останні роки спостерігається стрімке поширення інтересу до технологій пов'язаних з нейронними мережами. Нині вони активно застосовуються в різних областях, таких як - медицина, фізика, інженерія тощо. Також нейронні мережі широко застосовуються в галузях де необхідно вирішувати завдання аналізу або наприклад прогнозування. Такий успіх обумовлено деякими факторами.

По-перше це багаті можливості. Нейронні мережі достатньо потужний метод проектування який надає можливість відтворити залежності, які людство не може вирішити без технічної допомоги. За собою нейронна мережа є нелінійною. Багату кількість років у моделювання використовувалось у більшості сферах вирішення задач. Оскільки для лінійних задач є вже готові процедури оптимізації. Але нейронні мережі дозволяють вирішити проблему розмірності.

По-друге нейронні мережі достатньо просто використовувати. Тренування нейронної мережі відбувається на певних прикладах. Тобто користувач, який користується нейронною мережею підбирає певні дані, які знаходяться в достатньо великому наборі даних. Наприклад архів на основі бази даних MNIST (Modified National Institute of Standards and Technology). Від користувача потребується мінімальний набір знань, а саме, як підготувати дані, яку архітектуру мережі вибрати. Нейронні мережі які використовуються для вирішення практичних завдань були спроектовані на основі моделі біологічної нервової системи. В майбутньому якщо ми будемо спостерігати такий стрімкий розвиток нейронних мереж можливо ми станемо свідками дійсно високошвидкісного та точного штучного інтелекту. Отже, оскільки як було сказано вище, що нейронна мережа використовується також в штучному інтелекті, то в цієї роботі буде розглянуто одну з таких, а саме – розпізнавання чисел та арифметичних операції над ними.

Метою дослідження є поліпшення точності нейронної мережі для розпізнавання рукописних символів на основі програмного продукту, який повинен бути реалізований як, консольний додаток, який на вхід буде отримувати партію рукописних чисел, та мережа повинна бути розпізнавати й видавати, точність. Нашою метою є що б на кожній епосі навчання мережі, точність розпізнавання рукописних символів, покращувалася. Та на кінцевому етапі навчання, ми отримуємо процент точності який вище інших реалізованих проектів на дану тематику. Що і є нашою головною метою дослідження.

Засоби реалізації. Засобами реалізації цієї роботи, є складні математичні алгоритми, зокрема ми скористаємося не просто градієнтним спуском, а стохастичним градієнтним спуском. Цим алгоритмом ми будемо користуватися для тренування мережі. Саме це спосіб надасть нам можливість отримати ефективніші результати у порівнянні з іншими мережами, звертаючи увагу на цю тематику.

Дипломна робота дозволяє розпізнавати рукописні числа, які будуть писати звичайні користувачі різноманітних професії. Тобто наша зробити висновки з отриманих результатів, та прослідкувати, як саме працює мережа у моменті її тренування.

Задачі дослідження. Виходячи з головної мети цієї дипломної роботи, впливають наступні задачі:

- рішення проблеми ефективного навчання мережі;
- огляд існуючих математичних та програмних рішень подібних проблем, які вже зустрічалися в схожих проектах.
- огляд існуючих проблем при розпізнанні символів;
- огляд існуючих проблем які виникають при тренуваннях мереж, та надання їм більшої точності;
- розглянути можливі спрощення об'єму розробки проекту за допомогою вбудованих та сторонніх бібліотек у використаних засобах реалізації.

Наукова новизна дослідження. Науковою новизною цієї роботи є те що, запропоновано новий метод стохастичного градієнтного спуску, який було

					<i>ДП 6224. 03.000 ПЗ</i>	Арк.
						6
Зм.	Арк.	№ докум.	Підпис	Дата		

модифіковано. Саме така модифікація та поліпшення алгоритму стохастичного градієнтного спуску, дозволяє досягти нам більш точних результатів, та буде відрізнятися від попередньо реалізованих мереж, тим що наша мережа буде розпізнавати рукописні числа з більшою точністю та швидкістю.

					ДП 6224. 03.000 ПЗ	Арк.
						7
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1

ОГЛЯД РОБОТИ ТА АРХІТЕКТУРИ НЕЙРОННОЇ МЕРЕЖІ

Проблема у вирішенні задач, які реалізують розпізнання рукописного тексту, полягає у тому що не вдається одразу так просто сформулювати алгоритм. Бо при спробах зробити алгоритм, виникає купа винятків, та унікальних випадків.

Нейронні мережі розглядають цю проблему з іншого боку. Вирішення цієї задачі складається з того, що береться велика кількість рукописних символів, їх ми вважаємо навчальними, а вже потім розробляємо модель, яка буде вчитися на цих навчальних прикладах. Таким чином нейронна мережа буде використовувати приклади для того щоб вивести певні правила розпізнавання рукописних символів. Чим більше наша система продивиться кількість навчальних прикладів, тим краще ми зможемо підвищити її точність. У даній дипломній роботі буде розглянуто важливі типи штучних нейронів, а саме персептрон і сигмовидний нейрон. А також алгоритм тренування нейронної мережі - стохастичний градієнтний спуск. [4]

1.1. Персептрон

Персептрон працює наступним чином. Він приймає декілька двійкових входів, та на виході видає двійковий вихід.

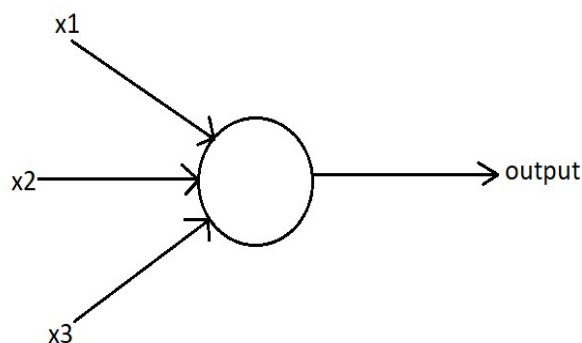


Рис. 1.1. Схема роботи персептрона

На малюнку 1.1 зображений персептрон який має три входи, та один вихід. Взагалі він може мати меншу або більшу кількість входів. При розробці такої моделі було запропоноване правило для обчислення результату. Введено

деякі вагові коефіцієнти які позначимо як $w_1, w_2, \dots, w_N$. Ці вагові коефіцієнти відповідають важливості вхідних даних для того щоб зробити вивід. Результат виводу, який буде 0 або 1, буде визначатися відповідно до вагової суми $\sum_j w_j x_j$, тобто, менше або більше це значення буде відповідно до деякого порогового значення. Ваги та поріг являють собою дійсні числа. Це можна виразити через алгебраїчний вираз:

$$Вивід = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{порогове значення} \\ 1 & \text{if } \sum_j w_j x_j > \text{порогове значення} \end{cases}$$

Рис. 1.2. Математична модель роботи персептрона

Звісно персептрон не являє собою дійсною моделлю яка відображує прийняття рішень людиною. Але ця система ілюструє, як можна зважити деякі види факторів для прийняття рішень. Проте модель мережі персептронів здатна розглядати та приймати більш важкі рішення.

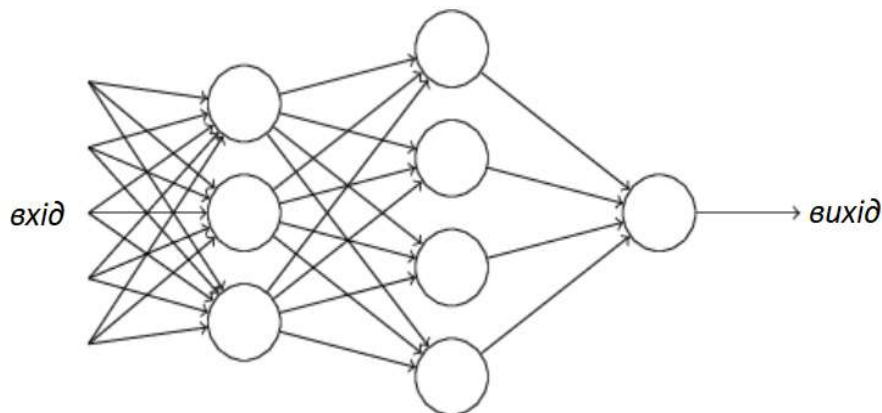


Рис. 1.3. Математична модель мережі персептронів

Якщо подивитись на малюнок 1.3 то можна побачити декілька стовбців персептронів, ці стовпці прийнято називати шарами. Перший шар здатний приймати рішення на основі зважування вхідних даних. В персептронах другого шару приймає рішення на основі зважування результатів першого шару. Отже персептрони другого шару приймає рішення більш складного

рівня ніж в першому шарі. Також якщо розглядати систему даль ніше то ми отримуємо, що на третьому рівні персеπτрони приймають рішення ще складніше. Також, хочу зазначити що коли мова розпочалася про персеπτрон, було визначено, що ця модель має тільки один вихід, але хочу зазначити, що в даній моделі нашої мережі, яка зображена на малюнку 1.3, складається уява проте що, у персеπτронів мається одразу декілька виходів але насправді це також один вихід. Так було позначено для того щоб вказати що один вихід персеπτрона може служити декількома входами для других персеπτронів. [5]

Спростимо опис представлення персеπτрона. При умові виду $\sum_j w_j x_j$ порогове значення здається достатньо громіздкий і ми можемо зробити деякі зміни для прощення цього виразу. Перша зміна яку ми зробимо, це вираз $\sum_j w_j x_j$ ми представимо у вигляді $w^*x \equiv \sum_j w_j x_j$ де w і x вектори, компоненти яких вагами і входами відповідно. Друга зміна перемістити порогове значення на іншу сторону нерівності і замінити його так званим персеπτроним зміщенням $b \equiv \text{порогове значення}$. Використовуючи зміщення замість порога, правило персеπτрона може бути переписано наступним чином:

$$w_{\text{ивид}} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Рис. 1.4. Нове математичне правило для персеπτрона

Таке поняття як зміщення можна пояснити з біологічної точки зору, тобто зміщення це міра того наскільки легко збуджувати наш персеπτрон. Зміщення в деякому сенсі задає деяку зміну в описі персеπτронів. Та подальше ми побачимо, що це призводить до подальших спрощень. В даній роботі я описував персеπτрон як деякий спосіб для зважування факторів для прийняття рішень. Другим способом використання персеπτронів є обчислення деяких елементарних логічних функцій. Тобто такими функціями, як AND, OR, NAND. Припустимо, що в нас мається персеπτрон з двома входами, де

кожен з входів з вагою -2 , та загальним зміщенням 3 . На малюнку 1.5 зображено як буде виглядати наш персептрон в такому разі:

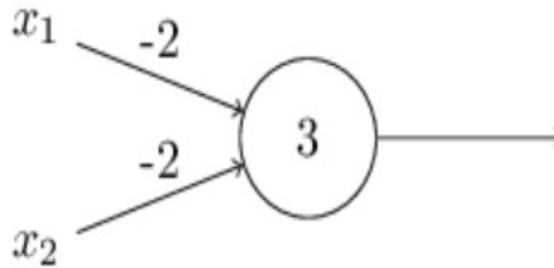


Рис. 1.5. Вигляд персептрона с ваговими коефіцієнтами та загальним зміщенням.

Тоді, ми можемо зрозуміти, що з входами $x_1 = 0$, $x_2 = 0$, результат буде 1. Бо $(-2)*0 + (-2)*0 + 3 = 3$. Результат буде 3, це число позитивне, тому результат виходу буде 1. Такі самі розрахунки з входами 0 та 1 або 1 та 0 дають також результат 1. Проте якщо в нас мається вхід 1 та 1, то вихід буде 0, оскільки $(-2)*1 + (-2)*1 + 3 = -1$. Результат негативний, отже вихід - 0. Тому цей персептрон є реалізацією функції NAND. На прикладі NAND, було продемонстровано, що ми можемо скористатися персептроном для обчислення простої логічної функцій. Насамперед мережі персептронів ми можемо використовувати для обчислення якої завгодно логічної функцій. Причиною цьому є, що NAND є універсальним для обчислення. Наприклад, ми можемо скористатися функцією NAND для того щоб побудувати схему, яка додає два бітових входи x_1 та x_2 . Для цього нам потрібно обрахувати побітову суму, через $x_1 \oplus x_2$, та біт переносу, значення якого встановлено в одиницю, в той час коли x_1 та x_2 дорівнюють 1, тобто біт переносу являє собою побітовий добуток x_1 та x_2 :

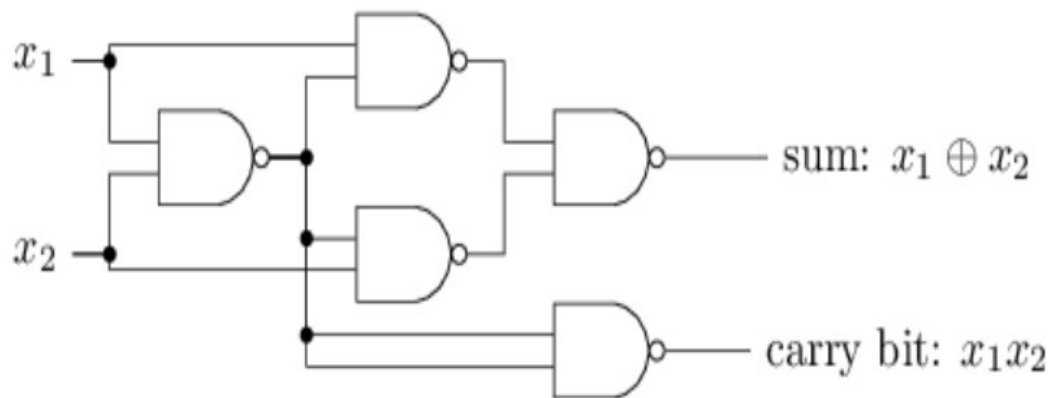


Рис. 1.6. Схема роботи персептрону за допомогою бітового переносу та побітової суми.

Переробимо схему на малюнку 1.6 до еквівалентної схеми, змінюючи усі входи NAND на персептрони з двома входами, де у кожного з них вага являє собою -2 та зміщенням 3. Отримуємо наступну модель:

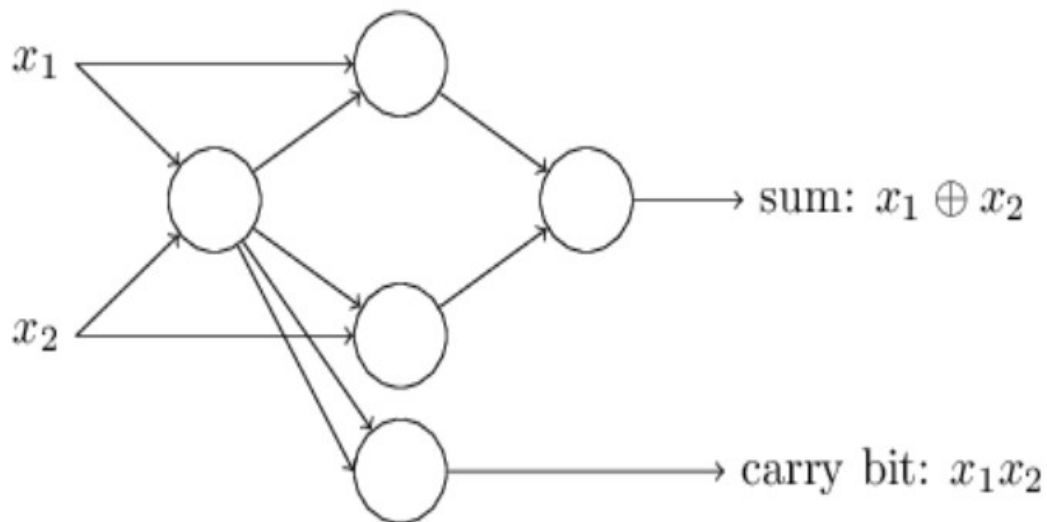


Рис. 1.7. Нова математична модель персептрона за допомогою бітового переносу та побітової суми.

Однією з властивостей цієї мережі являється те, що вихід із нейрона першого шару, буде використовуватися двічі для входу до нижнього персептрону другого шару. Тепер наша мережа виглядає наступним чином: усі невідмічені ваги мають значення -2, зміщення дорівнюють 3, проте одна вага дорівнює -4. [10]

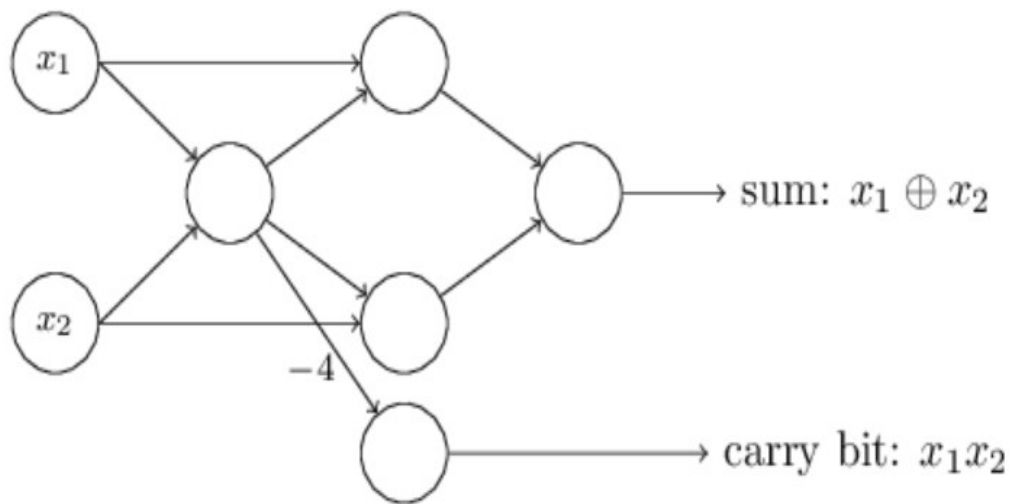


Рис. 1.8. Модель мережі з новим ваговим коефіцієнтом

На малюнку 1.8 вхідні дані x_1 та x_2 було перемальовано так щоб це виглядало як окремий шар персептронів тобто це ж вхідний шар. [11]



Рис. 1.9. Модель персептрона у якого є вихід, проте немає виходу.

Насамперед він має входи, просто в нього немає вагових коефіцієнтів. Ці персептрони можна вважати як звичайні вхідні дані. В універсальності персептронів є свої плюси та мінуси. Плюс полягає в тому що , мережі персептронів, які в свою чергу є різновидом нейронних мере, можуть бути такими ж потужними як і будь-який обчислювальний пристрій. Мінус у тому що здається враження нібито персептрон лише новий тип реалізації NAND.

Однак, ми можемо створити алгоритми тренування, які будуть автоматично підбирати зміщення та вагові коефіцієнти для нашої мережі нейронів. Така автоматизація буде налаштовуватися у відповідь через деякі зовнішні подразники, без будь якого втручання програміста. Саме такий алгоритм тренування даси змогу нам використовувати нашу мережу таким щоб вона вкрай відрізнялася від схеми яка була схожу на реалізацію NAND. [16]

1.2. Сигмовидний нейрон

Наприклад у нас є мережа персептронів, яку ми плануємо використовувати, для того щоб вирішувати певні задачі. Уявимо, що вхідні дані для нашої мережі це піксельні дані зі від сканованого зображення рукописної цифри. Та ми хочемо, щоб наша мережа нейронів, вивчила зміщення та ваги, для того щоб у результаті наша мережа правильно розпізнавала цифра. Щоб дізнатися як працює навчання припустимо, ми під коректуємо деяку вагу в нашої мережі. Ми це намагаємося зробити щоб, під коректувавши цю вагу, відповідно викликало деяку зміну на виході нашої мережі. Саме ця властивість і робить здатність нашої мережи навчатися. Схематично це зображено на наступному малюнку:

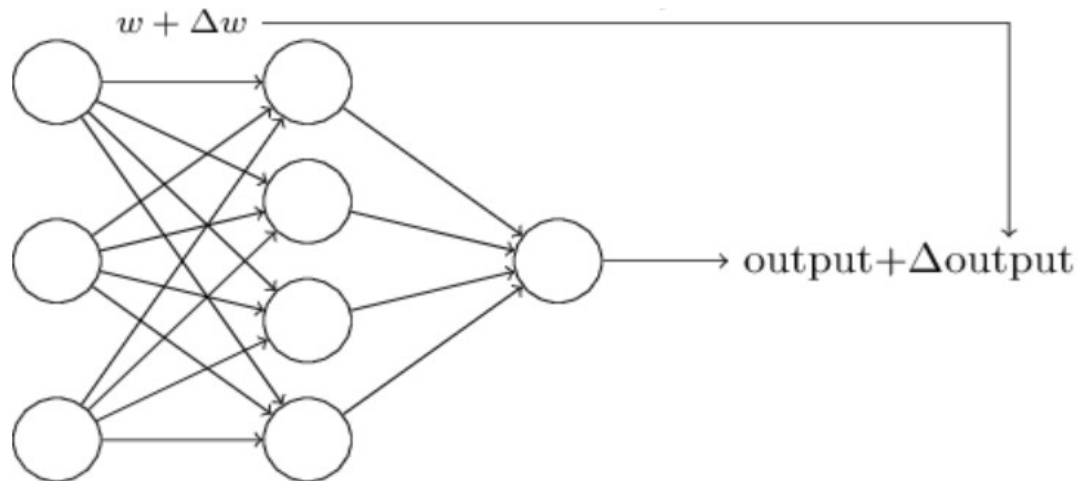


Рис. 1.10 Схема нейронної мережі з деяким зміщенням у вагах, з метою викликати зміщення на виході.

Якщо б правда полягала у тому, що невелика зміна ваги або зміщення викликає невелику зміну величини виробництва, тоді ми могли б користатися тим фактом що для зміни ваг та зміщень, з метою змусити нашу нейронну мережу вести себе більше приблизно до того, як нам би того хотілося. Уявимо собі, що мережа помилково розпізнала цифру "8", коли насамперед цифра являє собою "9". Можна трішки з'ясувати, як робиться невелика зміна ваг та зміщень, щоб точність нашої мережі з класифікації зображень класифікувала вхідну цифру як "9". Потім повторимо змінюючи ваги і зміщення ще раз і ще

раз, з метою отримати більш точні та якісніші результати. Так мережа буде навчатиметься. Проблемою є те, що це трішки не те, що відбувається, в той момент коли нейронна мережа містить у собі персептрони. Насамперед невеличка зміна ваги чи зміщення одного окремого персептрона в мережі як наслідок призводить, що вихід у цього персептрона може зовсім змінитися, наприклад від 0 до 1. Такий переворот спричиняє те що решта мережі повністю зміниться достатньо важким чином. Отже, незважаючи на те що наша цифра "9" у даний момент може бути класифікована правильно, поведінка нашої мережі для інших зображень, скоріш за все кардинально зміниться важко контрольованим способом. Саме це ускладнює представлення того, як плавно змінювати ваги та зміщення, з метою щоб мережа наближалась до бажаної поведінки. Очевидно, що повинен бути деякий спосіб подолати таку проблему. Але не одразу зрозуміло, яким чином можна отримати мережу персептронів для тренування. Для подолання цієї проблеми, ми введемо новий тип штучного нейрона який має назву сигмоїдний нейрон. Ці сигмоїдні нейрони мають схожість до персептронів, однак модифіковані таким чином, що невеличка зміна їхньої ваги і зміщення викликають невеличку зміну у їхньому виходу. Що є важливим фактом, котрий дозволить мережі сигмоїдних нейронів навчатися. Таким чином, опишемо структуру сигмоїдного нейрону. Я буду зображати сигмоподібний нейрон так само, як раніше зображав персептрон:

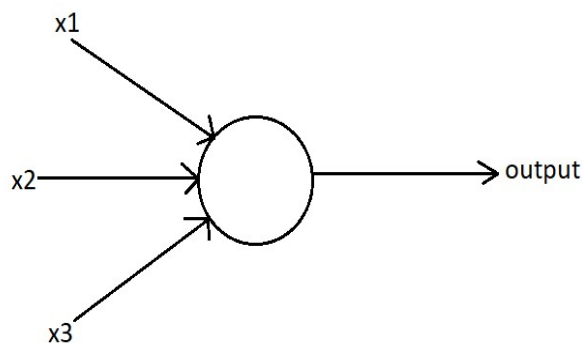


Рис. 1.11 Схема роботи сигмоїдного нейрону.

Сигмоїдний нейрон також як і персептрон має входи, $x_1, x_2, \dots$. Проте сигмоїдний нейрон, на відміну від персептрона на виході видає не тільки 0 чи

1, а може видавати значення з 0 до 1. Таким чином, число наприклад 0,452 ... може бути входом для сигмоїдного нейрона. Сигмоїдний нейрон також як і персептрон має ваги для входів, $w_1, w_2, \dots$, а також має загальний зміщення, b . Проте як було сказано вище може бути не тільки 0 або 1. Проте $\sigma(w \cdot x + b)$, де σ називають сигмоїдною функцією. Між іншим, σ іноді називають логістичною функцією, а цей новий клас нейронів називають логістичними нейронами. Корисно пам'ятати цю термінологію, оскільки цими термінами користуються багато людей, що працюють з нейронними мережами. Однак я буду дотримуватися сигмоїдної термінології. І математично це визначається як:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

Рис. 1.12 Математична формула сигмоїдної функції.

Якщо конкретизувати цю функцію, з урахуванням входів x_1, x_2 . Ваговими коефіцієнтами w_1 та w_2 , та зміщенням, то отримаємо наступну формулу:

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}.$$

Рис. 1.13. Математична формула сигмоїдної функції з врахуванням ваг, вхідних даних, та зміщення.

З початку складується враження що сигмоїдний нейрон дуже відрізняється від персептрону. Насамперед є достатньо багато спільності між персептроном і сигмоїдним нейроном, та алгебраїчний вираз сигмоїдної функції є деякою технічним нюансом. Для більш кращого розуміння подібності моделі персептрону та сигмоїдного нейрону, припустимо що, $z \equiv w \cdot x + b$ – є якимось великим додатнім числом. Звідси $e^{-z} \approx 0$ і тоді $\sigma(z) \approx 1$. Інакше кажучи, в той момент коли $z = w \cdot x + b$ є великим та додатнім числом, вихід з сигмоїдного нейрона буде дорівнювати приблизно 1, таке ж саме значення

могло би бути і для персептрона. Проте уявимо собі протилежну ситуацію, що $z = w \cdot x + b$ дуже від'ємне число. Звідси $e^{-z} \rightarrow \infty$, тоді $\sigma(z) \approx 0$. Таким чином, коли $z = w \cdot x + b$ сильно від'ємне число, сигмоїдний нейрон поводить себе дуже близько до персептрону. Але треба зазначити що, коли $w \cdot x + b$ має невеликі розміри з точки зору значення, може бути достатньо різниці від моделі персептрона.

Розглянемо алгебраїчну форму σ . Насамперед точна форма σ , не містить у собі для нас великого пріоритету, більш важливим для нас є форма побудови графіку функції. Ось як вона виглядатимуть:

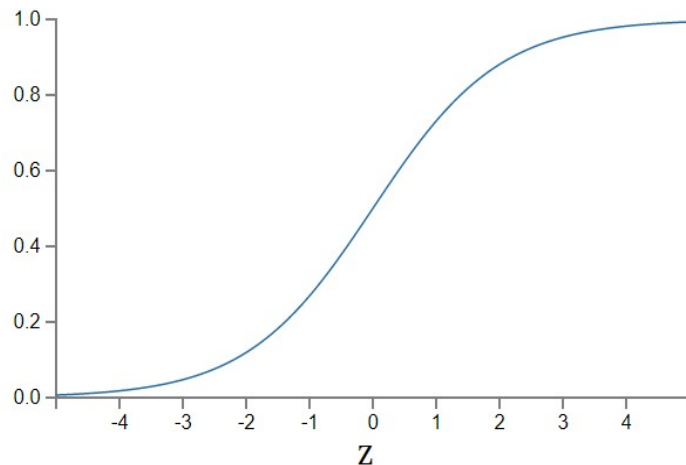


Рис. 1.14 Графік сигмоїдної функції.

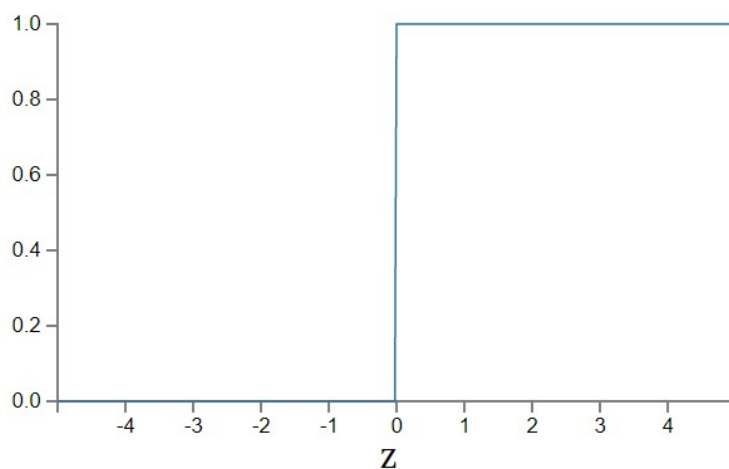


Рис. 1.15 Крокова функція(Згладжена версія)

Якщо б σ дійсно була б ступінчатою функцією тоді наш сигмоїдний нейрон був би персептроном, тому що вихід буде дорівнювати 0 або 1 залежачи від того що, буде $w \cdot x + b$ позитивним або негативним. Бо, коли $w \cdot x + b = 0$ вивід персептрону буде 0, коли вивід крокової функції буде 1. Таким чином нам треба було б змінити крокову функцію у цій одній точці. Якщо використати нашу функцію σ то ми отримуємо, як було зазначено раніше, згладжений персептрон. Насправді саме гладкість функції σ для нас має високий пріоритет, а не її детальною формою. Гладкість σ значить те, що деякі зміни Δw_j у вагах та Δb в зміщенні як наслідок приводить до змінах в значенні - Δ виходу на виході з нейрона. Наші обчислення говорить нам, про те що Δ вихід достатньо наближений до

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b,$$

Рис. 1.16 Зміщення виходу наближено дорівнює (output – це вихід)

Тобто тут сума має більше значення ніж всі ваги, w_j , а $\partial \text{output} / \partial w_j$ і $\partial \text{output} / \partial b$ це є часткові похідні з виходу стосовно w_j і b відповідно. Незважаючи на те що вираз на малюнку 1.16 складає враження що він є достатньо складний, з його частковими похідними, насамперед є цей вираз можна інтерпретувати як достатньо простий : Δoutput - це лінійна функція змін Δw_j і Δb у наших вагах та у зміщеннях. Така лінійність дає можливість достатньо легко вибирати маленькі зміни вагів та зміщень для досягнення якої завгодно бажаної маленької зміни у виході. Незважаючи на те що сигмоїдні нейрони мають теж добру модель поведінки, як і персептрони, їм занадто простіше зрозуміти, як саме зміна ваги і зміщення можуть вплинути на вихід.

Якщо форма σ має більший пріоритет, ніж її точна форма, то звідси стає логічне питання – з якою метою використовувати у рівнянні зображеному на малюнку 1.12 ту форму, яка використовується для σ ? Насправді, далі у моїй дипломній роботі я буду періодично звертати увагу на нейрони, де вихід являє собою $f(w \cdot x + b)$ для будь-якої іншої функції активації $f(\dots)$. Більшим

пріоритетом для нас є те, що змінюється, коли ми будемо використовувати якусь іншу функцію активації, пріоритетом є те що конкретні значення з наших часткових похідних з рівнянні зображеному на малюнку 1.16 змінюються. Звідси випливає що, коли ми обчислюємо наші часткові похідні пізніше, використовуючи σ буде спрощено алгебру, тільки тому, що експоненти мають гарні властивості для диференціювання. У будь-якому разі, σ як правило використовується для роботи з нейронними мережами.

Очевидно, що є деяка велика різниця між персептронами і сигмоїдними нейронами, і вона полягає в тому, що сигмоїдні нейрони виводять не тільки 0 або 1. Як вже було зазначено, вони на виході можуть мати довільне дійсне число між 0 і 1, отже значення по типу - 0,287... і 0,543..., можуть бути нашими вихідними результатами. Така властивість може бути корисною, як приклад, якщо ми хочемо користуватися вихідним значенням для того щоб представити середню інтенсивність пікселів на вході зображення в нейронну мережу. Але також це іноді може бути для нас і неприємністю. Уявимо що, для нас бажано, щоб вихід з мережі вказував або "цифра зображена на малюнку - це 9", або щоб "цифра зображена на малюнку не є 9". На перший погляд, найпростішим способом для нас було б якщо, результат був 0 або 1, як у персептроні. Проте на практиці можна створити конвенцію з метою вирішення такого питання, як приклад, якщо ми бажаємо інтерпретувати деякий вихід не менше 0,5 як знак "9", а будь-який вихід менше ніж 0,5 як знак того що це "не 9". Зазначу, що, коли ми використовуємо такого типу конвенцію, це не повинно викликати якихось труднощів.

1.3. Архітектура нейронних мереж.

У даному розділі ми будемо розглядати нейронну мережу, котра може робити достатньо непогані результати, розпізнаючи рукописні символи. Припустимо, що у нас є така мережа:

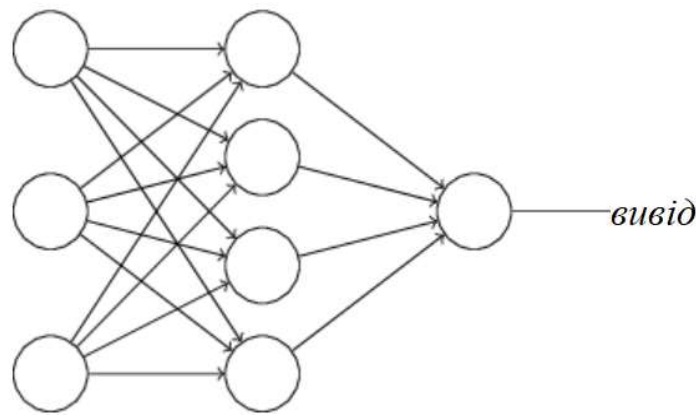


Рис. 1.17 Схема нейронної мережі по розпізнанню рукописного тексту

Як я зазначав раніше у попередніх розділах, лівий шар нейронів, котрий найлівіший, у такій мережі має назву вхідний шар, а нейрони всередині такого шару мають назву вхідних нейронів. Шар який міститься з крайнього правого боку має назву вихідний шар та містить він вихідні нейрони, або, як на малюнку 1.17 один вихідний нейрон. Середній шар називається прихованим, тому що нейрони в цьому шарі не являють собою входами, ні виходами. Термін "прихований", можна інтерпретувати як "не вхід чи вихід". Мережа яка зображена на малюнку 1.17 має лише один прихований шар, проте є деякі мережі які мають кілька прихованих шарів. Так наприклад, мережа яка зображена на малюнку 1.18 має одразу два прихованих шари. [16]

Схема вихідних та вхідних шарів у нейронній мережі часто не є складною. Так наприклад, уявимо що наша задача полягає в тому щоб визначити, зображено на малюнку цифра "9" чи ні. Звичайним чином для проектування мережі може бути кодування інтенсивності пікселів вхідного зображення для вхідних нейронів. Якщо вхідне зображення є зображенням сірого кольору, з масштабом 64 на 64, тоді звідси кількість вхідних нейронів була б $64 \times 64 = 4,096$ вхідних нейронів, інтенсивності яких будуть масштабуватися між 0 та 1. Вихідний шар буде складатися лише з одного нейрону, значення виходу буде менше, ніж 0,5 то це можна інтерпретувати як "зображення яке подається на вхід не є 9", а значення яке буде більше ніж 0,5, інтерпретується як зображення яке подається на вхід не є 9". Багатошарові мережі такі як ми використовуємо можуть ще називати як багатошаровими

персептронами але, навіть незважаючи на те що така мережа складається зі сигмоїдних нейронів, а не персептронів.

Незважаючи на те що дизайн вихідних та вхідних шарів нейронної мережі достатньо простий, архітектура прихованих шарів нейронної мережі може бути достатньо складною.

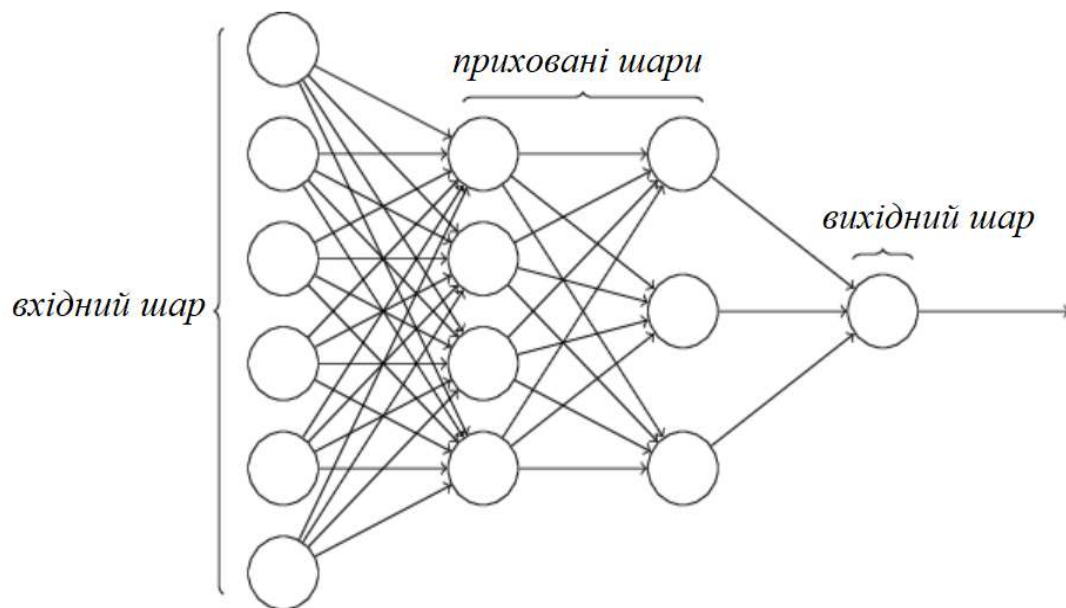


Рис. 1.18 Схема нейронної мережі з двома прихованими шарами.

Зазначу що достатньо не легко зробити якийсь підсумок для процесу проектування прихованих шарів при змозі вивести якісь правила. Однак, дослідники нейронних мереж спроектували багату кількість дизайнів архітектури саме для прихованих шарів, котрі допомагають користувачам нейронних мереж отримувати поведінку та модель, котру вони хочуть, від своєї мережі. Як приклад, така архітектура може використовуватися, з метою допомогти визначити, як оперувати кількістю прихованих шарів на час, необхідних для тренування мережі.

До цього часу ми оглядали нейронні мережі, в котрих вихід з шару використовується для входу до наступного. Такі мережі називаються - feedforward нейронні мережі. Це значить, що у нейронній мережі відсутні цикли, тобто інформація подається тільки вперед, та ніколи не буде подаватися

назад. Якщо б ми мали петлі, то це б закінчилися ситуаціями, коли б вхід до функції σ міг залежати від результату. Така ситуація була би важкою для розуміння, саме тому ми не будемо допускати таких циклів.

Проте маються й інші моделі штучних нейронних мереж, котрі містять у собі петлі зворотного зв'язку. Такі моделі мають назву - рекурентні нейронні мережи. Основна ідея таких архітектур полягає у розробці нейронів, котрі спрацьовують протягом деякого обмеженого тривалого часу, перш ніж вони стають у не збудженому стані. Таке збудження має здатність стимулювати інші нейрони, котрі можуть перейти у збуджений стан трішки пізніше, а також на якийсь обмежений термін. Це призводить до збудження ще більшої кількості нейронів, саме тому з потягом часу ми отримуємо каскад старту нейронів. Петлі не створюють проблем в таких архітектурах, тому що вихід нейрону має вплив тільки на його вхід у більш пізній час, а не одразу. Нейронні мережі які обладують властивістю рекурентності є менш впливовими, у порівнянні з мережами нашого типу, це тому, що алгоритми тренування для мереж з циклами є не настільки потужними. Проте циклічні мережі все ще містять цікаві властивості. Такі мережі являють собою прямою проекцією того як саме працює наш мозок. Та цілком можливо, що мережі які містять у собі петлі можуть вирішувати важливі задачі, котрі можна вирішувати лише через певні труднощі за допомогою мереж як мережа в цій дипломній роботі. Саме тому я зупинюся на більш широко використовуваних штучних мережах.

1.4. Проста мережа для класифікації рукописних цифр.

Тепер коли ми визначилися з нейронною мережею, я повернуся до розпізнавання нейронною мережею рукописного тексту. Ми можемо розділити задачу розпізнавання рукописних символів на дві підзадачі. По-перше, я хочу розглянути шлях розбиття зображення, яке містить багато символів, на послідовність окремих зображень одного символу. Так наприклад, ми можемо робити зображення цифр на малюнку 1.4.1.

524091

Рис. 1.18 Зображення рукописних чисел.

Наприклад нам хотілося б розділити це зображення на шість окремих символічних зображень. Що б на виході ми отримали ось так:

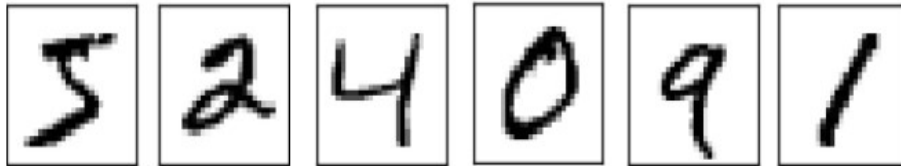


Рис. 1.19 Розбиття на окремі цифри.

Людині вирішити таку проблему сегментації достатньо легко, проте для комп'ютерної програми складно розділити таке зображення. Після сегментації зображення, нашій програмі необхідно розпізнати кожний окремий символ. Наприклад якщо, ми хочемо, щоб наша мережа була здатна визнати, що перша цифра на малюнку 1.20, являє собою 5.



Рис. 1.20 Рукописне зображення п'ятірки.

Зосередимося на розробці програми для вирішення другої задачі, тобто розпізнання окремих символів. Ми зробимо так тому що, як ми вже з'ясували, що проблему сегментації вирішується не досить складно, ця задача добре розв'язується, коли ми маємо більш менш вдалий спосіб для класифікації окремих символів. Є багато підходів для вирішення цієї задачі сегментації. Один із таких підходів полягає у випробуванні багатої кількості різних

способів розділення зображення, використовуючи індивідуальний розрядний класифікатор з метою оцінювання кожної пробної сегментації. Пробна сегментація може отримати високу оцінку, якщо окремий розрядний класифікатор достатньо точний у своїй класифікації для всіх сегментів, та низький бал, у випадку якщо класифікатор стикається з багатою кількістю проблем на одному чи на декількох сегментах. Головна ідея полягає в тому, що якщо у класифікатора де-небудь виникають певні проблеми, то, ймовірно, ще виникнуть проблеми, тому що сегментація обрана неправильно. Таку ідею та інші способи можна достатньо добре використовувати для вирішення проблеми сегментації. Тому ,зараз зосередимось на розробці нейронної мережі, яка здатна вирішувати більш цікаві та складні проблеми, а саме розпізнавання окремих рукописних символів. [15]

Для розпізнавання окремих символів в цій роботі ми будемо використовувати тришарову нейронну мережу , яка зображена на малюнку:

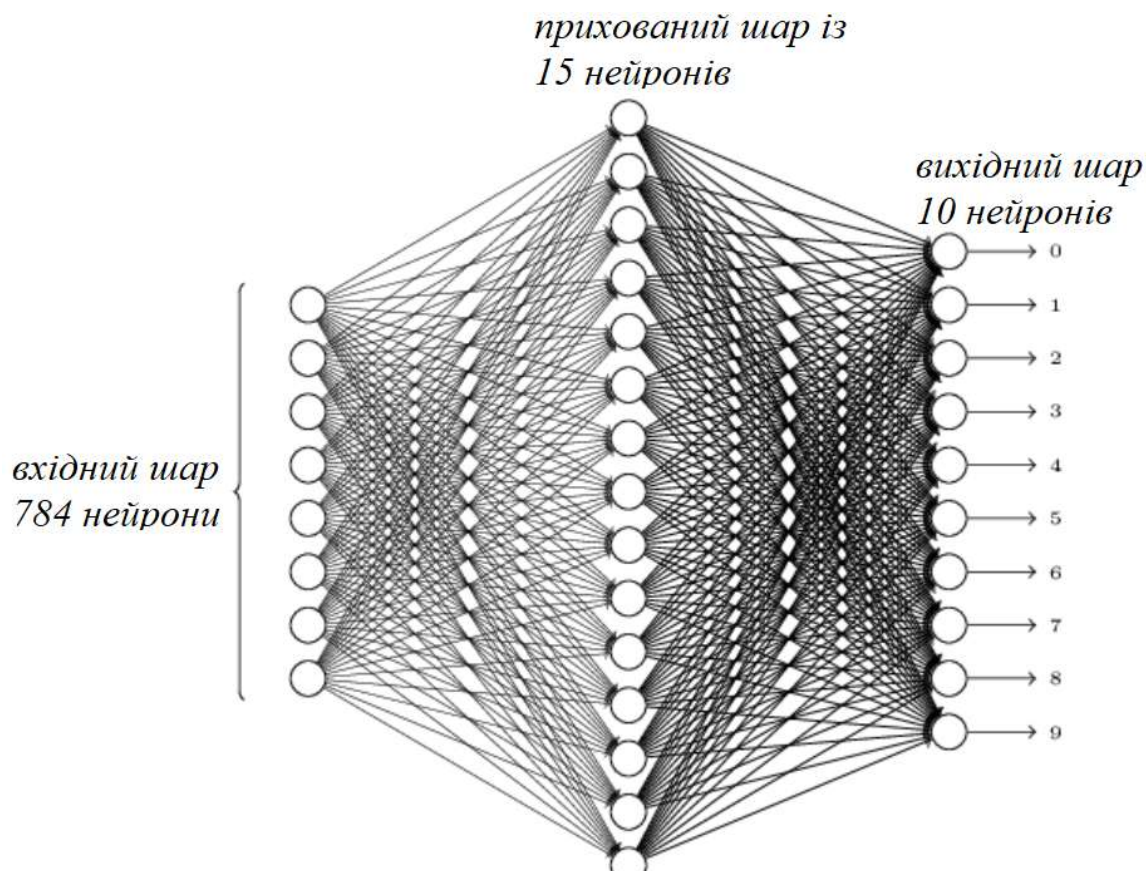


Рис. 1.21 Тришарова нейронна мережа для розпізнавання окремих цифр

Вхідний шар нейронної мережі містить нейрони, які будуть кодувати значення вхідних пікселів. Наші дані для тренування нашої мережі будуть складатися з багатої кількості зображень від 28 до 28 пікселів відсканованих рукописних цифр, саме тому вхідний шар містить $28 \times 28 = 784$ нейронів. Для спрощення я не відображав повністю більшість з 784 вхідних нейронів на малюнку 1.4.4. Пікселі які подаються до входу мають відтінки сірого, можна інтерпретувати значення 0.0 як біле, значення 1.0 як чорне, а між значеннями 1 та 0, це є поступово темніші відтінки сірого.

Другий шар штучної мережі - це прихований шар. Позначимо у цьому прихованому шарі кількість нейронів буде дорівнювати n , та ми будемо тестувати з різними значеннями для кількості n . У прикладі що зображений на малюнку проілюстровано невеликий прихований шар, що містить у собі 15 нейронів.

Вихідний шар мережі складається з 10 нейронів. У випадку коли перший нейрон збуджується, тобто його вихід ≈ 1 , то це позначатимуть те, що мережа вважає, що цифра являє собою 0. У випадку коли другий нейрон спрацьовує, то це позначатиме, мережа вважає, що цифра являє собою 1. І так далі . Пронумеруємо більш точніше вихідні нейрони від 0 до 9 та з'ясуємо, який саме нейрон має саме високе значення активації. Якщо такий нейрон, наприклад, має номер 6, тоді наша мережа зробить висновок, що вхідною цифрою була 6. І так продовжуємо далі для інших нейронів на виході.

В даній роботі використовуємо 10 вихідних нейронів. Так зроблено тому що, мета нашої штучної мережі - відповісти нам на запитання, яка саме цифра з діапазону - (0,1,2,..., 9) відповідає до вхідного зображення. Нібито природний спосіб так зробити, це користуватися лише чотирма вихідними нейронами, розглядаючи кожен окремий нейрон як вхід деякого двійкового значення, залежачи від того, чи є насправді вихід нейрона близьким до 0 чи до 1. Для того щоб закодувати відповідь достатньо користуватися чотирма нейронами, тому що $2^4 = 16$ - це може бути більше десяти можливих значень для вхідної цифри. Тоді постає логічне питання - чому мережа повинна

користатися 10 нейронами замість цього? Можна спробувати обидві різні конструкції нашої мережі, та виявляється, що для такої конкретної задачі мережа з десятима вихідними нейронами тренується розпізнавати символи значно точніше, у порівнянні з мережею з чотирма вихідними нейронами. Але перед нами постає такі питання як, чому користуючись десятима вихідних нейронів працює краще. Та чи існує певний спосіб, який би сказав нам відповідь би нам, чи має сенс використовувати кодування з десятима виходами замість чотирьох вихідного кодування?

Для зрозуміння, навіщо це робити, допомагає поміркувати про те, що саме робить наша нейронна мережа з перших принципів. Для початку розглянемо випадок, коли ми користуємося десятима вихідними нейронами. Ми зосередимось на першому вихідному нейроні, оскільки він намагається вирішити, чи дорівнює цифра 0. Це буде робитися шляхом обрахування доказів із прихованого шару нейронів. Розглянемо, що саме роблять ці нейрони, які знаходяться в прихованому шарі. Припустимо, для аргументації, що перший нейрон у прихованому шарі вирішує, чи являє собою зображення на зразок зображеного на малюнку:



Рис. 1.22 Фрагмент рукописної цифри.

Ми можемо це зробити за допомогою великої ваги вхідних пікселів, які мають перетин із зображенням, та лише трішки зважуючи інші входи. Аналогічно, припустимо, що для аргументу, другий нейрон, третій, також четвертий в прихованому шарі вирішують, чи є такі зображення чи ні:

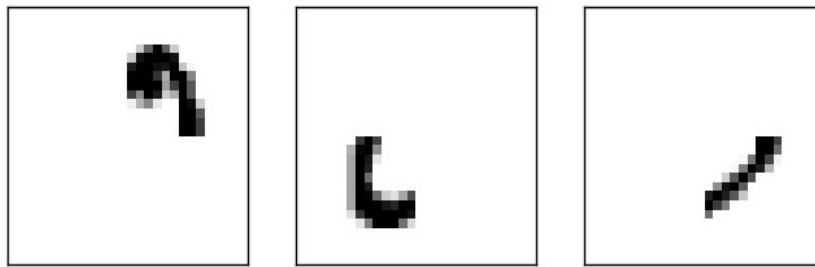


Рис. 1.23 Декілька фрагментів з однієї рукописної цифри.

Ці чотири зображення укупі складають зображення цифри - 0, таке зображення ми вже бачили в рядку цифр, на малюнку 1.4.2:

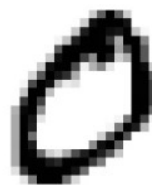


Рис. 1.24 Зображення рукописного нуля, який складається з тих фрагментів які ми розглянули вище.

Таким чином, якщо всі чотири з цих прихованих нейронів будуть збуджуватися, тоді можна зробити висновок, що цифра являє собою - 0. Звісно, це не єдиний спосіб доказу, яким ми можемо користатися з метою, що вирішення, що зображення було 0 - ми могли дійсно отримати 0 багатьма іншими способами. Проте важно впевнено сказати, що принаймні у цьому випадку ми б могли зробити такий висновок, що вхід був би 0. Якщо припустити таким чином, що функції нашої нейронної мережі, можна дати більш точне пояснення, чому дійсно краще мати десять виходів з мережі, а ніж чотири. Якщо у нас було би чотири виходи, то перший вивід з нейронну намагався б вирішити, що найбільше показниковий біт був у цифри. Та немає легкого способу для співвідношення такого показникового біта із простими фігурами, як зображено на малюнку. Важко представити, що є деяка гарна історична причина, що складові сегменти нашої цифри будуть певним чином тісно пов'язані з найбільш показниковим бітом з висновка. Ніщо нам не говорить про те, що нейронна мережа яка складається з трьох шарів повинна функціонувати так, як було описувано вище, приховані нейрони виявляють

прості компоненти інших компонентів. Цілком ймовірно що, розумний алгоритм навчання нейронної мережі знаходить певні ваги, які дозволяють нам користуватися лише чотирма вихідними нейронами. Проте як ми вже описали евристичний спосіб, мислення працює достатньо точно, і такий спосіб може заощадити багато часу при розробці гарних архітектур штучної нейронної мережі.

					ДП 6224. 03.000 ПЗ	Арк.
						28
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ ДО РОЗДІЛУ 1

В даному розділі було розглянуто основні теоретичні відомості, щодо будування, архітектури нейронних мереж. Ми прийшли до висновку що в основі нейрону, на котрих будується наша мережа, буде використана функція сигмоїди. Для згладжування результату. Також потрібно зазначити, що на виході даного розділу ми отримали нейрону мережу яка складається з трьох шарів. Перший шар містить в собі 784 нейрони. Так складається тому що містить в собі зображення 28 на 28 пікселів. Другий шар являє собою прихований шар, на основі нейронів з цього шарів буде збуджуватись нейрон з остатнього вихідного шару та видаватись результат.

					ДП 6224. 03.000 ПЗ	Арк.
						29
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2

НАВЧАННЯ І ТРЕНУВАННЯ НЕЙРОННОЇ МЕРЕЖІ

2.1 Набір даних MNIST

Таким чином, коли ми створили архітектуру нашої нейронної мережі, виникає питання як вона буде навчатися розпізнавати символи? Перше, чим ми скористаємося - це певний набір даних, з якого ми зробимо вибірку - так званий набір даних для тренування. Ми будемо користуватися набором даних MNIST, який містить декілька десятків тисяч відсканованих зображень рукописних символів у купі з їхньою правильною класифікацією. MNIST являє собою модифіковану підмножину з двох наборів даних, які були зібрані інститутом стандартів та технологій США NIST.

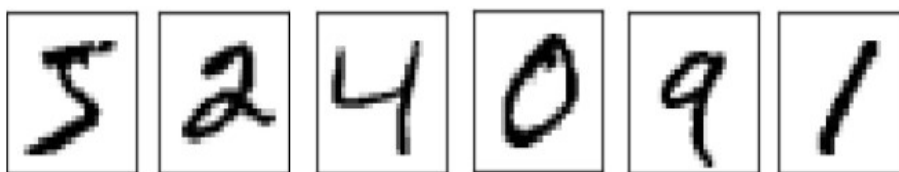


Рис. 2.1 Зображення відсканованих цифр з набору даних MNIST.

Дані з цього набору збережені у двох частинах. Перша частина складається з 60 000 зображень, які ми будемо використовувати для навчання нашої мережі. Такі зображення являють собою відскановані зразки рукописного тексту від 250 осіб, деяка частина з цих людей були співробітниками бюро перепису США, а інша частина - старшокласниками. Зображення мають відтінки сірого кольору та вони зображені розміром 28 на 28 пікселів. Інша частина з нашого дата сету MNIST - це 10 000 зображень, якими ми будемо користатися як тестовими даними. Ми будемо користуватися цими даними з тесту, з метою оцінення, того наскільки якісно наша мережа натренувалася розпізнавати символи. Для того щоб зробити це якісним тестуванням працездатності, дані для тестування були взяті з іншого набору, а вихідні дані для навчання з іншого. Це допоможе нам впевнитися в тому, що наша мережа здатна розпізнавати символи з почерку людини, яких вона не раніше не бачила під час навчання.

					ДП 6224. 03.000 ПЗ	Арк.
						30
Зм.	Арк.	№ докум.	Підпис	Дата		

Ми будемо користуватися позначанням x для того щоб позначати вхід для тренування. Для нас буде зручніше розглядати кожен з входів x як $28 \times 28 = 784$ тобто 784-мірний вектор. Кожний запис у цьому векторі дорівнює значенню сірого одного пікселя для вхідного зображення. Позначимо бажаний для нас вихід як $y = y(x)$, де y являє собою 10-мірний вектор. Представимо що, якщо на конкретному тренувальному зображенні, x , зображено 2, тоді $y(x) = (0,0,1,0,0,0,0,0,0,0)^T$ - бажаний для нас результат, тобто, вихід з мережі. Зазначимо, що T у виразі вище являється операцією транспонування, яка перетворює векторний рядок у вектор стовпчик. З яким нам більш зручніше працювати.

2.2 Функція витрат

Бажано б мати алгоритм, який дозволить нам знаходити зміщення та ваги, таким чином, щоб значення на виході з нейронної мережі було приблизно таким як $y(x)$ при всіх тренувальних входах x . Для того щоб якимось кількісно оцінити, наскільки якісно ми досягаємо такої мети, визначимо функцію витрат.

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Рис. 2.2 Функція витрат.

Де, w знатимуть сукупність усіх ваг з нашої мережі, b є всіма зміщеннями, n - загальна кількість входів для навчання, a - вектор виходу з мережі якщо ми ввели x , а сума більше ніж всі входи для тренування, x . Звісно, вихід з a залежатимуть від x , w та b , проте, щоб позначення було простіше, я не вказував прямо цю залежність. Позначення $\|v\|$ значиме просту функцію обчислення довжини вектора v . Квадратичну вартість витрат я позначу як - C ; іноді це називають середньою помилкою в квадраті. Якщо перевірити форму функції квадратичної вартості, то ми побачимо, що $C(w, b)$ є від'ємне значення, тому що кожний доданок при сумуванні є від'ємним. Крім цього, значення $C(w, b)$ стає невеликим, тобто $C(w, b) \approx 0$, в той момент, коли $y(x)$

буде приблизно дорівнювати виходу, а, для всіх тренувальних входів, x . Отже, наш алгоритм навчання зробив непогану роботу, при умові що він може знайти ваги та зміщення, таким чином щоб $C(w, b) \approx 0$. З іншої боку, це не так добре, коли значення $C(w, b)$ є великим, тому що це означає, що $y(x)$ дуже різниться від виходу, та близький для великої кількості входів. Отже, метою нашого алгоритму тренування нейронної мережі буде мінімізація витрат $C(w, b)$ як функції зміщень і ваг. Інакше кажучи, ми намагаємось знайти такі зміщення та ваги, які роблять вартість значно меншою. Ми будемо це робити за допомогою алгоритму, який був вже згадано - градієнтного спуску.

2.3 Градієнтний спуск

Головна проблема полягає в тому, що зображення які ми намагаємось розпізнати, та які у результаті правильно класифікуються, не являють собою гладкою функцією зміщень та ваг в мережі. Як правило, невеликі внесення змін у вагах та зміщеннях, як наслідок, не спричиняють певні зміни у кількості правильно класифікованих зображень. Така властивість ускладнює пошуку методів зміни зміщення і ваги з метою підвищення продуктивності. Якщо замість цього ми скористаємось функцією плавної вартості на зразок квадратичної вартості, тоді виявиться, як можна змінити зміщення та ваги, для того щоб отримати підвищення вартості. З цієї причини ми для початку зосередимось на мінімізації квадратичної вартості, а вже після цього зазначимо точність класифікації. Квадратична функція витрат яка зображена на малюнку 2.2 достатньо добре працює для вступу в основи тренування нейронних мереж.[16]

Нашою мета в тренуванні нейронної мережі є пошук ваг і зміщень, які будуть мінімізувати квадратичну функцію витрат $C(w, b)$. Це точно поставлена задача, проте вона має багато відволікаючих факторів, як вказано в даний момент - інтерпретація w і b як зміщень та ваг, σ функція, яка ховається на задньому плані, вибір архітектури мережі, MNIST тощо. Звідси висновок що, ми здатні оволодіти величезною кількістю, ігноруючи при цьому більшу частину даної структури та лише зосередивши увагу на факторі мінімізації.

Таким чином наразі ми відкинемо все про форму функції витрат, а також підключення до нейронних мереж тощо. Замість цього ми уявимо, що нам лише була дана функція багатьох змінних та ми намагаємося мінімізувати цю функцію. Тому я буду розробляти техніку яка називається градієнтний спуск, якою ми можемо користуватися для рішення таких задач мінімізації. Таким чином повернемося до конкретної функції, котру ми бажаємо мінімізувати для нейронних мереж.

Отже, припустимо, що ми намагаємося мінімізувати деяку функцію, $C(v)$. Це може бути будь-яка функція з дійсними значеннями для багатьох змінних, $v = v_1, v_2, \dots$. Одразу зазначу, що ми замінили позначення w і b на v , для уточнення, що це може бути будь-яка функція. Для того щоб, мінімізувати $C(v)$ ми можемо уявити функцію C як функцію лише двох змінних, які ми будемо позначати через v_1 і v_2 :

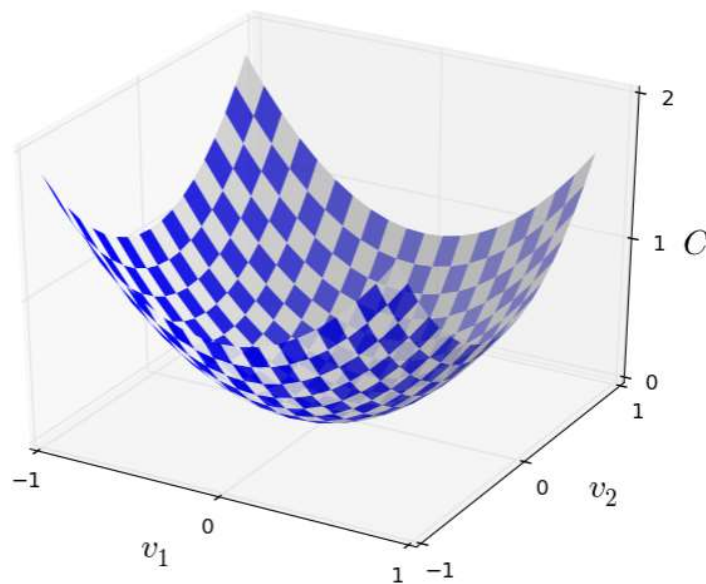


Рис. 2.3 Графік градієнтного спуску.

Нам хотілося б відшукати, де C досягає свого глобального мінімуму. Тепер, для функції, на малюнку вище, ми можемо розглянути графік та знайти мінімум. Загальна функція C може бути складною для багатьох змінних, та як правило не вдається лише на огляд, на графіку відшукати мінімум. Один із способів вирішити таку задачу полягає в тому щоб, використовувати обчислення, з метою спробувати відшукати мінімум аналітично. Для початку

ми можемо знайти похідні, а потім спробувати користатися ними, щоб відшукати місця, де C – є екстремум. Коли C є функцією лише однієї чи декількох змінних, тобто у вдалому випадку, тоді це, може спрацювати. Проте це буде сильним мінусом, коли у нас буде ще більше змінних. Для нейронних мереж за частую нам бажано набагато більше змінних, оскільки найбільші нейронні мережі містять у собі вартісні функції, котрі залежать від мільярдів ваг та зміщень достатньо складним чином. Використовуючи обчислення мінімізації, яке просто не буде працювати! [16]

Отже ми дійшли до висновку що, обчислення не працює. На щастя, є достатньо схожа аналогія, з якої впливає алгоритм, який працює достатньо гарно. Ми випадковим чином обираємо початкову точку для деякої уявної кулі, а вже потім ми будемо моделювати рух кулі, коли він спускається донизу долини. Можна було б зробити таке моделювання, лише шляхом обчислення похідних функції C - ці похідні повідомили б нам всю інформацію, яку нам необхідно знати про локальну "форму".

Отже, розробимо алгоритм мінімізації C . Обчислення кажуть нам про те, що C змінюється наступним чином:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

Рис. 2.4 Формула мінімізації.

Ми будемо намагатися знайти спосіб вибору Δv_1 та Δv_2 , так, щоб зробити ΔC від'ємним. Щоб з'ясувати, як зробити, такий вибір, нам допоможе в цьому Δv , який являє собою вектор змін v , $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, де T - операція транспонування, яка перетворює вектори рядків у вектори стовпці. Визначимо також градієнт C , який буде нашим вектором часткових похідних, $(\partial C / \partial v_1, \partial C / \partial v_2)^T$. Вектор градієнта позначимо як ∇C , таким чином:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T.$$

Рис. 2.5 Вектор градієнта

Наступним кроком перепишемо нашу зміну ΔC через Δv та градієнт ∇C . Скориставшись такими визначеннями, можемо переписати вираз на малюнку 2.4 для ΔC , наступним чином:

$$\Delta C \approx \nabla C \cdot \Delta v.$$

Рис. 2.6 Формула мінімізації.

Рівняння на малюнку 2.6 допоможе з'ясувати, чому ∇C ми називаємо градієнтним вектором: ∇C зв'язує зміни v зі змінами в C , це і робить те, що ми називаємо градієнтом. Але, що дійсно має пріоритет в нашому рівнянні, це те, що це рівняння дозволить нам побачити, яким чином слід вибрати Δv , щоб зробити ΔC від'ємним. Припустимо, що ми обираємо таким чином:

$$\Delta v = -\eta \nabla C,$$

Рис. 2.7 Формула вектора v .

де η - малий позитивний параметр ступінь навчання, який має назву ступінь навчання. Звідси випливає, що рівняння на малюнку 2.6 говорить нам про те, що $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$. Та оскільки $\|\nabla C\|^2 \geq 0$, це гарантує нам, що $\Delta C \leq 0$, таким чином C буде завжди зменшуватися, та не буде ніколи збільшуватись, при умові якщо ми замінимо v так як зображено на малюнку 2.6. Таким чином, ми отримали, те що й хотіли. Тому візьмемо рівняння 2.6 для алгоритму градієнтного спуску. Тому що ми будемо користуватися рівнянням 2.6 для того щоб обчислювати значення Δv , а вже потім позицію кулі v перемістимо на цю суму:

$$v \rightarrow v' = v - \eta \nabla C.$$

Рис. 2.8 Формула

Потім ми повторно скористаємося цим правилом оновлення, з метою зробити ще один крок. Якщо ми будемо продовжуватимемо користуватися цим правилом знову і знову, ми будемо продовжувати знижувати C до тих пір, поки ми не дійдемо до глобального мінімуму. Якщо підсумувати, то метод алгоритму градієнтного спуску – це є по-перше багаторазовим обчисленням градієнта ∇C , а вже потім переміщенням в зворотному напрямку. Відобразимо це графічно наступним чином:

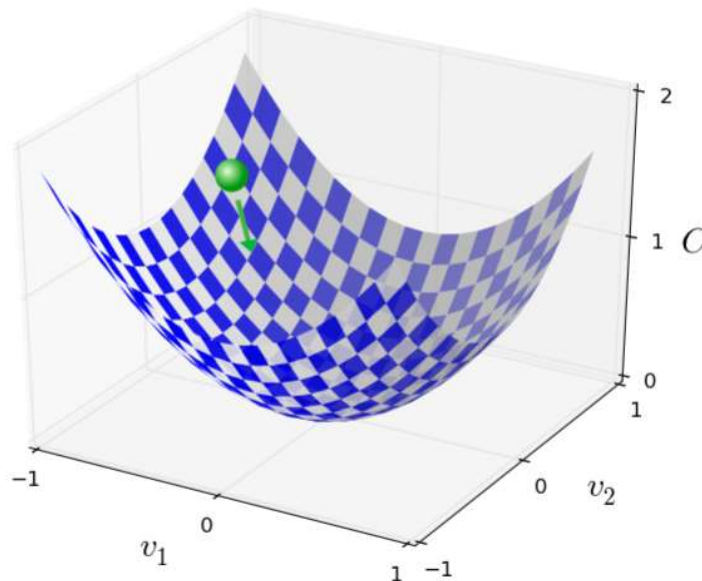


Рис. 2.9 Графік обчислення градієнтного спуску.

Для того щоб цей градієнтний спуск працював добре, нам необхідно обрати коефіцієнт навчання η , про який я казав вище, таким чином щоб цей коефіцієнт був достатньо малим, для того щоб рівняння на малюнку 2.6 було добре наближено. Якщо ми цього не зробимо, ми можемо отримати що $\Delta C > 0$, що не є гарним результатом для нас. В той момент, нам бажано, щоб коефіцієнт η був дуже малим, тому що це зробить зміни Δv дуже крихітним, та, як наслідок алгоритм градієнтного спуску буде працювати дуже повільно. Далі це буде продемонстровано. [15]

Пояснення градієнтного спуску, було через те коли C являє собою функцією двох змінних. Проте градієнтний спуск буде працювати так само якісно, і тоді коли C являє собою функцію багатьох змінних. Уявимо, що C - є функцією m змінних, $v_1, \dots, v_m$. Тоді зміна ΔC з C , виробляється через невелику зміну $\Delta v = (\Delta v_1, \dots, \Delta v_m)^T$, є рівнянням на малюнку 2.6 де градієнт ∇C – вектор.

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T.$$

Рис. 2.10 Формула для вектора градієнтного спуску

Як і для випадків з двома змінними, ми можемо обрати рівняння на малюнку 2.7 та ми будемо впевнені в тому що, що наш рівняння на малюнку 2.6 для ΔC буде від'ємним. Це дає нам можливість направляти градієнт до мінімуму, навіть для випадку якщо C являє собою функцією багатьох змінних, шляхом багаторазового застосування правила оновлення. Яким чином ми можемо використати алгоритм градієнтного спуску для тренування нашої нейронної мережі? Рішення полягає у тому щоб використовувати градієнтний спуск для пошуку зміщення b_l та ваг w_k , які мінімізують витрати у рівнянні на малюнку 2.2. Для того щоб відслідкувати, яким чином це працює, переформулюємо правило оновлення градієнта спуску, з зміщеннями та вагами замінюючи змінні v_j . Інакше кажучи, наше положення має такі компоненти як w_k та b_l , а вектор нашого градієнта ∇C як наслідок містить відповідні компоненти $\partial C / \partial w_k$ та $\partial C / \partial b_l$. Переформулювавши правило оновлення градієнта спираючись на компоненти, ми отримуємо наступну формулу:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Рис. 2.10 Правило оновлення градієнта по відношенню до компонентів

					ДП 6224. 03.000 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		37

2.4 Стохастичний градієнтний спуск

Тема яка має назву - стохастичний градієнтний спуск, ми можемо використати для прискорення тренування мережі. Основна думка полягає у тому, що ми будемо робити оцінку градієнта ∇C через обчислення ∇C_x для певної маленької вибірки входів для тренування які ми будемо обирати випадковим чином. Через те, що ми усереднимо цей невеликий зразок як наслідок, виявиться, що можна буде швидко отримати достатньо точку оцінку справжнього градієнта ∇C , і саме це буде сприяти прискоренню градієнтного спуску, та таким чином, тренуванню. Для того щоб реалізувати усі ці плани, стохастичний градієнтний спуск буде працювати таким чином що він буде обирати невелику кількість m випадково вибраних входів для навчання. Помітимо ці випадково обранні навчальні входи $X_1, X_2, \dots, X_m$ та назовемо це міні-партією. При умові, якщо розмір вибірки m буде достатньо великим, ми будемо очікувати, що середнє значення ∇C_{X_j} буде приблизно дорівнювати середньому значенню всіх ∇C_x , тобто

$$\frac{\sum_{j=1}^m \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

Рис. 2.11 Формула стохастичного градієнтного спуску.

Друга сума більше ніж весь набір даних для навчання. Якщо поміняти сторони, та в нас вийде наступне:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{X_j},$$

Рис. 2.12 Формула стохастичного градієнтного спуску

Таким чином, рівняння на малюнку 2.12, підтверджує, той факт, що ми здатні оцінити загальний градієнт, обрахувавши градієнти для випадково обраної міні-партії. Щоб пов'язати це правило з тренуванням нейронних мереж, припустимо що, b_l та w_k , позначають зміщення та ваги в нашій нейронній мережі. Потім стохастичний градієнтний спуск буде працювати таким чином,

що він буде обирати випадковим чином маленьку партію входів для навчання та тренуватися на них,

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

Рис. 2.13 Використання стохастичного градієнтного спуску.

Де суми всіх прикладів для навчання X_j з нашої міні-партії. Наступним кроком ми обираємо випадковим чином ще одну міні-партію та навчаємося на ній. І так будемо робити, доки ми в нас не закінчаться матеріали для тренування мережі, які нас інформують про закінчення епохи навчання. Та з цього моменту ми починаємо заново нову епоху навчання. Також, треба, одразу зазначити, що конвенції різняться відносно масштабування функції витрат та оновлень у міні-пакетах відповідно до ваг та зміщень. У рівнянні на малюнку 2.2 ми масштабували функцію витрат через коефіцієнт $1/n$. Іноді у деяких випадках нехтують $1/n$, з метою підсумування витрат на окремих приклади тренування замість усереднення. Це буде корисно, коли не відома загальна кількість навчальних прикладів. Таке може трапитися, коли, наприклад, генерується більше даних для тренування у режимі реального часу. Також, схожим чином, у правилі оновлення міні-партій через формули на малюнку 2.13 іноді опускають $1/n$ із передньої частини.

Якщо провести аналогію, то можна вважати, що стохастичний градієнтний спуск є схожим до політичного опитування: гараздо легше взяти невелику міні-партію, ніж скористатися градієнтним спуском до повної партії, а також якщо наприклад провести опитування легше, ніж провести повні вибори. Представимо, що ми маємо набір розміром $n = 60\,000$ саме для тренування, як у MNIST, та обиремо міні-пакетний розмір (наприклад) $m = 10$, це означитимуть, що ми отримали коефіцієнт прискорення 6000 при оцінювання градієнта! Звісно, оцінка не буде стовідсотково точною, оскільки відбудуться статистичні коливання, проте вона і не повинна обов'язково бути

ідеальною: все, що для нас дійсно має високий пріоритет, то знаходиться в загальному напрямку, що допоможе нам знизити C , а це означає, що нам не потрібне точне обчислення градієнта. На практиці стохастичний градієнтний спуск - це потужний та поширений спосіб тренування нейронних мереж, та це є базисом для більшості методик тренування.

2.5. Алгоритм зворотного розповсюдження

Якщо ми скористаємося рівнянням алгоритму зворотного розповсюдження, то це наддасть нам можливість обчислити градієнт функції витрат. Нижче покрокове, представлений цей алгоритм.

Алгоритм прямого розповсюдження:

- 1.) *вхід x : буде встановлює активацію a^l для вхідного шару.*
- 2.) *Інформація яку ми будемо отримувати на виході: Для кожного $l = 2, 3, \dots, L$, обчислювати $z^l = w^l a^{l-1} + b^l$ та $a^l = \sigma(z^l)$.*
- 3.) *Похибка на виході δ^L : Обчислення вектору $\delta^L = \nabla_a C \odot \sigma'(z^L)$.*
- 4.) *Похибка зворотного розповсюдження: для кожного $l = L - 1, L - 2, \dots, 2$ обчислювати $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.*
- 5.) *Вихід: Градієнт функції витрат буде задаватися як*

$$\partial C / \partial w_{jk}^l = a_k^{l-1} \delta_j^l \text{ та } \partial C / \partial b_j^l = \delta_j^l.$$

Детально розглядаючи алгоритм, не важко зрозуміти чому саме цей алгоритм має назву зворотного розповсюдження. Обчислення вектору помилок δ^l , відбувається з початку, з кінцевого шару. На перший погляд, це здається незручним. Незручним те що проходимо нашу мережу навпаки, тобто починаючи з останнього шару. Проте, якщо ми поміркуємо з точки зору доказу зворотного розповсюдження. То як наслідок можна зробити висновок, що рух назад - це наслідок того, що вартість є функція вартості являється виходом із мережі. Для того щоб отримати уявлення про те, як варіюється вартість якщо порівнювати її з попередніми зміщеннями та вагами. То для цього нам необхідно неодноразово застосовувати правило ланцюга. Ми будемо працювати навпаки через шари, з метою отримання зручні вирази.

Як було описано вище, алгоритм зворотного розповсюдження буде обчислювати градієнт функції витрат для одного тренувального прикладу, $C = C_x$. На практиці достатньо розповсюджено комбінувати зворотні розмноження з таким алгоритмом тренування, як стохастичний градієнтний спуск. У такому алгоритмі ми обраховуємо градієнт для багатьох прикладів тренування мережі. Зокрема, якщо розглянути міні-вибірку, яка складається з m прикладів тренування. То алгоритм який описаний нижче буде застосовувати етап тренування градієнтного спуску на основі цієї міні-вибірки.

Алгоритм :

1.) Введення набору прикладів для навчання.

2.) Для кожного навчального прикладу x : Встановлюємо відповідний вхід активації $a^{x,l}$, та виконаймо наступні кроки

2.1) Інформація на виході: Для кожного $l = 2, 3, \dots, L$ обчислюємо $z^{x,l} = w^l a^{x,l-1} + b^l$ і $a^{x,l} = \sigma(z^{x,l})$.

2.2) Вихідна похибка $\delta^{x,L}$: Обчислюємо вектор $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.

2.3) Зворотне розповсюдження помилки: Для кожного $l = L - 1, L - 2, \dots, 2$ обчислити $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

Звісно що, для того щоб реалізувати стохастичний градієнтний спуск на практиці нам необхідна буде зовнішня петля, яка буде генерувати міні-партії навчальних прикладів . А також зовнішня петля яка буде покроково проходити через кілька епох тренувань.

ВИСНОВКИ ДО РОЗДІЛУ 2

У нейронних мережах S , являє собою функцію багатьох змінних – усіх зміщень та ваг - саме тому визначатиме поверхню в дуже великому просторі. Іноді виникає думка про те що ми повинні вміти візуалізувати всі додаткові виміри. І тоді як наслідок виникає хвилювання про те що ми не можемо думати в чотирьох вимірах. Навіть більшість професійних фахівців не здатні наочно відобразити чотири виміри. Проте вони використовують, деякі інші способи розробки того, що буде відбуватися. Цим самим ми, скористувалися вище. Тобто ми скористувалися алгебраїчним, а не візуальним представленням ΔS , з метою щоб розібратися, як діяти, щоб зменшити S . Такі засоби можуть і не мати такої простоти, якою ми звикли користуватися, коли наприклад ми візуалізували три виміри, проте якщо ми створимо бібліотеку таких засобів, ви зможемо достатньо гарно працювати з великими розмірами.

					ДП 6224. 03.000 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		42

РОЗДІЛ 3

РЕАЛІЗАЦІЯ НЕЙРОННОЇ МЕРЕЖІ

3.1 Реалізація класу нейронної мережі

Отже, почнемо реалізовувати програму, яка тренує нашу нейронну мережу для розпізнавання рукописних цифр, для цього ми скористаємося стохастичним градієнтним спуском та великим набором даних для тренування. Реалізація буде реалізована на мові Python. Ми залишимо зображення для тренування такими, якими вони і були на початку, але поділимо наш набір для тренування, який складається з 60 000 зображень, на дві частини. Перший набір буде складатися з 50 000 зображень, якими ми будемо користатися для навчання нашої нейронної мережі, та другий набір для перевірки 10 000 зображень. Ми не будемо користуватися валідаційними даними в цьому розділі, однак пізніше ми з'ясуємо корисним для знаходження способів встановлення певних параметрів нашої нейронної мережі, тобто таких параметрів як ступінь навчання тощо. Які не є обиралися безпосередньо нашим алгоритмом тренування мережі. Однак дані для перевірки не являють собою частину оригінальної специфікації з нашого набору даних для навчання, багато розробників користуються набором даних таким чином, та використання цими даними перевірки є поширеними в нейронних мережах. Коли в даній роботі зустрічається термін "навчальні дані", то треба зазначити, що мається на увазі наш набір даних який складається з 50 000 зображень, а не оригінальний набір даних із 60 000 зображень.

Крім даних із нашого дата сету нам також необхідна бібліотека мови програмування Python яка має назву Numpy, для того що б ми могли користуватися вбудованими функціями лінійної алгебри. [1]

Для початку я обґрунтую основні особливості програмного коду нейронних мереж. Головна частина нашого проекту - це клас нашої майбутньої мережі, яким ми скористаємося для представлення нейронної мережі. Нижче приведений лістинг коду, яким ми будемо користуватися для ініціалізації об'єкта мережі:

					ДП 6224. 03.000 ПЗ	Арк.
						43
Зм.	Арк.	№ докум.	Підпис	Дата		

```

class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

```

Рис. 3.1 Реалізація класу нейронної мережі.

У даному коді на малюнку 3.1 розмірність списку визначається кількістю нейронів у відповідних шарах. З цього класу виникає, наприклад, якщо нам потрібно створити об'єкт мережі з 3 нейронами в першому шарі, 5 нейронами в другому шарі та з 1 нейроном у вихідному шарі, то ми зробимо наступним чином:

```

net = Network([3, 5, 1])

```

Рис. 3.2 Створення екземпляру класу нейронної мережі.

Усі ваги та зміщення в об'єкті мережі будуть ініціалізуватися випадковим чином, тобто ми будемо користатися функцією із бібліотеки Numpy - `numpy.random.randn` для генерації Гауссовських розподілів зі середнім значенням 0 і стандартним відхиленням у 1. Така випадкова ініціалізація дає місце нашому алгоритму стохастичного градієнтного спуску, місце, з якого ми будемо починати. Треба зазначити, що код який буде ініціалізувати нашу мережу передбачає, що вхідний шар нейронів є першим шаром та не він не дає можливості встановлювати певні зміщення для цих нейронів з цього шару, тому що зміщеннями користуються тільки тоді коли обчислюються виходи з пізніших шарів.

Одразу треба зазначити, що ваги та зміщення зберігаються як списки матриць Numpy. Так, з нашого коду якийсь `net.weights[1]` являє собою матрицю Numpy, яка зберігає ваги які з'єднують між собою другий та третій шари нашої нейронної мережі. Треба одразу відзначити, що 1 в даному випадку не значить зв'язок між першим та другим, а саме 2 та 3, оскільки

індекси в списках у мові програмування Python починаються з 0, а не з одиниці. Оскільки оператор `net.weights[1]` є достатньо об'ємним з точки зору символів, то для спрощення ми позначимо цю матрицю як w . Тобто матриця буде являти собою w , де w_{jk} – являє собою вагу для зв'язку між k -м нейроном з другого шару та j -м нейроном з третього шару. Така індексація як k та j на перший погляд здається незручною. Тобто звичайному розробнику здається на перший погляд, що було б значно краще міняти індекси j та k . Великою плюсом у використанні такого порядку є те, що це значиме. Що активаційний вектор для третього шару нейронів буде:

$$a' = \sigma(wa + b).$$

Рис. 3.3. Рівняння активацій третього шару нейронів

Рівняння на малюку 3.3 не є дуже об'ємним. Тому більш конкретної деталізації, розберемо його по частинах. Де a – являє собою вектор активації другого шару нейронів. Для того щоб отримати a' , ми помножимо нашу a на матрицю w , яка являється матрицею ваг. Та складемо з вектором зсуву b . А вже потім ми скористаємося функцією σ до кожного входу у нашому векторі $wa + b$. Така властивість має назву – векторизація функції σ . Не є складним перевірити, те що, рівняння на малюнку 3.3 дає той самий результат, як і рівняння на малюнку 1.2.4, яке використовують для обчислення виходу сигмоподібного нейрона.

Враховуючи усі ці властивості, не складно написати програму, яка буде обчислювати вихід з екземпляру класу нейронної мережі. Отже, ми почнемо с того що визначимо сигмоїдну функцію:

```
def sigmoid(z):  
  
    return 1.0/(1.0+np.exp(-z))
```

Рис. 3.4. Визначення сигмоїдної функції на мові Python.

Треба одразу зазначити, що коли вхідний параметр z являється вектором чи масивом Numpy. Тоді Numpy буде автоматично скористається функцією сигмоїдних елементів. Однак він це зробить у векторній формі. [2]

Після цього ми додаємо метод який буде називатися `feedforward`, до нашого класу нейронної мережі, до котрої ми задаємо вхід a для нашої мережі, та вона буде повертати відповідний вихід. Також, треба, треба одразу зазначити, що вхід a являє собою $(n, 1)$ Numpy масивом, в той же момент a не являє собою $(n,)$ вектор. Де n – являє собою кількістю входів у нашу мережу. Якщо спробувати скористатися вектором $(n,)$ як входом, тоді, ми отримаємо не дуже втішаючи результати. Однак, якщо ми будемо користуватися $(n,)$ вектором, то це виявиться більш природним вибором. Але якщо ми будемо користуватися $(n, 1)$ масивом, то це буде значно зручніше для того щоб редагувати код, а також для представлення декількох входів одночасно. Такі властивості надають зручнішого користування.

Таким чином формула на малюнку 3.3 для кожного шару буде матиме наступну реалізацію:

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

Рис. 3.5. Визначення рівняння активацій функції на мові Python.

3.1.1 Тренування нейронної мережі.

Звісно що, головна яка перед нами постає, це зробити так, щоб об'єкт нашої мережі міг навчатися. Тому до нашого класу ми додаємо метод під назвою `SGD`, у якому ми реалізуємо стохастичний градієнтний спуск. Нижче наведена його реалізація. [3]

```

def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    test_data = list(test_data)
    n_test = len(test_data)
    training_data = list(training_data)
    n = len(training_data)
    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)

    print("Epoch {0}: {1} / {2}".format(
        j, self.evaluate(test_data), n_test))

```

Рис. 3.6. Реалізація стохастичного градієнтного спуску на мові Python.

В даному коді `training_data` являє собою список кортежів (x, y) , які являють собою входи для тренування, а також відповідні бажані результати. Змінні які в коді ми позначили через `epochs` та `mini_batch_size` – це як раз те що нам і треба, тобто, це кількість епох, для яких мережа навчається, а також це розмір міні-пакетів, якими ми будемо користуватися для вибірки. Параметр `eta` являється ступінь навчання- η . Якщо до цього методу надати необов'язковий аргумент `test_data`, то наша програма буде робити оцінювання для нашої мережі після кожної епохи навчання та інформувати нас про її частковий прогрес. Це інформація буде корисною для нас, для того щоб ми могли у реальному часі бачити прогрес у навчанні, проте це впливово сповільнює роботу мережі.

Код працює наступним чином. На етапі кожної епохи він починає переміщувати дані для тренування випадковим чином, а вже наступним кроком розбиває ці дані на міні-партії відповідного розміру. Такий спосіб являється достатньо простим способом для вибіркового відбору даних з даних для тренування. Наступним кроком для кожної міні-партії ми будемо застосовувати один крок градієнтного спуску. Все це ми будемо робити за рахунок того що ми зробимо виклик методу `self.update_mini_batch(mini_batch, eta)`, саме цей код буде оновлювати зміщення та ваги нашої мережі в той момент коли буде виконуватися одна ітерація градієнтного спуску. Для цього

ми будемо користуватися лише даними для навчання з параметру `mini_batch`.
Нижче наведена реалізація методу `update_mini_batch`:

У методі на малюнку 3.7 є важливою строчка коду. Оскільки тут викликається алгоритм зворотного розповсюдження, що здатне прискорювати обрахунки градієнта функції витрат. Таким чином метод `update_mini_batch` буде працювати достатньо просто. Він буде обраховувати, ці градієнти для кожного тренувального прикладу в `mini_batch`, а вже наступним кроком буде оновлюватися `self.biases` і `self.weights` відповідним чином.

```

48
49 def update_mini_batch(self, mini_batch, eta):
50
51     nabla_b = [np.zeros(b.shape) for b in self.biases]
52     nabla_w = [np.zeros(w.shape) for w in self.weights]
53     for x, y in mini_batch:
54         delta_nabla_b, delta_nabla_w = self.backprop(x, y)
55         nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
56         nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
57     self.weights = [w-(eta/len(mini_batch))*nw
58                     for w, nw in zip(self.weights, nabla_w)]
59     self.biases = [b-(eta/len(mini_batch))*nb
60                   for b, nb in zip(self.biases, nabla_b)]
61
62     delta_nabla_b, delta_nabla_w = self.backprop(x, y)

```

Рис. 3.7. Реалізація методу `update_mini_batch`.

3.1.2 Реалізація алгоритму зворотного розповсюдження.

Отже коли ми докладно описали як працюватиме зворотне розповсюдження в даній роботі. То ми можемо реалізувати програмний код, який ми розглядали вище для здійснення зворотного розповсюдження. Повертаючись до моменту, де код який містить у собі такі методи як `backprop` та `update_mini_batch` класу `Network`. Код який є складовою методів являє собою прямий переклад алгоритму, який був описаний вище. Треба одразу зазначити що, метод `update_mini_batch` буде оновлювати ваги та зміщення

мережі, через обчислення градієнту для поточної міні-партії прикладів для тренування мережі.

Метод `backprop` повністю реалізовує алгоритм який був описаний вище. Але треба зазначити що ми будемо трішки по іншому індексувати шари. Таку зміну у коді ми зробили спеціально для того щоб використовувати функції мови Python, а якщо більше точніше то ми будемо використовувати від'ємні індекси списку для того щоб підрахувати від кінця списку. Наприклад, якщо ми візьмемо `[-2]` то це значиме що ми візьмемо другий елемент з кінця з нашого списку. Реалізація методу зворотного розповсюдження знаходиться нижче на малюнку 3.3.1.

Треба одразу зауважити, що змінна `l` в циклі буде використовуватися трохи по-іншому, ніж то як я позначав раніше у даній роботі. Там, де `l = 1` буде позначати останній шар нашої нейронної мережі. А `l = 2` – другий останній шар. Ми будемо використовувати таке перенумерування для того, щоб скористатися властивістю мови програмування Python, яка містить можливість користуватися від'ємними індексами у списках. Саме у цих списках міститься необхідні дані для обробці, та подальшої роботи з об'єктом нашої нейронної мережі.

Як результат метод буде повертати кортеж `(nabla_b, \nabla_w)`, що являє собою градієнт функції витрат `C_x`. `"nabla_w"` та `"nabla_b"` - це списки які містять відношення шар до шару. Тобто це списки типу масиву, які схожі на `self.weights` та `self.biases`.

```

def backprop(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    activation = x
    activations = [x]
    zs = []
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)

```

Рис. 3.8. Реалізація алгоритму зворотного поширення.

Треба також, додатково зазначити методи, які ми добавили до нашого класу Network. А саме ми додали метод evaluate, який буде демонструвати прогрес у навчанні.

```

def evaluate(self, test_data):
    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)

```

Рис. 3.9. Реалізація методу evaluate.

Метод на малюнку 3.9 повертає кількість прецедентів вибірки для тестування, у яких наша нейронна мережа буде видавати точний результат. Вибірка яка обирається для тестування складається з пар (x, y) . У такій парі x являє собою вектор розмірністю 784. Тобто 784 вхідних пікселя які відповідають кількості пікселів у вхідному зображенні, яке містить рукописний символ. А y являє собою числове значення цифри, зображеної на вхідному

малюнку. Результатом на виході зі нейронної мережі буде індекс нейрона у вихідному шарі. Що буде мати найбільше значення функції активації. Метод `evaluate` який ми реалізували, буде викликатися в методі стохастичного градієнтного спуску, в коді він має назву `SGD`, після того, як буде завершуватися чергова епоха навчання.

Окрім цього, нам необхідно додати метод, який буде обраховувати вектор часткових похідних. Даний метод буде визиватися зі методу під назвою `backprop`.

```
def cost_derivative(self, output_activations, y):
    return (output_activations-y)
```

Рис. 3.10. Реалізація методу `cost_derivative`.

3.2 Реалізація файлу з функціями для підключення набору даних для тренування.

Для тренування нашої нейронної мережі ми скористуємося архівом <http://deeplearning.net/data/mnist/mnist.pkl.gz> зі веб-сторінки сайту лабораторії машинного навчання. Який сформували на базі великого набору даних, який складається зі 70 000 зображень рукописних символів. Та який ми поділимо на три складові:

Перший це `training_data` – це набір який складається з 50 000 зображень, який буде використовуватися спеціально для тренування мережі.

Другий, `validation_data` – це набір який складається з 10 000 зображень, який буде використовуватися для поточної оцінки роботи алгоритму тренування, а також підбору параметрів для тренування;

Та останній це – набір який складається з 10 000 зображень, та саме цей набір буде використано для перевірки роботи мережі. Кожний такий набір містить у собі два списки. Перший - це список зображення, в градаціях сірого. Та другий містить у собі відповідний список цифр в діапазоні від 0 до 9. Вхідне зображення буде представлено у форматі одновимірної `numpy`-масиви довжиною 784. Оскільки зображення являється розмірністю 28×28 . Тобто в

нас буде 784 значень від 0 до 1, де 0 являє собою значення чорного кольору пікселя, а 1 білого.

Також, одразу для папки свого проекту, треба додати папку яка містить при собі архів з набором даних MNIST.

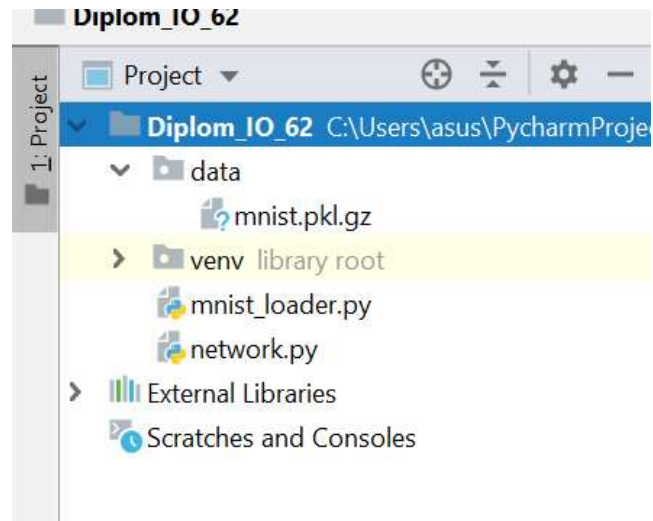


Рис. 3.11. Структура проекту.

Функції, які будуть працювати з базою даних MNIST, для зручності ми винесемо їх в окремий файл. Ми створимо новий файл який буде мати назву `mnist_loader` та збережемо його в папці проекту.

```
import gzip
import pickle
import numpy as np

def load_data():
    f = gzip.open('data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f,
                                                             encoding='latin1')
    f.close()
    return (training_data, validation_data, test_data)
```

Рис. 3.12. Структура проекту.

В коді на малюнку 3.12 було створено метод який буде відкривати стислий файл у двійковому форматі, та грузити дані для мережі, які саме будуть ділитися на три типи які ми розглянули раніше.

Для того щоб скористуватися нашим набором даних, нам необхідно в нашій програмі відредагувати формати зі змінних які буду повертатися функцією `load_data` а саме `training_data`, `validation_data` та `test_data`. Таке корегування робиться в методі `load_data_wrapper`. Нижче наведений лістинг коду цієї функції:

```
def load_data_wrapper():

    tr_d, va_d, te_d = load_data()

    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)

    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])

    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])

    return (training_data, validation_data, test_data)
```

Рис. 3.13. Реалізація методу `load_data_wrapper`.

Функція яка реалізована на малюнку 3.13 буде перетворювати `training_data` в список, який буде містити у собі 50 000 пар (x, y) . Ці містять у собі x , який є 784-мірним `numpy`-масивом, як раз це і є той масив про який було сказано. Оскільки він містить вхідне зображення. А також y - це 10-мірний `numpy`-масив, який являється вектором, та який має координату з порядковим номером ,який буде дорівнювати цифрі на зображенні, яка дорівнює одиниці, а інші координати будуть нуль. Також такі перетворення робляться з наборами `test_data` та `validation_data`. Для того щоб перетворити число у вектор-стовпець, а як в нашому випадку ми хочемо перетворити це у 10-мірний `numpy` масив, то нам треба реалізувати функцію під назвою `vectorized_result`. Нижче наведений лістинг коду цієї функції:

```
def vectorized_result(j):
    |
    e = np.zeros((10, 1))
    e[j] = 1.0
    |
    return e
```

Рис. 3.14. Реалізація методу vectorized_result.

Отже, таким чином в нас вже повністю готовий файл, який містить функції, для зчитування нашого набору даних.

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділі було детально та покроково реалізовано програмний код. З цього розділу ми остаточно отримали реалізований клас для майбутнього об'єкту нашої нейронної мережі. Клас містить у собі конструктор по ініціалізації, та реалізацію необхідних нам функцій, математичні моделі яких ми зазначали у попередньому розділі. А саме стохастичний градієнтний спуск, функція зворотного поширення тощо. Також було реалізовано окремий файл, який ми створили для того щоб, в ньому реалізувати всі функції для роботи з нашим дата сетом .

					ДП 6224. 03.000 ПЗ	Арк.
						55
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 4

АНАЛІЗ ГОТОВОЇ МЕРЕЖІ ТА ЇЇ РОБОТИ

4.1 Огляд роботи реалізованого проекту

Тепер коли в нас повністю реалізовані усі необхідні додатки, та функції. Ми можемо розглянути як саме буде працювати наша нейронна мережа.

Отже, нам потрібно відкрити IDLE Python, або скористатися відкрити Python Console нашого проекту, якщо ми знаходимося в PyCharm.

По-перше що ми зробимо, це підключимо модуль `mnist_loader`, а потім зробимо ініціалізацію набором даних для тренування об'єкту нашої мережі.

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
```

Рис. 4.1. Підключення модулю та початкова ініціалізація.

Наступним кроком ми підключимо модуль `network.py`, в якому було реалізовано клас нашої мережі, та створимо об'єкт нашого класу.

```
>>> import network
>>> netw=network.Network([784,30,10])
```

Рис. 4.2. Створення об'єкту мережі.

Треба одразу зазначити, що при створенні об'єкту мережі, до його конструктору по ініціалізації ми передаємо список який буде визначати топологію нашої мережі. А саме, ці три числа які ми передаємо в списку визначають те, що наша мережа містить 3 шари. Де вхідний шар містить у собі 784 нейрони, другий шар 30, та вихідний 10.

Вже після цього ми можемо запустити нашу мережу для навчання, а якщо казати більш точніше то ми запустимо стохастичний градієнтний спуск.

```
>>> netw.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Рис. 4.3. Створення об'єкту мережі.

Таким чином, після того як ми запустимо строку ,яка зображена на малюнку 4.3, ми запускаємо нашу нейронну мережу на навчання, яка включає в себе 30 епох. Взагалі параметри які ми вказуємо при запуску навчання будуть наступні: кількість епох навчання, навчальна вибірка, розмір підвибірки, тестуюча вибірка, швидкість навчання. При запуску, треба почекати кілька хвилин, доки мережа буде тренуватися. А також, в цей самий час, поки вона тренується, буде відображатися інформація, про пройдені епохи. Тобто в кожній епосі буде відбуватися, вивід відношення правильно розпізнаних цифр, до сумарної кількості цифр у тестовій виборці. Наприклад запис по типу *Epoch 9: 9453 / 10000*, буде значиме , що на 9 епосі тренування досягнута точність розпізнання $9453 / 10000 = 0.94$, тобто 94 відсотки. Нижче відображене результат роботи мережі по розпізнанню чисел.

```
Epoch 0: 9087 / 10000
Epoch 1: 9236 / 10000
Epoch 2: 9280 / 10000
Epoch 3: 9332 / 10000
Epoch 4: 9402 / 10000
Epoch 5: 9367 / 10000
Epoch 6: 9398 / 10000
Epoch 7: 9449 / 10000
Epoch 8: 9448 / 10000
Epoch 9: 9453 / 10000
Epoch 10: 9477 / 10000
Epoch 11: 9410 / 10000
Epoch 12: 9456 / 10000
Epoch 13: 9454 / 10000
Epoch 14: 9473 / 10000
Epoch 15: 9485 / 10000
Epoch 16: 9487 / 10000
Epoch 17: 9476 / 10000
Epoch 18: 9480 / 10000
Epoch 19: 9477 / 10000
Epoch 20: 9492 / 10000
Epoch 21: 9474 / 10000
Epoch 22: 9477 / 10000
Epoch 23: 9479 / 10000
Epoch 24: 9492 / 10000
Epoch 25: 9489 / 10000
Epoch 26: 9495 / 10000
Epoch 27: 9506 / 10000
Epoch 28: 9515 / 10000
Epoch 29: 9505 / 10000
```

Рис. 4.4. Результат роботи мережі по розпізнанню рукописних символів

					ДП 6224. 03.000 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		57

ВИСНОВКИ ДО РОЗДІЛУ 4

В даному розділі ми остаточно розглянули як працюватиме наша мережа, по розпізнанню чисел. Перше, що ми зробили, це підключили файл з функціями для роботи з дата сетом, та підключили наш великий обсяг даних до трьох змінних. Та вже наступним кроком ми створили нашу мережу, та розпочали її тренування. Якщо слідкувати за епохами навчання, то як ми можемо помітити, що на кожній наступні епосі навчання, процент точності, з розпізнання чисел стає вище та вище. Це значиме що роботу зроблено добре. Для порівняння, перша епох яка номер 0, має точність 90 відсотків, але вже 30 епоха, яка мала номер 29, мала вже результат точності 95 відсотків. Що дуже добрим результатом, та що вказує на те, що ми з нашою роботою вправилися добре.

					ДП 6224. 03.000 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		58

ВИСНОВКИ

В даній дипломній роботі, було детально розглянуто тему нейронної мережі. Структура диплома складається з 4 розділів. В кожному розділі було оглянуто необхідний матеріал для розробки проекту у відповідності до теми. Ми розпочали від огляду теоретичного матеріалу, перейшли до математичної реалізації. Після визначення математичної реалізації, яка було дуже важливою та складною частиною цього розділу, оскільки дуже багато застосовується практичних вирішень за допомогою навичок математичного аналізу. Зокрема це торкається таких тем, як стохастичний градієнтний спуск, метод зворотного поширення, функція вартості тощо.

З точки зору реалізації програмного продукту. То тут труднощів не виникало, оскільки нейронна мережа було реалізована на мові програмування Python, з цієї мови програмування ми скористалися сторонніми бібліотеками numpy. Тому що цей модуль містить у собі потужні функції для обробки великих масивів та великий обсяг даних.

Після реалізації було розглянуто роботу мережі, та було остаточно визначено що точність розпізнання нашої мережі складає приблизно 95 відсотків, що є гарним результатом, для того щоб цей проект можна було цілком використовувати у житті, для розпізнання рукописних цифр.

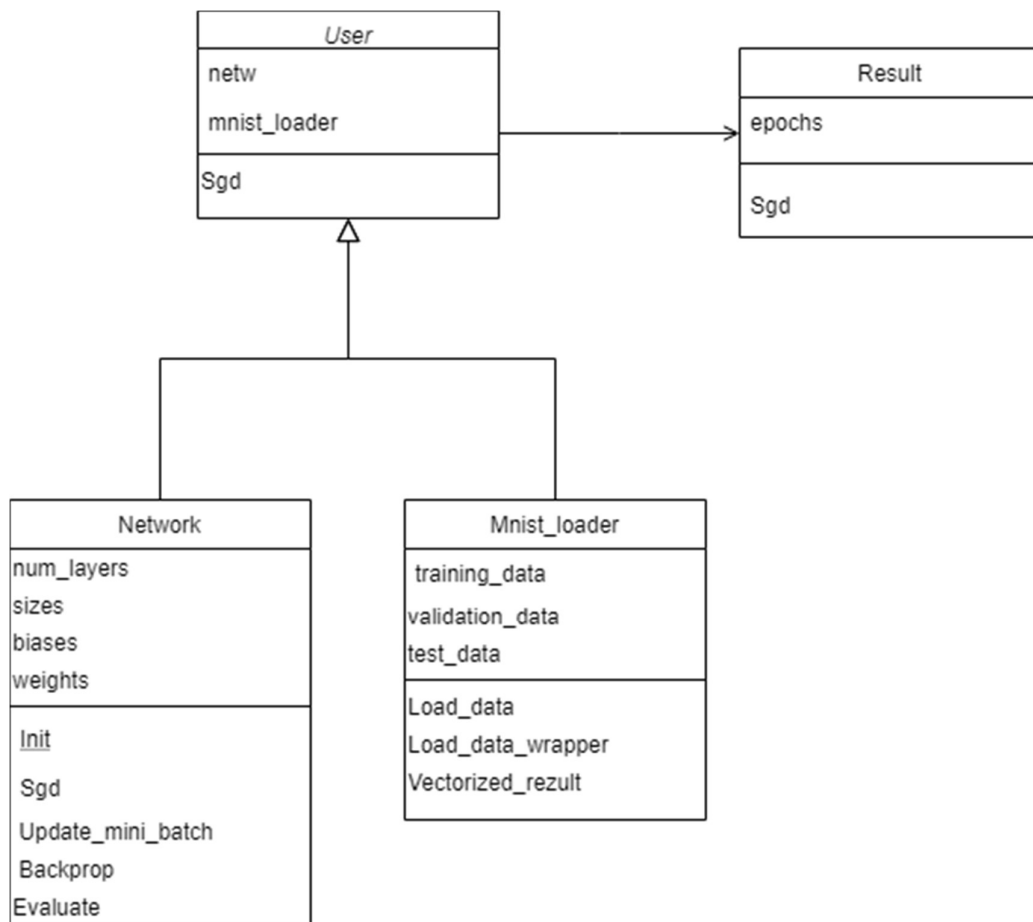
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Марк Лутц, Программирование на Python, том 1 [Текст]. – О’Reilly. – 2011. – 992 с.
2. Марк Лутц, Изучаем Python [Текст]. – О’Reilly. – 2011. – 720 с
3. Марк Лутц, Python. Кишеньковий довідник [Текст]. – О’Reilly. – 2015. – 320 с.
4. Гудфеллоу Я., Бенджио И., Курвилль А. – Глубокое обучение, 2017 г.
5. Франсуа Шолле – Глубокое обучение на Python, 2018 г.
6. Орельен Жерон – Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow, 2018 г.
7. Ричард Саттон, Эндрю Барто – Обучение с подкреплением, 2017 г.
8. «Легкий спосіб вивчити Python», Зед Шоу
9. «Автостопом по Python», Рейтц Кеннет
10. Как создать собственную нейронную сеть с нуля на языке Python. – Режим доступа: <https://neurohive.io/ru/tutorial/kak-sozdat-nejronnuju-set-s-nulja-na-jazyke-python/> - Дата доступа: грудень: 2019
11. Наглядное введение в нейросети на примере распознавания цифр – Режим доступа: <https://proglib.io/p/neural-network-course> - Дата доступа: січень: 2019
12. Язык программирования Python 3 для начинающих и чайников – Режим доступа: <https://pythonworld.ru/> - Дата доступа: грудень: 2019
13. Программирование на Python: от новичка до профессионала – Режим доступа: <https://proglib.io/p/python-from-newbie-to-professional/> – Дата доступа: січень: 2019
14. Программирование на Python: от новичка до профессионала – Режим доступа: <https://proglib.io/p/python-from-newbie-to-professional/> – Дата доступа: січень: 2019
15. Машинное обучение для начинающих: создание нейронных сетей – Режим доступа: <https://python-scripts.com/intro-to-neural-networks> Дата доступа: січень: 2019

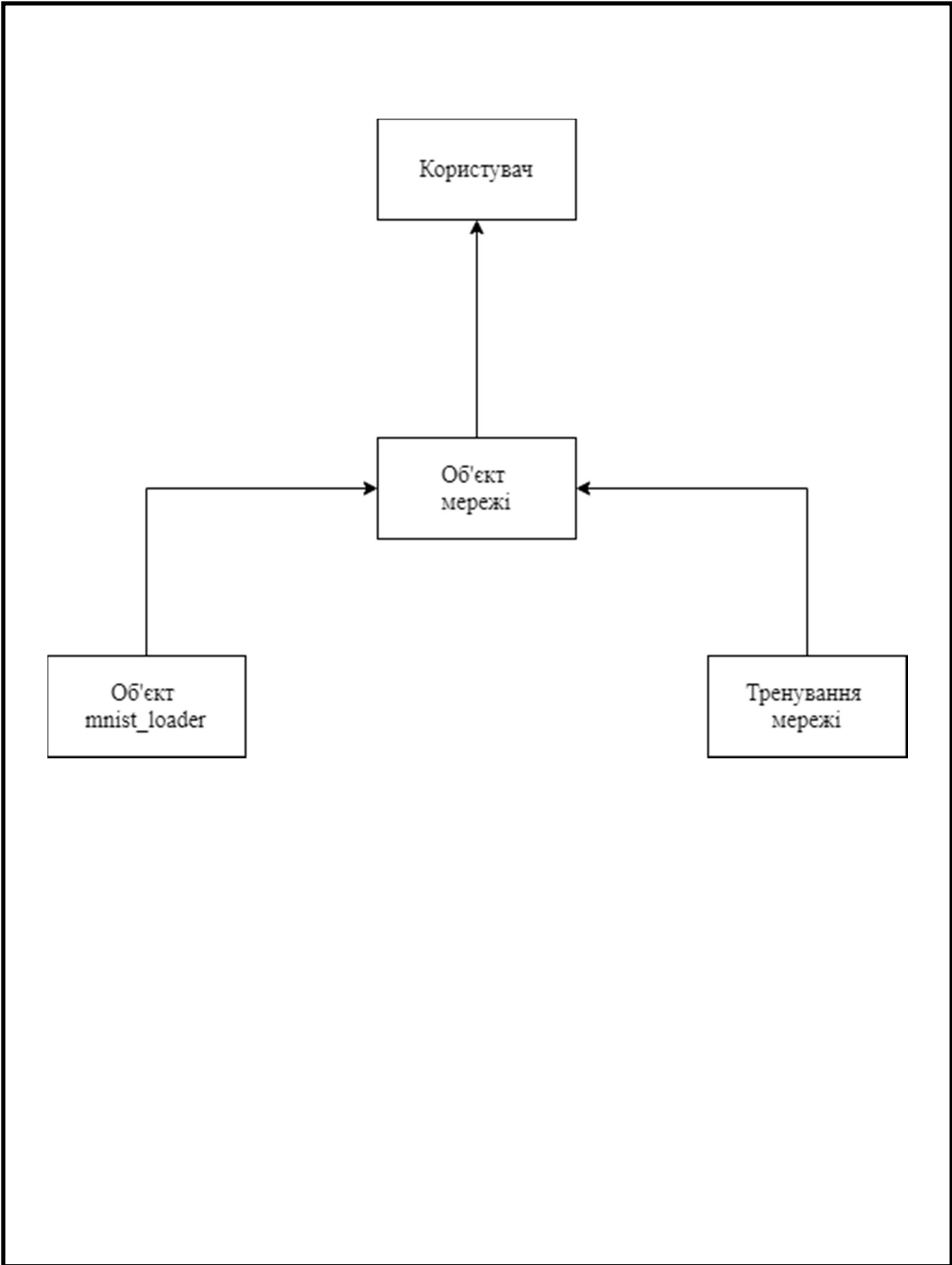
16. Нейронные сети для чайников: Искусственный интеллект против человеческого разума— Режим доступа: <https://yablyk.com/626140-nejronnye-seti-dlya-chajnikov-iskusstvennyj-intellekt-protiv-chelovecheskogo-razuma/> Дата доступа: лютий: 2019
17. Градиентный спуск по косточкам— Режим доступа: <https://habr.com/ru/post/467185/> Дата доступа: січень: 2019

					<i>ДП 6224. 03.000 ПЗ</i>	Арк.
						61
Зм.	Арк.	№ докум.	Підпис	Дата		

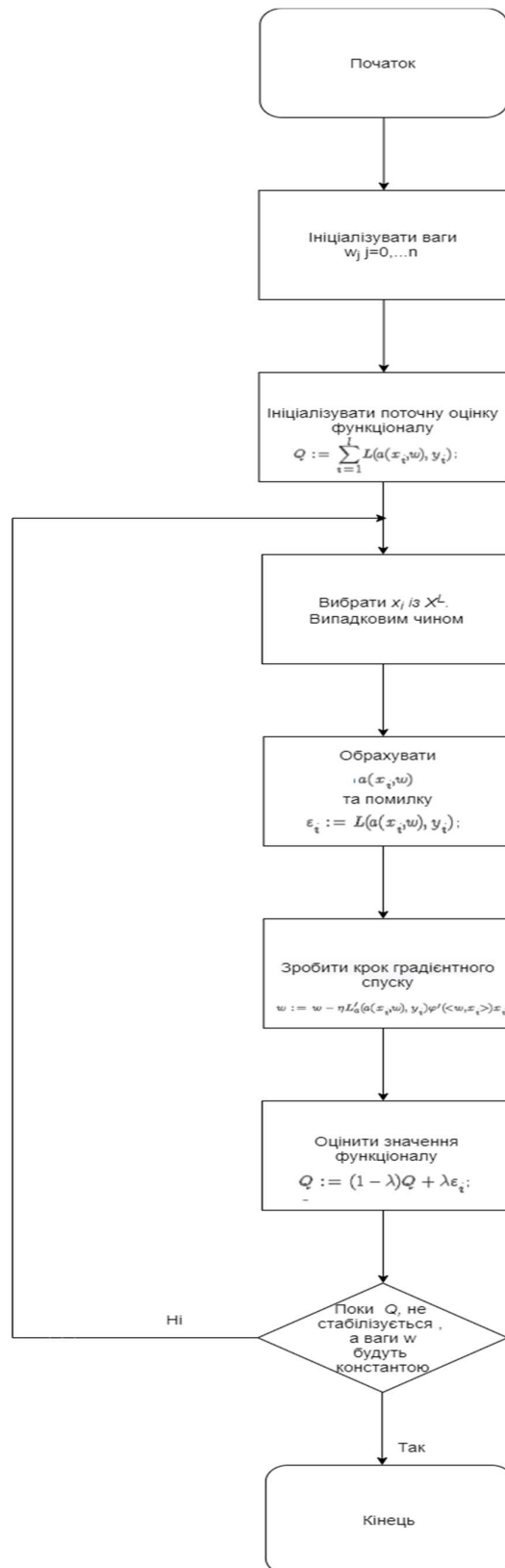
Додатки



					ДП.6224.04.000 Д1			
Зм.		№ документа	Підп.	Дата				
Розробив		Смирнов М.А			Структурна схема Додаток А		Літ.	Аркуш
Перевір.		Луцький Г.М					Т	1
Н.контр		Симоненко В.П.					НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 62	
Затв.		Стіренко С.Г						



				ДП.6224.05.000 Д2								
Зм.		№ документа	Підп.	Дата	Функціональна схема Додаток Б				Літ.	Аркуш		
Розробив	Смирнов М.А								Т		1	2
Перевір.	Луцький Г.М								НТУУ «КПІ імені Ігоря Сікорського», ФІОТ			
									Група ІО - 62			
Н.контр	Симоненко В.П.											
Затв.	Стіренко С.Г											



Зм.		№ документа	Підп.	Дата
Розробив		Смирнов М.А		
Перевір.		Луцький Г.М		
Н.контр		Симоненко В.П.		
Затв.		Стіренко С.Г		

ДП.6224.06.000 ДЗ

Блок схема алгоритму стохас-
тичного градієнтного спуску
Додаток В

Літ.	Аркуш	
Т	1	3
НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 62		