ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення автоматизованих систем

Пояснювальна записка

до кваліфікаційної роботи
на ступінь вищої освіти магістр

на тему: **«Розробка нейронної мережі з використанням мови програмування Python»**

Виконав: студент 6 курсу, групи ІСДМ-61
спеціальності 126 Інформаційних систем та технологій
освітня програма «Інформаційні системи та технології»
(шифр і назва спеціальності)

Бондаренко Д.А.
(прізвище та ініціали)

Керівник Ткаленко О.М.
(прізвище та ініціали)

Рецензент
(прізвище та ініціали)

Нормоконтролер
(прізвище та ініціали)

Київ – 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення автоматизованих систем
Ступінь вищої освіти «Магістр»
Напрям підготовки 126 Інформаційних систем та технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
автоматизованих систем
_____К.П.Сторчак
“____”_____2022 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Бондаренку Данилу Андрійовичу

(прізвище, ім’я, по батькові)

1. Тема роботи Розробка нейронної мережі з використанням мови програмування Python

Керівник роботи Ткаленко Оксана Миколаївна, к.т.н., доц.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від “___” 2022 року №

2. Строк подання студентом роботи: 10.01.2023

3. Вихідні дані до роботи:
Інтегроване середовище розробки;
Інформаційна платформа;
Програмне забезпечення;
Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
4.1 Дослідження необхідності впровадження систем на основі нейронних мереж в IoT;
4.2 Аналіз програмного забезпечення та інструментальних засобів для розробки нейронних мереж;
4.3 Результати виконаної роботи та висновки.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень):
5.1 Актуальність і завдання роботи;
5.2 Перцептрон;

5.3 Будова нейрона/перцептрона;
5.4 Багатошаровий перцептрон;
5.5 Навчання мережі;
5.6 Результати тестування;
5.7 Механізм передбачення результатів;
5.8 Датасет для тренування мережі;
5.9 Середовище розробки;
5.10 Мова програмування;
5.11 Висновки по роботі.

6. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1.	Підбір науково-технічної літератури	15.09.2022-29.09.2022	Виконано
2.	Вивчення матеріалів для подальшої взаємодії з ними	30.09.2022-01.10.2022	Виконано
3.	Дослідження необхідності впровадження нейронних мереж в IoT, та подальшому їх розвитку	01.10.2022-19.10.2022	Виконано
4.	Оцінка можливостей нейронних мереж в IoT	20.10.2022-21.10.2022	Виконано
5.	Розробка програми з можливістю класифікації та розпізнавання фото	21.10.2022-14.11.2022	Виконано
6.	Підсумки та оформлення роботи	15.11.2022-19.11.2022	Виконано
8.	Розробка обов'язкових демонстраційних листів, доповідь	20.11.2022-25.11.2022	Виконано
9.	Попередній захист роботи	26.12.2022	Виконано
10.	Подання роботи в деканат	10.01.2023	Виконано

Студент _____ Бондаренко Д.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Ткаленко О.М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи : 70 с., 1 табл., 54 рис., 22 джерела.

ПЕРЦЕПТРОН, НЕЙРОМЕРЕЖА, БАГАТОШАРОВИЙ ПЕРЦЕПТРОН, МОВИ ПРОГРАМУВАННЯ, МЕТОДИ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ, СЕРЕДОВИЩА РОЗРОБКИ, ІНТЕЛЕКТУАЛЬНІ ТЕХНОЛОГІЇ, ШТУЧНИЙ ІНТЕЛЕКТ.

Об'єкт дослідження – використання нейронних мереж для розпізнавання об'єктів на фото.

Предмет дослідження – нейронні мережі з можливістю розпізнавання фотозображень.

Мета роботи – розробка програми на основі нейронних мереж з можливістю розпізнавання фотозображення.

Методи дослідження – евристичні методи навчання мереж, модель навчання з вчителем.

В даній магістерській роботі був проведений аналіз найбільш популярних типів нейронних мереж. Досліджені найпопулярніші мови програмування та середовища розробки для створення нейронних мереж. Виконаний аналіз алгоритмів навчання нейронних мереж.

Проведена оцінка отриманих результатів, виявлені недоліки. Введені оптимізаційні алгоритми, проаналізована структура людського та штучного нейрону. Структурована інформація по навчанню нейронних мереж. Запропонований подальший розвиток нейронної мережі.

Проаналізована доцільність впровадження систем на основі нейронних мереж та IoT.

ЗМІСТ

ВСТУП

Актуальність теми: штучна нейронна мережа - це метод штучного інтелекту, який вчить комп'ютери обробляти дані таким же способом, як і людський мозок. Це тип процесу машинного навчання, що називається глибоким навчанням, який використовує взаємопов'язані вузли або нейрони в шаруватій структурі, що нагадує людський мозок. Він створює адаптивну систему, за допомогою якої комп'ютери навчаються на своїх помилках та постійно вдосконалюються. Таким чином, штучні нейронні мережі намагаються вирішувати складні завдання, такі як резюмування документів або розпізнавання осіб з більш високою точністю. Нейронні мережі допомагають комп'ютерам приймати розумні рішення з обмеженою участю людини. Вони можуть вивчати та моделювати відносини між нелінійними та складними вхідними, та вихідними даними. Існує безліч систем на основі нейронних мереж: прогнозування погоди, підбір відео та музики, розпізнавання об'єктів на фотографіях. Розробка програм на основі нейронних мереж відкриває безліч перспектив так як швидкість їх навчання і точність результатів за останні декілька років значно зросла і сягає майже 100%.

Об'єкт дослідження: використання штучних нейронних мереж для розпізнавання образів на фотозображеннях.

Предмет дослідження: нейронні мережі для розпізнавання образів рослин на фото.

Мета і завдання дослідження: розробка нейронної мережі з використанням мови програмування Python. У роботі передбачено проаналізувати і класифікувати різні способи навчання нейронних мереж, фреймворки мови програмування Python для їх проектування, визначити вимоги до нейронних мереж для розпізнавання образів на фотозображеннях.

Методика дослідження: технології нейронних мереж, функція активації ReLU.

Наукова новизна: нейронна мережа з графічним виводом результату розпізнавання фото, а також механізм прогнозування подальших результатів

роботи.

Практична значущість результатів: розроблена нейронна мережа може використовуватись у сільськогосподарському секторі для класифікації певних видів рослин, а також може бути використана в загальній класифікації зображень після перенавчання під конкретні цілі.

Апробація результатів магістерської роботи:

Бондаренко Д.А. «Вплив IoT у аграрному секторі». Тези доповіді на Всеукраїнській Науково-технічній конференції «Сучасні інфокомунікаційні технології». – Київ, 21 травня 2021 р.

Публікації:

Бондаренко Д.А. «Застосування технологій інтернету речей в сільському господарстві». Стаття у загальногалузевому науково-виробничому журналі «Телекомунікаційні та інформаційні технології», м. Київ.

1 ДОСЛІДЖЕННЯ НЕЙРОІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

1.1 Розвиток штучних нейронних мереж

Ідея нейронних мереж виникла як модель функціонування нейронів у мозку і використовувала зв’язані схеми для імітації розумної поведінки. У 1943 році нейрофізіолог Уоррен МакКаллох і математик Уолтер Піттс представили просту електричну схему. Дональд Хебб розвинув цю ідею далі у своїй книзі «Організація поведінки» (1949), припустивши, що нервові шляхи зміцнюються з кожним наступним використанням, особливо між нейронами, які мають тенденцію активуватися одночасно, таким чином починаючи довгий шлях до кількісного визначення складних процесів мозку.

Дві основні концепції, які є попередниками нейронних мереж:

- Діодно-транзисторна логіка — перетворення безперервного входу на дискретний вихід;
- Теорія Хебба — модель навчання, заснована на пластичності нейронів, запропонована Дональдом Хеббом у його книзі «Організація поведінки», яку часто підсумовують фразою: «Клітини, які працюють разом, з’єднуються разом».

Обидві запропоновані в 1940-х роках. У 1950-х роках, коли дослідники почали намагатися перевести ці мережі на обчислювальні системи, перша мережа Хебба була успішно реалізована в MIT у 1954 році.

Приблизно в цей час Френк Розенблатт, психолог Корнелла, працював над розумінням відносно простіших систем прийняття рішень, присутніх в оці мухи, які лежать в основі та визначають її реакцію на втечу. У спробі зрозуміти та кількісно оцінити цей процес, він запропонував ідею персептрона в 1958 році, назвавши його Mark I Perceptron. Це була система з простим співвідношенням вхідних даних і вихідних даних, змодельована на основі нейрона МакКаллоха-Піттса, запропонованого в 1943 році Уорреном МакКаллохом, нейробіологом, і Уолтером Піттсом, логіком, щоб пояснити складні процеси прийняття рішень у

мозку за допомогою лінійної функції з затримкою. Нейрон МакКаллоха-Піттса приймає вхідні дані, отримує зважену суму та повертає «0», якщо результат нижче порогового значення, і «1» в іншому випадку. Нейрон МакКаллоха-Піттса зображений на рис. 1.1.

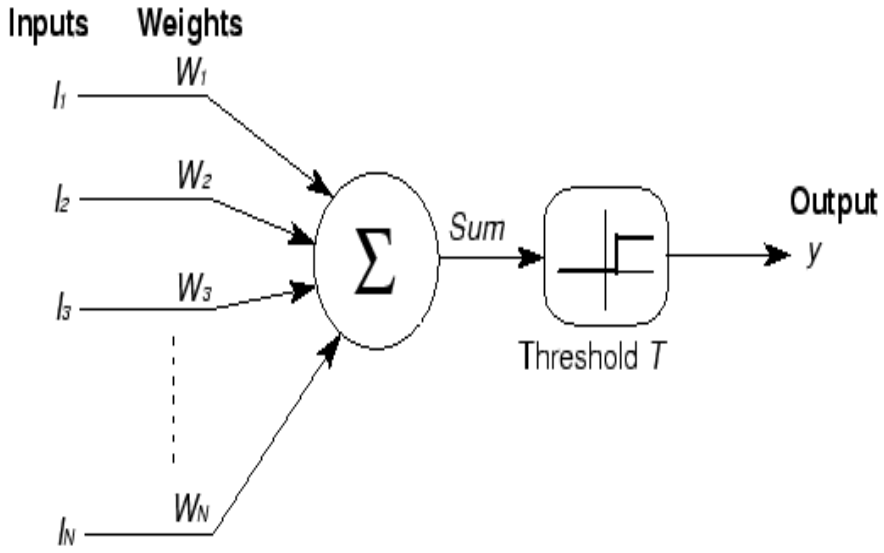

Рисунок 1.1 - Нейрон МакКаллоха-Піттса

Новизна Mark I Perceptron полягала в тому, що його ваги «навчались» через послідовно передані вхідні дані, мінімізуючи при цьому різницю між бажаним і фактичним результатом.

Приблизно в цей же час нейронні мережі почали дуже швидко вдосконалюватись. В 1959 році в Стенфорді Бернард Уідроу та Марсіан Гофф розробили першу нейронну мережу, успішно застосовану для усунення шуму на телефонних лініях. Ці системи отримали назви ADALINE і MADALINE.

У 1969 році дослідник зі світовим ім’ям М. Мінський публікує формальний доказ обмеженості персептрону, а відповідно і його нездатність вирішувати досить широке коло завдань. Все це разом узяте призводить до зниження інтересу багатьох дослідників до нейронних мереж.

Кількома роками пізніше, завдяки дослідженню таких вчених, як Кохонен, Гроссберг, Андерсон, сформувався теоретичний фундамент, на основі якого стало можливим конструювання потужних багатошарових мереж. Проте проблема

полягала у їхньому навчанні. У 1974 р. П. Вербосом розроблено алгоритм зворотного поширення помилки для навчання багатошарових персептронів, перевідкритий наново в 1982 р. Д. Паркером і в 1986 році Девідом І. Румельхартом, Дж. Є. Хінтоном і Рональдом Дж. Вільямсом. Цей систематичний метод навчання багатошарових мереж долає обмеження, зазначені Мінським.

Подальші дослідження показали, що цей метод не є універсальним, незважаючи на багато успішних практичних результатів. Проблема полягає у дуже довгому процесі навчання, а в деяких випадках мережа може взагалі не навчитися. Останнє можливе з двох причин: параліч мережі та потрапляння до локального мінімуму.

1975 р. Фукусіма представляє когнітрон - мережу, що самоорганізується, призначену для інваріантного розпізнавання образів.

1980 р. у спробах поліпшити когнітрон Фукусімою була розроблена потужна парадигма, названа неокогнітрон.

1982 р. Дж. Хопфілд розробив нейронну мережу із зворотними зв'язками. Хоча мережа мала цілий ряд недоліків і не могла бути використана на практиці, вчений заклав основи нейронних рекурентних мереж, після чого про штучні нейронні мережі стало можливим говорити як про асоціативну пам'ять.

1982 р. Кохоненом представлені моделі мережі, що навчається без вчителя на основі самоорганізації.

1987 р. Роберт Хехт-Нільсон, вирішуючи тимчасові обмеження мережі зворотного поширення помилки, розробив мережі зворотного поширення (ЗВР). Час у таких мережах навчання порівняно із зворотним розповсюдженням може зменшуватись у сто разів.

Наступною проблемою штучних нейронних мереж виявилася проблема стабільності-пластичності, суть якої в тому, що навчання новому образу знищує або змінює результати попереднього навчання.

Починаючи з 2000 р. проблема попадання в локальний мінімум була вирішена, зокрема, застосуванням стохастичних методів навчання (метод Больцмана, метод Коші).

2007 р. Джеффрі Хінтоном в університеті Торонто створено алгоритми глибокого навчання багатошарових нейронних мереж. Успіх обумовлений тим, що Хінтон під час навчання нижніх шарів мережі використовував обмежену машину Больцмана (RBM (англ.) -Restricted Boltzmann Machine).

Наразі створюється все більше нових моделей нейронних мереж які мають свої особливості.

1.2 Аналіз біологічного нейрона

В даний час інтерес до штучних нейронних мереж швидко зростає. Цей інтерес підігрівається здатністю фахівців у різних областях, таких як філософія, технічний дизайн, фізіологія та психологія, застосовувати нові технології у своїх дисциплінах.

Штучні нейронні мережі починають свій розвиток у біології, тому що вони складаються з елементів, функціональність яких відповідає частині елементарних функцій біологічного нейрона. Ці елементи потім формуються відповідно до методу, який може відповідати чи не відповідати анатомії мозку. При розгляді мережевих конфігурацій та алгоритмів вчені використовують терміни, взяті з принципів організації мозкової діяльності. На даний момент знання про роботу мозку у людини досить обмежені. Більшість моделей є припущеннями, які ще не були підтверджені з практичної точки зору. Відповідно, при пошуку структур, які можуть виконувати корисні функції, творці мережі не обмежуються біологічними знаннями. У в більшості випадків це призводить до необхідності відмовитися від біологічної ймовірності того, що мозок є просто метафорою, створюючи мережі, які неможливі в живій матерії, або які вимагають неймовірно великих припущень про анатомію та функціонування мозку. Хоча зв’язок з такою наукою, як біологія, у деяких місцях досить слабкий та незначний, штучні нейронні мережі продовжують порівнювати з людським мозком. Їхнє функціонування часто схоже з людським пізнанням, тому важко уникнути цієї аналогії.

Таким чином, можна стверджувати, що штучна нейронна мережа – це

математична модель, побудована на засадах людського мозку. Штучна нейронна мережа - це набір взаємопов’язаних штучних нейронів, зв’язки між якими залежать від вибраної топології мережі. Нервова система людини, що складається з нейронів, є досить складною. Близько 10^{11} нейронів беруть участь у приблизно 10^{15} лініях передачі, довжина яких становить один метр чи більше. Кожен нейрон має багато спільних характеристик з іншими органами тіла, але він також має абсолютно унікальну здатність приймати, обробляти та передавати електрохімічні сигнали по нервах, які становлять систему зв’язку мозку. На рис. 1.2 зображений взаємозв’язок між нейронами.

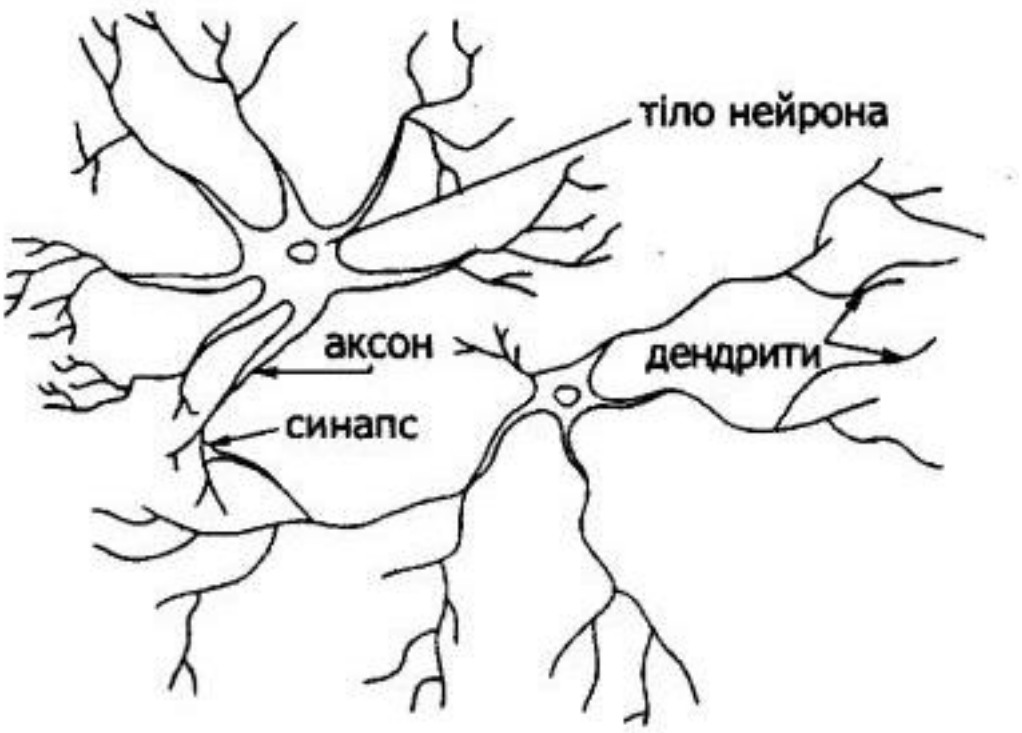

Рисунок 1.2 - взаємозв'язок між нейронами

Процеси нервових волокон або дендритів мігрують із тіла нервової клітини до інших нейронів, де вони отримують імпульси у вузлах, так звані синапси. Вхідні сигнали, отримані синапсом, надходять у тіло нейрона. Тут вони підсумовані, деякі входи стимулюють нейрон, інші перешкоджають його стимуляції. Коли збудження у тілі нейрона перевищує певний поріг нейрон стає збудженим і посилає сигнал вздовж аксона іншим нейронам. Ця базова функціональна схема має багато труднощів та винятків. Однак більшість штучних нейронних мереж лише

моделюють ці прості властивості.

1.2 Аналіз штучного нейрона

Штучний нейрон моделює властивості біологічного нейрона. На вхід штучного нейрона надходить певний набір сигналів, кожен із яких є виходом іншого нейрону. Кожен вхід множиться на відповідну вагу, яка дуже схожа на силу синапсів, і всі вони складаються разом та визначають ступінь активації нейрона. На рис. 1.3 відображена така модель. Ця конфігурація є основою багатьох мережевих парадигм, незважаючи на те, що всі вони досить різноманітні. X1, x2, xn – це кілька вхідних сигналів, які йдуть на штучний нейрон. Ці сигнали разом позначаються вектором X. Вони відповідають сигналам, які надходять до синапсів біологічного нейрона. Кожен сигнал множиться на відповідну вагу w1, w2, …, wn, та вводить значення суми, позначену ∑. Кожна вага відповідає «силі» одного біологічного синаптичного зв'язку (набір ваг, в сукупності званий вектором W). Одиниця суми, що відповідає тілу біологічного елемента, додає алгебраїчно зважені вхідні дані, генеруючи вихідні дані NET (тобто вихідний результат нейронної мережі.

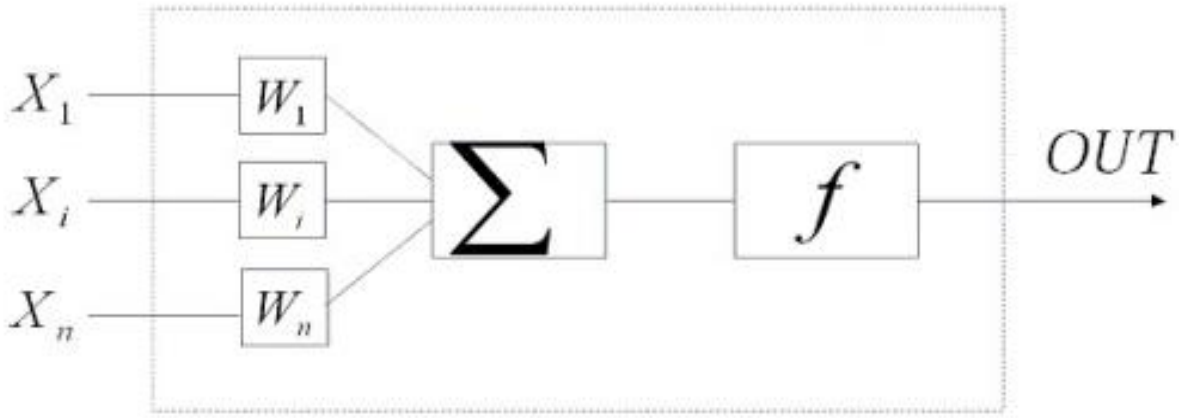

Рисунок 1.3 - Штучний нейрон з функцією активації

Потім сигнал NET зазвичай перетворюється на функцію активації F і подає нейронний вихідний сигнал OUT. Функція активації може мати інший характер. Найпростіший варіант – звичайна лінійна функція (1.1):

$$\mathrm{OUT} = \mathrm{F(NET)} = \mathrm{K} \times \mathrm{NET} = \begin{pmatrix} \mathrm{K},\ \mathrm{NET} > \mathrm{T} \\ 0, \mathrm{NET} \leq \mathrm{T} \end{pmatrix} \quad (1.1)$$

де K – стала порогової функції,

а T – поріг функції активації.

Для більш точного моделювання біологічного нейрона та його нелінійної характеристики використовуються нелінійні функції. Це дозволяє нейронній мережі в теорії мати великий потенціал. На рисунку 1.3 блок, позначений як F приймає сигнал NET і видає сигнал OUT. Якщо блок F обмежує діапазон зміни значення NET так, що для будь-яких значень NET значення OUT належать кінцевому інтервалу, F називається функцією стиснення (обмеження). За аналогією з електронними системами, функцію активації можна розглядати як нелінійну особливість посилення штучного нейрона. Коефіцієнт посилення розраховується як відношення збільшення значення OUT до невеликого збільшення значення NET, що його викликав. Він виражається нахилом кривої за певного рівня збудження та змінюється від невеликих значень з великими негативними збудженнями (майже горизонтальна крива) до максимального значення при нульовому збудженні, і знову зменшується, коли збудження стає позитивним.

У 1973 році Гроссберг виявив, що така нелінійна характеристика вирішує проблему перенасичення шуму. Як нейронна мережа може обробляти як слабкі, і сильні сигнали? Слабкі сигнали вимагають великого підсилення і «бажання» мережі отримати корисний вихід. Однак ступінь посилення з високим коефіцієнтом підсилення можуть призвести до перенасичення вихідного сигналу шумом підсилювача (випадковими флуктуаціями), присутнім у фізично реалізованій мережі. Сильні вхідні сигнали, свою чергу, насичують каскади підсилювача, тим самим виключаючи можливість корисного виводу. Центральна область логістичної функції, яка має велике посилення, вирішує проблему обробки слабких сигналів, тоді як області з зменшенням посилення на позитивному і негативному кінцях підходять для великих збуджень.

Таким чином, нейрон працює з високим коефіцієнтом посилення у широкому діапазоні рівнів вхідного сигналу. Логістична або «сигмоїдальна» (S-подібна)

функція часто використовується як функція для стиснення і корекції результату. Вона зображена на рис. 1.4.

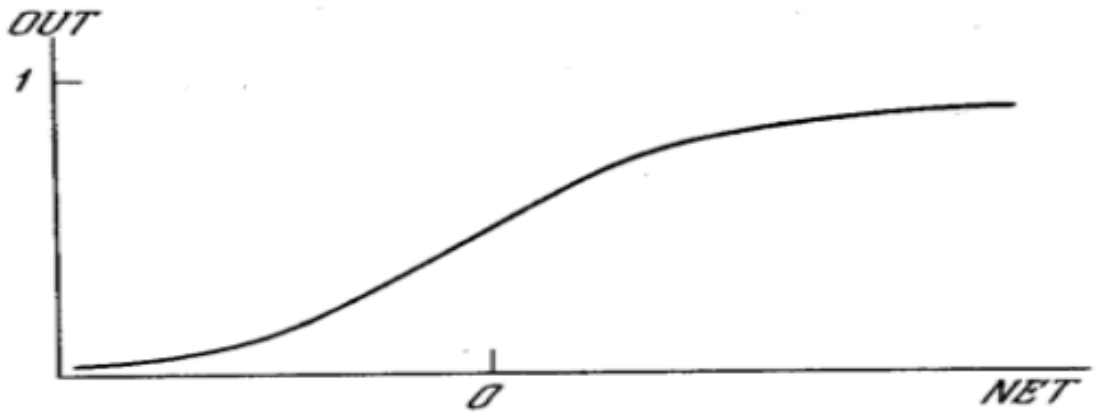

Рисунок 1.4 - Сигмоїдальна логічна функція

Математично, таку функцію можна зобразити формулою (1.2):

$$OUT = F(NET) = \frac{1}{1 + e^{-NET}} \quad (1.2)$$

Інша функція активації, яку часто використовують – це функція гіперболічного тангенса. За формою він дуже схожий на логістичну функцію і часто використовується біологами як математична модель для активації нервових клітин. Подібно до логістичної функції, гіперболічний тангенс є S-подібною функцією, але симетрична до початку координат, і в точці, де NET = 0 значення вихідного сигналу OUT дорівнює нулю. На відміну від логістичної функції, гіперболічний тангенс набуває значень різних знаків, що є вигідним для низки мереж. Ця функція представлена на рис. 1.5. Математично представлено у вигляді формули (1.3).

$$OUT = F(NET) = th(NET) \quad (1.3)$$

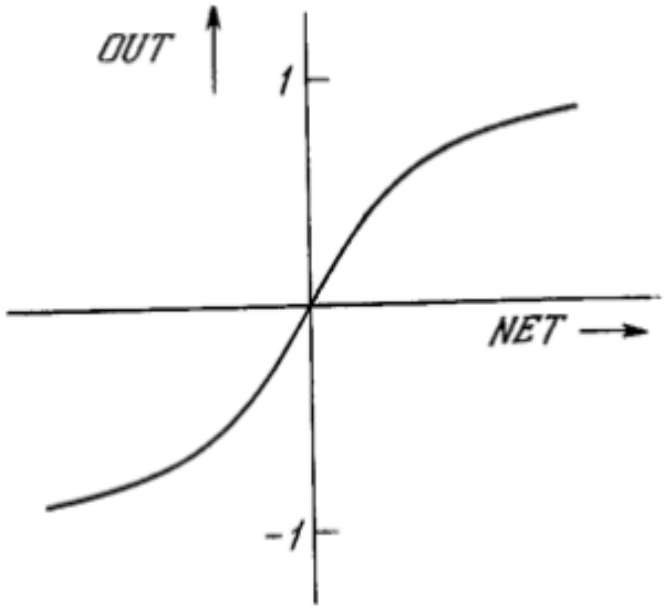

Рисунок 1.5 – Функція гіперболічного тангенса

Незважаючи на те, що нейрон здатний виконувати найпростіші методи виявлення зображень, потужність нейронних обчислень походить із того, як сильно зв’язані нейрони у мережі. Найпростіша мережа складається з групи нейронів, які утворюють шар. Великі і складні нейронні мережі зазвичай мають набагато більші обчислювальні можливості. Хоча мережі всіх можливих конфігурацій вже були створені, багаторівнева організація нейронів копіює шаруваті структури певних частин мозку. Такі багатошарові мережі мають більше можливостей, ніж одношарові мережі, хоча вони потребують більш складніші алгоритми навчання.

На сьогоднішній день можна класифікувати три типи нейронних мереж в залежності від архітектури:

- Повнозв’язні, зображені на рис. 1.6 (а);
- Багатошарові (згорткові), рис. 1.6 (б);
- Мережа з локальними зв’язками (слабозв’язні), рис. 1.6 (в);

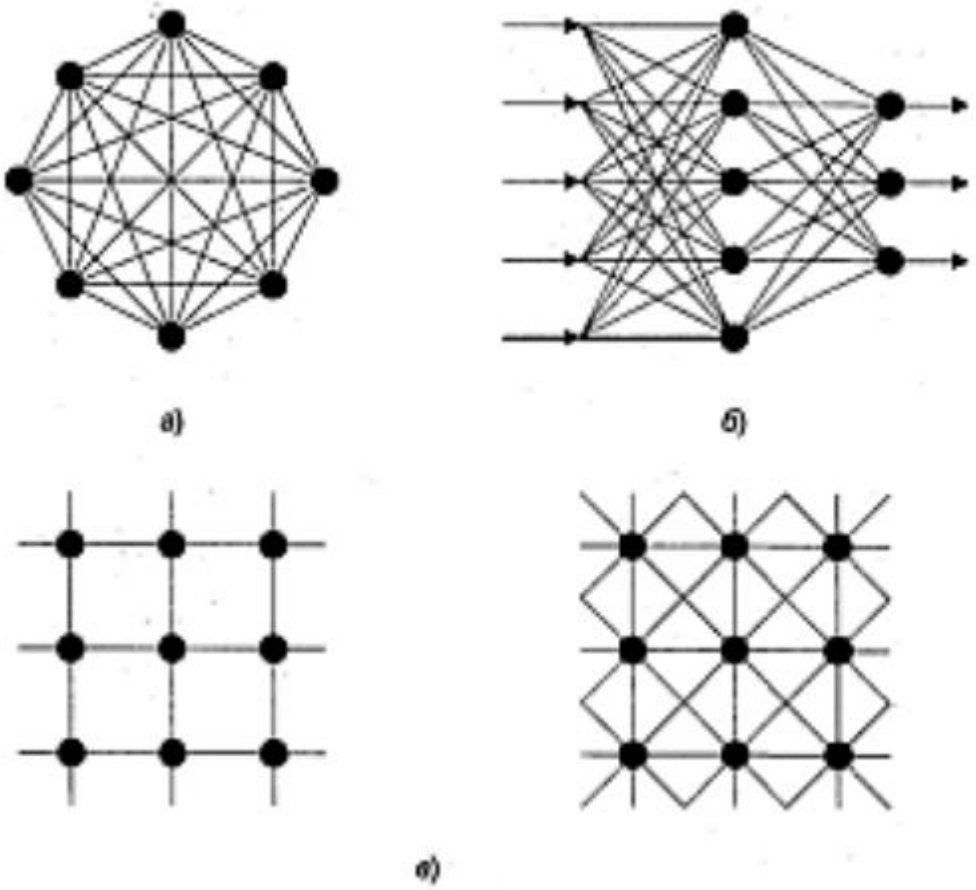

Рисунок 1.6 - Архітектури нейронних мереж

У повністю зв’язаних нейронних мережах кожен нейрон передає свій вихід решті нейронів, включаючи себе. Усі вхідні дані застосовуються до всіх нейронів. Вихідний сигнал мережі може бути цілим або частиною вихідного сигналу нейрона після кількох циклів роботи мережі.

У слабкозв’язних нейронних мережах, нейрони розташовані у вузлах прямокутної чи гексагональної сітки. У багатошаровій нейронній мережі, нейрони згруповані у шари. Ця площина містить нейрони із окремими вхідними сигналами. Кількість нейронів у шарі може бути довільною і не залежить від кількості нейронів в іншому шарі.

На рис. 1.7 та рис. 1.8 зображені мережі з одним шаром, та, відповідно, двома.

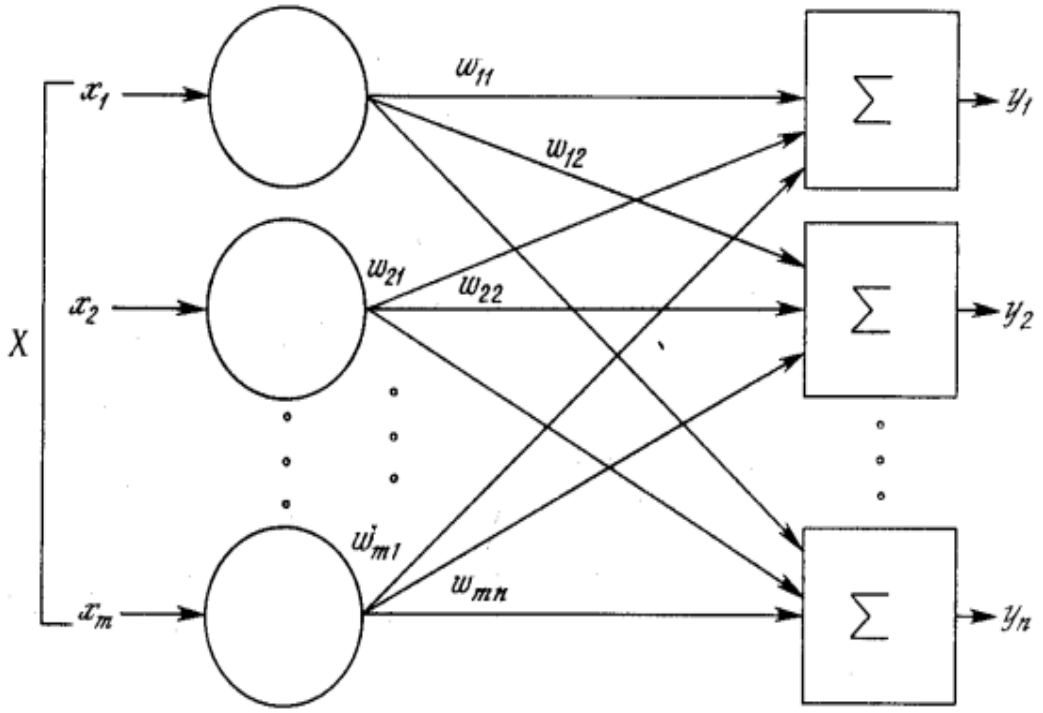

Рисунок 1.7 - Нейронна мережа з одним шаром

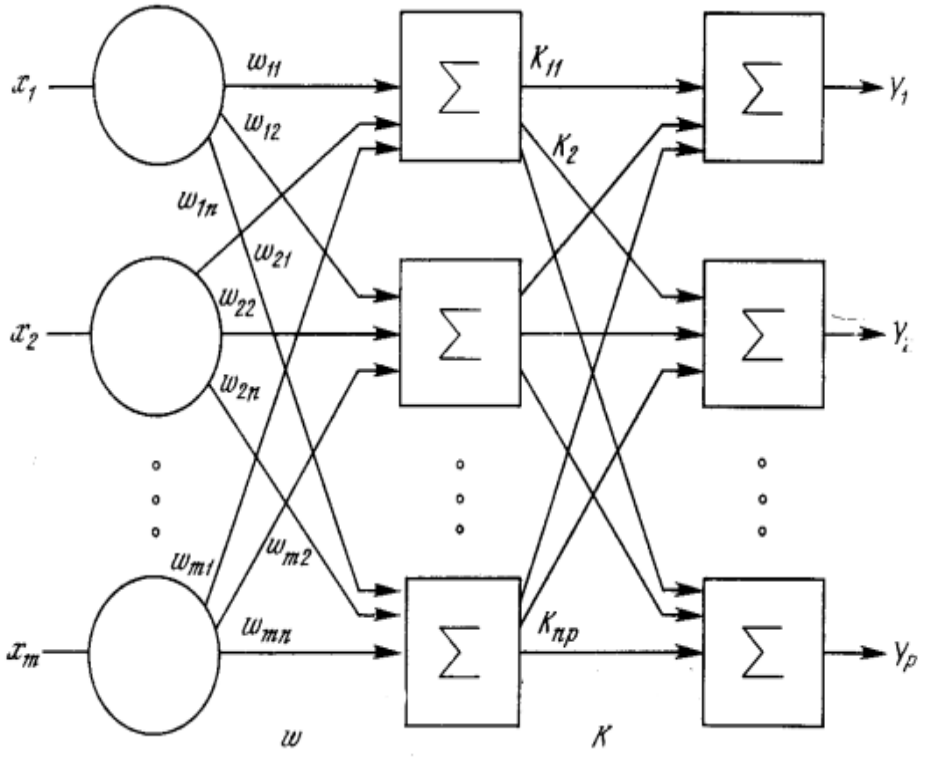

Рисунок 1.8 - Двошарова нейронна мережа

Обидві мережі не мають зворотного зв’язку – з’єднання від виходів конкретної площини з входами тієї ж площині тобто, це мережі без зворотного

зв’язку чи - мережі прямого поширення. Ці мережі представляють інтерес і широко поширені. Мережі найбільш загального виду з підключеннями від виходів до входів називають мережами зворотного зв’язку. Мережі без зворотного зв’язку не мають пам’яті, їх вихід повністю визначається поточними входами та вагами. У деяких конфігураціях мереж зворотного зв’язку попередні вихідні значення повертаються на вхід. Отже, вихід визначається як поточним входом, так і попередніми виходами. За цієї причини мережі зворотного зв’язку можуть мати властивості, подібні до людської короткочасної пам’яті. Вихід мережі частково залежить від попередніх входів.

1.4 Дослідження топологій нейронних мереж

Мною були проаналізовані найпопулярніші топології штучних нейронних мереж. Топологія – це принципова схема, по якій з’єднані нейрони в мережі, це є важливим фактом, який напряму впливає на навчання мережі в цілому.

Ми живемо в еру великих даних, коли всі сфери науки та промисловості генерують величезні обсяги даних. Це стикає нас з безпрецедентними проблемами щодо їх аналізу та інтерпретації. З цієї причини існує нагальна потреба в нових методах машинного навчання та штучного інтелекту, які можуть допомогти у використанні цих даних.

На рис. 1.9 зображена кількість публікацій на тему нейронних мереж, «Глибокого навчання» (англ.- DL, deep Learning), згорткових нейронних мереж (англ. – CNN, convolutional neural network), мереж глибокої довіри (англ.- DBN, deep belief network), мереж довгої короткострокової пам’яті (англ. - LTSM, long short-term memory), автоенкодерів (англ. - AEN, autoencoder), а також багатошарових персептронів (англ.- MLP, multilayer perceptron).

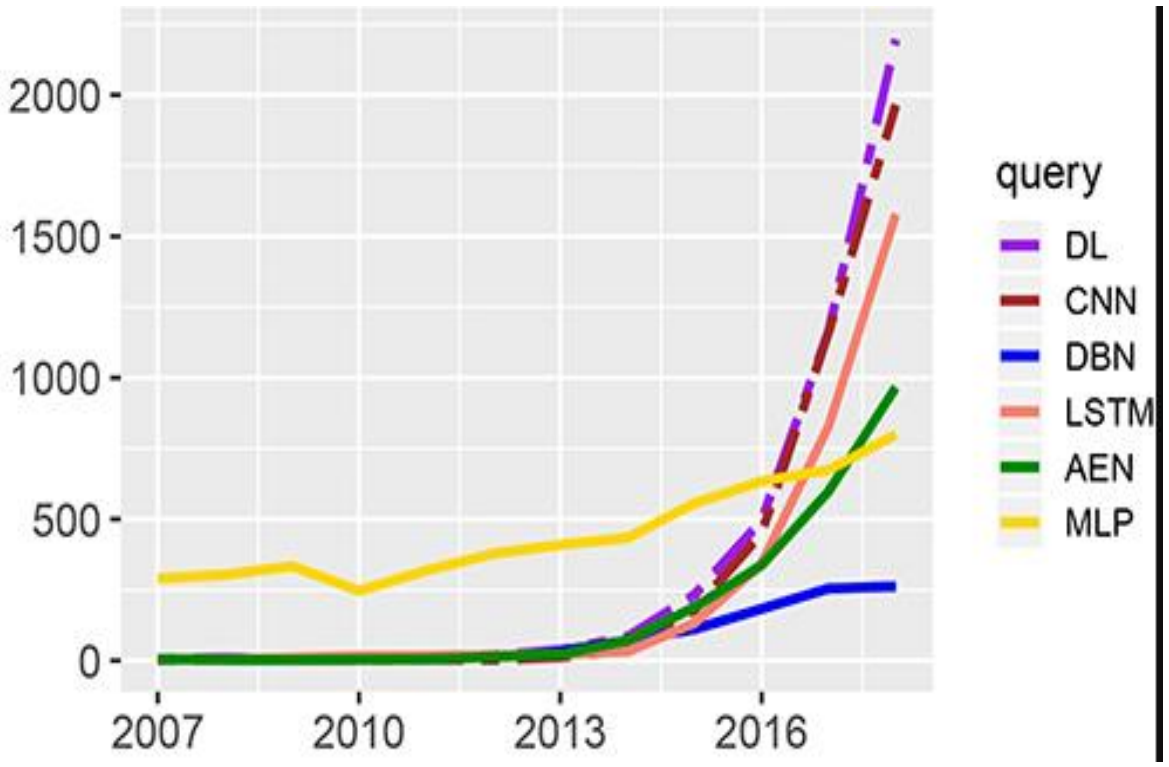

Рисунок 1.9 - Кількість публікацій на тему нейронних мереж

Мережа прямого поширення

Щоб побудувати штучні нейронні мережі (ШНМ), нейрони повинні бути з’єднані один з одним. Найпростішою архітектурою мереж є структура прямого поширення. На рис. 1.10, наведені приклади простої та глибокої мережі прямого поширення.

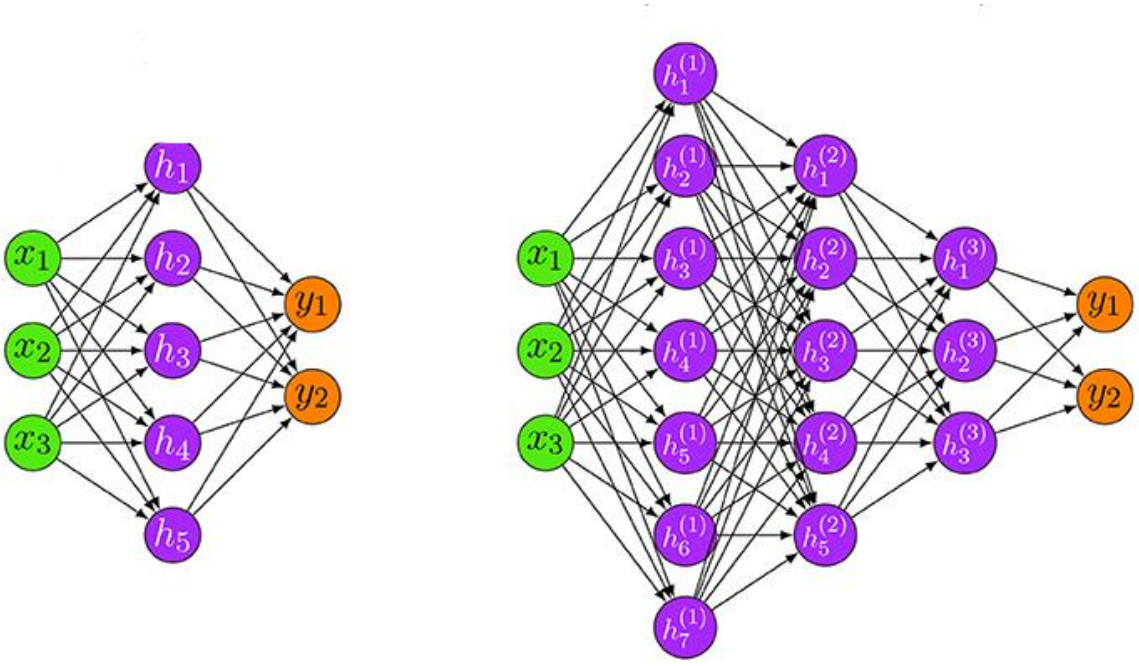

Рисунок 1.10 - Мережі прямого поширення

Літерою x позначені вхідні нейрони, h – прихований шар у якому відбуваються усі необхідні обчислення, літера y відповідає за вихідні дані мережі. Мережа що знаходиться зліва це проста мережа прямого поширення, а та що праворуч – глибинна мережа прямого поширення. Загалом, глибина мережі означає кількість нелінійних перетворень між шарами, тоді як розмірність прихованого шару, тобто кількість прихованих нейронів, називається його шириною.

Нейронна мережа прямого поширення, яку також називають багатошаровим персептроном (MLP), може використовувати лінійні або нелінійні функції активації. Важливо, що в нейронній мережі немає циклів, які б дозволяли прямий зворотний зв'язок. Формула 1.4 визначає, як вихідні дані MLP отримують із вхідних даних.

$$f(x) = \varphi^{(2)}(W^{(2)}\varphi^{(1)}\left(W^{(1)}x + b^{(1)}\right) + b^{(2)}) \quad (1.4)$$

Формула (1.4) є дискримінантною функцією нейронної мережі. Для знаходження оптимальних параметрів потрібне правило навчання. Загальний підхід полягає у визначенні функції помилки (або функції вартості) разом із алгоритмом оптимізації, щоб знайти оптимальні параметри шляхом мінімізації помилки для навчальних даних.

Рекурентні нейронні мережі.

Рекурентні нейронні мережі (RNN) має два підкласи, які можна виділити на основі їхньої поведінки обробки сигналів. Перший містить кінцеві імпульсні рекурентні мережі (FRNs), а другий нескінченні імпульсні рекурентні мережі (IIRNs). Ця відмінність полягає в тому, що FRN задається спрямованим ациклічним графом (DAG), який можна розгорнути в часі та замінити нейронною мережею прямого поширення, тоді як IIRN є спрямованим циклічним графом (DCG), для якого таке розгортання неможливе. На рис. 1.11 зображена рекурентна нейронна мережа.

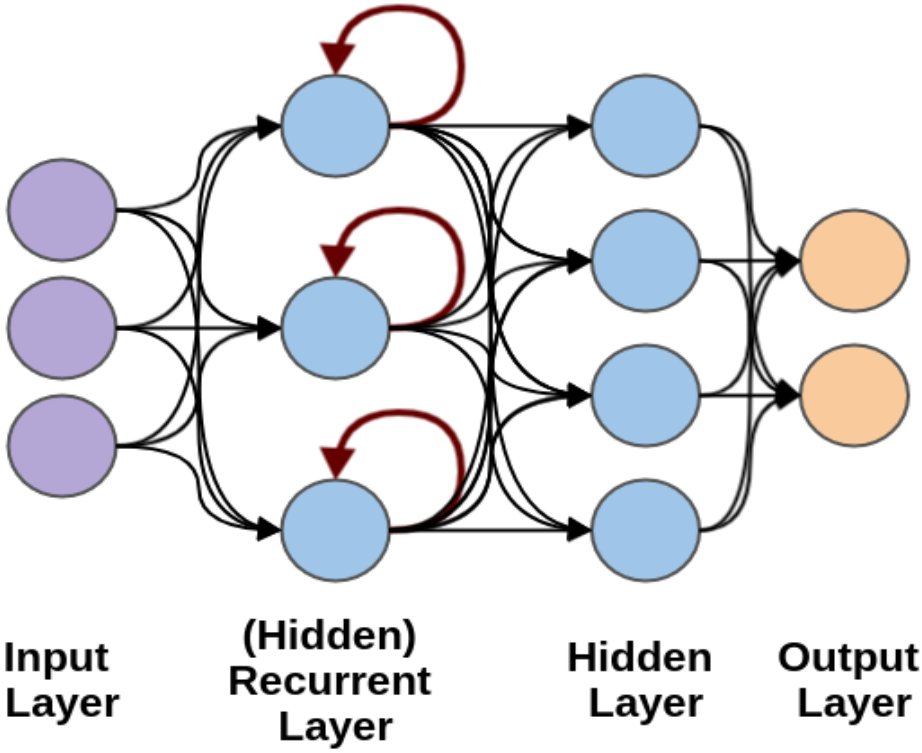

Рисунок 1.11 - Рекурентна нейронна мережа

Рекурентна нейронна мережа - це тип штучної нейронної мережі, яка використовує послідовні дані або дані часових рядів. Ці алгоритми глибокого навчання зазвичай використовуються для порядкових проблем, таких як переклад мови, обробка природної мови, розпізнавання мовлення та підписи до зображень; вони включені в такі популярні програми, як Siri, голосовий пошук і Google Translate. Подібно до нейронних мереж прямого поширення та згорткових нейронних мереж, рекурентні нейронні мережі використовують навчальні вибірки певних даних. Вони відрізняються своєю «пам'яттю», оскільки вони беруть інформацію з попередніх вхідних даних, щоб впливати на поточні вхідні та вихідні дані. Хоча традиційні глибокі нейронні мережі припускають, що входи та виходи незалежні один від одного, вихід рекурентних нейронних мереж залежить від попередніх елементів у послідовності. Хоча майбутні події також були б корисними для визначення результату даної послідовності, односпрямовані рекурентні нейронні мережі не можуть врахувати ці події у своїх прогнозах.

Іншою відмінною характеристикою рекурентних мереж є те, що вони спільно використовують параметри на кожному рівні мережі. У той час як мережі прямого

поширення мають різні ваги для кожного вузла, рекурентні нейронні мережі мають однаковий параметр ваги на кожному рівні мережі. Тим не менш, ці ваги все ще коригуються за допомогою процесів зворотного поширення та градієнтного спуску, щоб полегшити навчання з підкріпленням. Рекурентні нейронні мережі використовують алгоритм зворотного поширення в часі для визначення градієнтів, який дещо відрізняється від традиційного зворотного поширення, оскільки він специфічний для даної послідовності. Принципи алгоритму такі ж, як і традиційного зворотного поширення, коли модель навчається, обчислюючи помилки від вихідного до вхідного рівня. Ці розрахунки дозволяють нам правильно налаштувати та підігнати параметри моделі. Зворотне поширення відрізняється від традиційного підходу тим, що підсумовує помилки на кожному кроці часу, тоді як у інших мережах не потрібно підсумовувати помилки, оскільки вони не поділяють параметри на кожному рівні. Через цей процес рекурентні нейронні мережі, як правило, стикаються з двома проблемами, відомими як вибухові градієнти та зникнення градієнтів. Ці проблеми визначаються розміром градієнта, який є нахилом функції втрат уздовж кривої помилок. Коли градієнт занадто малий, він продовжує зменшуватися, оновлюючи вагові параметри, поки вони не стануть незначними, тобто 0. Коли це відбувається, алгоритм більше не навчається. Вибухові градієнти виникають, коли градієнт занадто великий, що створює нестабільну модель. У цьому випадку ваги моделі виростуть занадто великими, і в кінцевому підсумку вони будуть представлені як нескінченність. Одним із рішень цих проблем є зменшення кількості прихованих шарів у нейронній мережі, усунення частини складності в моделі.

Мережа Хопфілда

Примітно, що мережі Хопфілда були першим прикладом асоціативних нейронних мереж: архітектур рекурентних мереж, які здатні створювати асоціативну пам’ять. Асоціативна пам’ять, або пам’ять з адресацією вмісту - це система, в якій виклик пам’яті ініціюється асоціативністю вхідного шаблону із запам’ятованим. Іншими словами, асоціативна пам’ять дозволяє отримати та використати пам’ять, лише її неповну частину. Наприклад, людина може почути

пісню, яка їй подобається, і «повернутись» до моменту, коли вона почула її вперше. Контекст, людей, оточення та емоції в цій пам’яті можна відновити шляхом подальшого впливу лише на частину оригінального стимулу: лише на пісню. Ці особливості мереж Хопфілда зробили їх хорошими кандидатами для ранніх обчислювальних моделей асоціативної пам’яті людини та поклали початок нової ери в нейронних обчисленнях і моделюванні.

Унікальна мережева архітектура Хопфілда базувалася на моделі Ізінга, фізичній моделі, яка пояснює емерджентну поведінку магнітних полів, створених феромагнітними матеріалами. Модель зазвичай описується як плоский граф, де вузли представляють магнітні дипольні моменти регулярного повторюваного розташування. Кожен вузол може перебувати в одному з двох станів (тобто обертатися вгору або вниз, +1 або -1), і узгодження станів між сусідніми вузлами є енергетично вигідним. Коли кожен диполь «перевертається» або еволюціонує, щоб знайти локальні енергетичні мінімуми, весь матеріал має тенденцію до стаціонарного або глобального мінімуму енергії. Мережі Хопфілда можна розглядати подібним чином, коли мережа представлена як двовимірний графік і кожен вузол (або нейрон) займає один із двох станів: активний або неактивний. Подібно до моделей Ізінга, динамічна поведінка мережі Хопфілда в кінцевому підсумку визначається відповідно до метафоричного поняття «енергії системи», і мережа збігається до стану «енергетичного мінімуму», який у випадку навченої мережі є її пам’ять. Щоб зрозуміти структуру та функцію мережі Хопфілда, корисно розбити якості та атрибути окремого блоку чи нейрона.

Кожен нейрон у мережі має три якості, які слід враховувати:

- з’єднання з іншими нейронами - кожен нейрон у мережі з’єднаний з усіма іншими нейронами, і кожне з’єднання має унікальну силу або вагу, аналогічну синапсу. Ці сили з’єднання зберігаються в матриці ваг;
- активація - обчислюється за допомогою чистого вхідного сигналу від інших нейронів і їхніх відповідних ваг зв’язку, приблизно аналогічних мембранному потенціалу нейрона. Активація приймає одне скалярне значення;

- біполярний стан - це вихід нейрона, обчислений за допомогою активації нейрона та порогової функції, аналогічної «стану активації» нейрона. У цьому випадку -1 і +1.

Інформація або спогади мережі Хопфілда зберігаються завдяки міцності зв'язку між нейронами, приблизно так само, як вважається, що інформація зберігається в мозку відповідно до моделей довготривалої пам'яті. З'єднання та їх відповідні ваги безпосередньо впливають на активність мережі з кожним поколінням і в кінцевому підсумку визначають стан, до якого мережа сходиться. Ви можете уявити вагу між 2 нейронами, нейроном-a та нейроном-b, як ступінь, до якої вихід нейрона-a сприятиме активації нейрона-b, і навпаки.

Докладніше про активацію та її вплив на стан нейронів нижче. Важливо, що зв'язки між нейронами в мережі Хопфілда є симетричними. Це означає, що якщо нейрон-a з'єднаний з нейроном-b із силою +1, зв'язок нейрона-b з нейроном-a також дорівнює +1. Ми можемо зберігати ваги мережі з n нейронів у квадратній симетричній матриці розмірів n x n. Оскільки нейрони не з'єднуються самі з собою, ваги на діагоналі матриці фактично дорівнюють нулю. Важливо відзначити, що структура мережі, включаючи кількість нейронів і їх зв'язки в мережі, є фіксованою. Вага цих з'єднань є гнучким параметром. Коли ми говоримо про «навчання в мережі», ми насправді маємо на увазі точне налаштування кожної ваги в мережі з метою досягнення певного бажаного результату. Початкову конфігурацію стану можна розглядати як вхід, а її кінцеву конфігурацію стану як вихід. Оскільки кожен нейрон оновлюється залежно від лінійних комбінацій своїх входів, уся мережа насправді є однією масивною математичною функцією: динамічною системою, яка з кожною генерацією оновлення приймає свій минулий вихід як поточний вхід. Параметри цієї функції, ваги, визначають шлях розвитку мережі з часом. На формулі (1.5) зображена найпростіша функція активації для імпульсних рекурентних мереж:

$$s = sgn(x) = \begin{cases} +1 \; for \; x \geq 0 \\ -1 \; for \; x < 0 \end{cases} \tag{1.5}$$

Машина Больцмана

Машину Больцмана можна описати як «шумну» мережу Хопфілда, оскільки вона використовує ймовірнісну функцію активації (1.6):

$$p(s_i = 1) = \frac{1}{1 + \exp(-x_i)} \tag{1.6}$$

Ця модель важлива, оскільки це одна з перших нейронних мереж, яка використовує приховані одиниці (латентні змінні). Для вивчення вагових коефіцієнтів можна використовувати алгоритм контрастної дивергенції. Простіше кажучи, машини Больцмана - це нейронні мережі, що складаються з двох шарів - видимого та прихованого. Кожне ребро між двома шарами не спрямоване, що означає - інформація може надходити в двох напрямках. Вся мережа повністю з'єднана, кожен нейрон у мережі з'єднаний з усіма іншими нейронами через неорієнтовані краї.

На рис. 1.12 зображена мережа Больцмана.

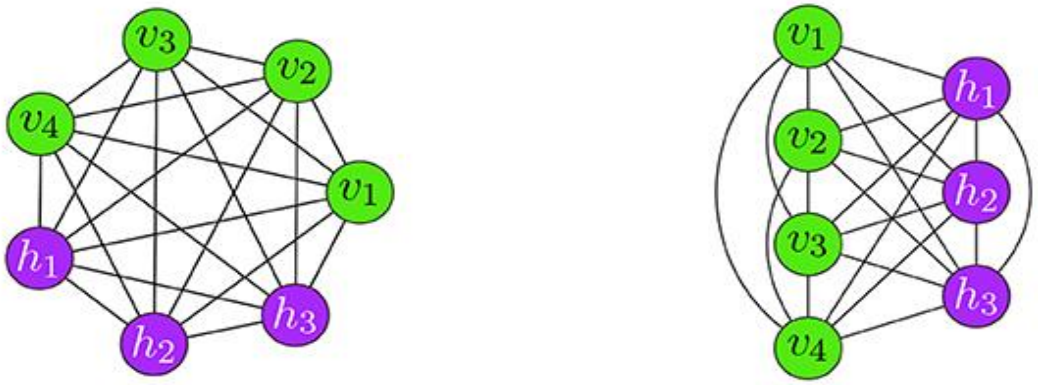

Рисунок 1.12 - мережа Больцмана

Архітектура машини Больцмана включає неглибоку двошарову нейронну мережу, яка також є блоками глибокої мережі. Перший шар цієї моделі називається видимим або вхідним шаром, а другий - прихованим. Вони складаються з нейроноподібних одиниць, які називаються вузлами, а вузли є областями, де відбуваються обчислення. Ці вузли з'єднані один з одним на різних рівнях, але два вузли одного рівня не пов'язані. Таким чином, немає внутрішньорівневого зв'язку - це одне з обмежень у машині Больцмана. Кожен вузол за допомогою обчислень обробляє вхідні дані та приймає стохастичні рішення про те, передавати ці вхідні

дані чи ні. Коли дані подаються як вхідні дані, ці вузли самостійно вивчають усі параметри, їх шаблони та кореляцію між ними та утворюють ефективну систему. Тому машину Больцмана також називають моделлю неконтрольованого глибокого навчання.

Потім цю модель можна навчити моніторингу та визначення аномальної поведінки залежно від того, чому навчали мережу. Коефіцієнти, які змінюють вхідні дані, ініціалізуються випадковим чином. Кожен видимий вузол бере ознаку низького рівня з елемента, наявного в наборі даних - для вивчення. Результат цих двох операцій подається у функцію активації, яка, у свою чергу, створює вихідні дані вузла, також відомі як потужність сигналу, що проходить через нього. Вихідні дані першого прихованого шару будуть передані як вхідні дані до другого прихованого шару, а звідти через стільки створених прихованих шарів, поки вони не досягнуть остаточного шару класифікації. Для простих рухів вперед вузли функціонують як автокодер. Навчання зазвичай відбувається дуже повільно в машинах Больцмана де присутня велика кількість прихованих шарів, оскільки великим мережам може знадобитися багато часу, щоб наблизитися до рівнозначного розподілу, особливо коли ваги великі, а розподіл дуже мультимодальний.

Глибинна мережа прямого поширення

Як правило, ви виявите, що глибокі нейронні мережі працюють краще, ніж інші. Однак не завжди необхідно використовувати глибоку мережу. Вибір багато в чому буде залежати від поставленого перед вами завдання. Якщо ви працюєте з багатьма вхідними даними, такими як дані зображення, то використання глибинних мереж прямого поширення або згорткової нейронної мережі, швидше за все, дасть кращі результати, ніж проста мережа прямого поширення. Однак припустімо, що ваше завдання полягає в тому, щоб виконати якусь базову класифікацію з обмеженою кількістю вхідних даних. У такому випадку вам може бути краще використовувати просту мережу або навіть деревоподібний алгоритм. Отже, повертаючись до точки глибини, проста відповідь полягає в тому, що глибші мережі, як правило, забезпечують кращу продуктивність у більш складних

завданнях. Існує кілька гіпотез про те, чому глибокі мережі працюють краще, починаючи від ефективності й закінчуючи покращеною здатністю вивчати більш абстрактні уявлення.

Архітектура нейронної мережі містить функції активації всередині прихованих вузлів і вихідних вузлів. Функція активації приймає вхідне значення, що надходить у вузол, виконує перетворення, а потім передає результат до наступного набору нейронів.

Причина використання мереж прямого поширення з більш ніж одним прихованим шаром полягає в тому, що універсальна теорема про наближення не надає інформації про те, як вивчити таку мережу. Пов’язана проблема, яка ускладнює вивчення таких мереж, полягає в тому, що їх ширина може стати експоненціально великою. Цікаво, що теорему про універсальну апроксимацію можна також підтвердити для мереж прямого поширення з багатьма прихованими шарами та обмеженою кількістю прихованих нейронів для яких були розроблені алгоритми навчання. Отже, глибинні мережі прямого поширення використовуються замість звичайних з практичних міркувань зручності навчання.

Для ефективності навчання таких мереж, сьогодні використовують стохастичний градієнтний спуск. Стохастичний градієнтний спуск обчислює градієнт для набору випадково вибраних навчальних зразків і послідовно оновлює параметри. Це призводить до швидшого навчання. Недоліком є збільшення похибки. Однак для наборів даних із великою кількістю вибірок перевага у швидкості переважує цей недолік. Глибока мережа прямого поширення зображена на рис. 1.13.

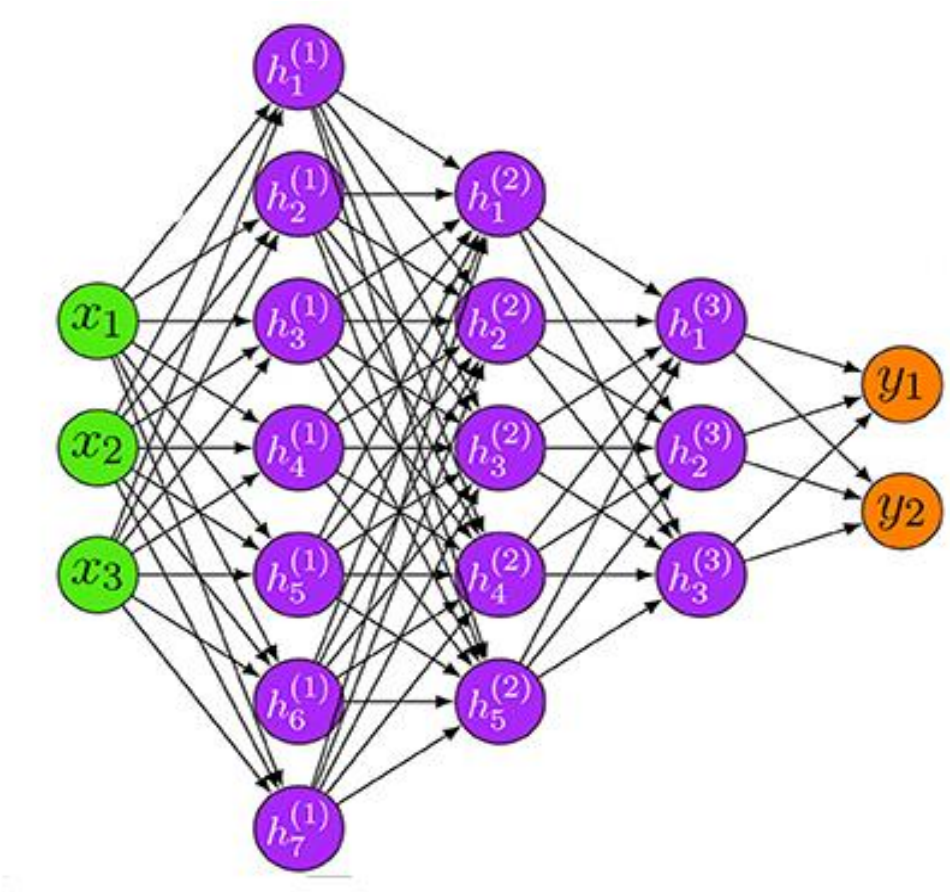

Рисунок 1.13 - Глибинна мережа прямого поширення

Згорткова нейронна мережа

Це спеціальна нейронна мережа прямого зв'язку, яка використовує згортку, алгоритм ReLU та рівні об'єднання. Стандартні згорткові мережі зазвичай складаються з кількох рівнів нейронної мережі прямого зв'язку, включаючи рівні згортки, об'єднання та повнозв'язні рівні.

Як правило, у традиційних ШНМ кожен нейрон на рівні з'єднаний з усіма нейронами на наступному рівні, тоді як кожне з'єднання є параметром у мережі. Це може призвести до дуже великої кількості параметрів. Замість використання повнозв'язних шарів, згорткова мережа використовує локальний зв'язок між нейронами, тобто нейрон з'єднаний лише з найближчими нейронами наступного шару. Це може значно зменшити загальну кількість параметрів у мережі.

Крім того, усі зв'язки між локальними полями та нейронами використовують набір ваг, позначений як ядро. Ядро буде спільно використовуватися з усіма іншими нейронами, які підключаються до їхніх локальних полів, і результати цих

обчислень між локальними рецептивними полями та нейронами, що використовують те саме ядро, зберігатимуться в матриці, позначеній як карта активації. Властивість спільного використання називається розподілом ваги. Отже, різні ядра призведуть до різних сценаїв активації, а кількість ядер можна регулювати за допомогою гіперпараметрів. Таким чином, незалежно від загальної кількості зв'язків між нейронами в мережі, загальна кількість ваг відповідає лише розміру локального рецептивного поля, тобто розміру ядра. Згорткова мережа представлена на рис. 1.14.

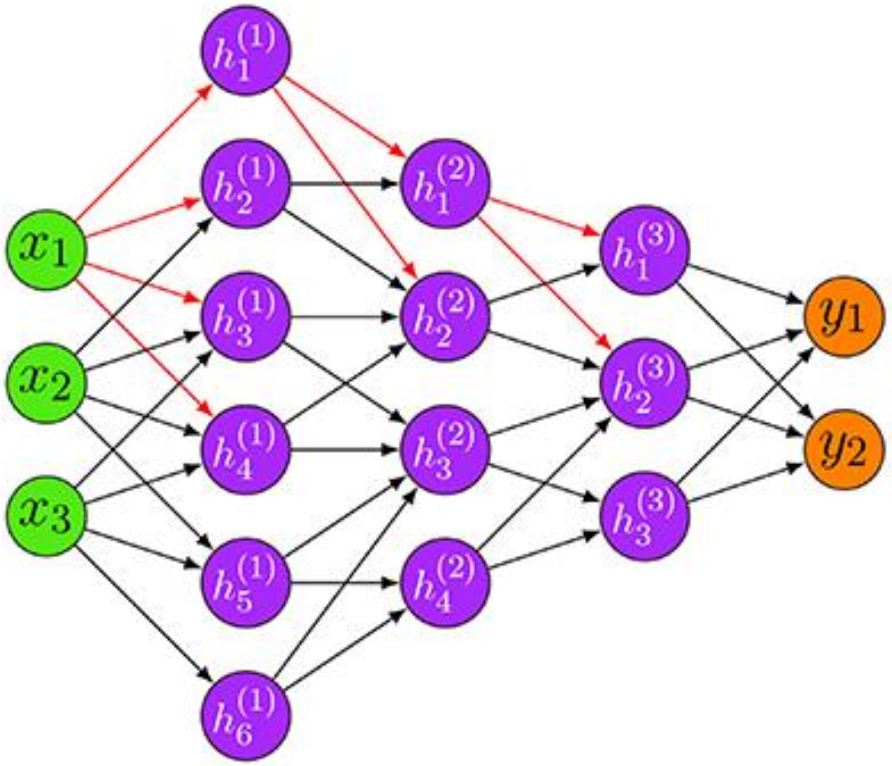

Рисунок 1.14 - Згорткова нейронна мережа

Поєднуючи розподіл ваги, мережа може обробляти дані з великими розмірами. На рис. 1.14 візуалізована мережа з трьома прихованими шарами. Червоні краї підкреслюють властивість локальності прихованих нейронів, тобто лише дуже мало нейронів підключаються до наступних шарів. Ця властивість локальності згортки робить мережу розрідженою порівняно з мережами прямого поширення, які є повнозв'язними.

Згортковий шар є важливою частиною побудови згорткової нейронної мережі. Подібно до прихованого рівня звичайної нейронної мережі, згортковий

рівень має ту саму мету, яка полягає в перетворенні вхідних даних у представлення більш абстрактного рівня. Однак замість використання повного зв’язку згортковий рівень використовує локальний зв’язок для виконання обчислень між вхідними та прихованими нейронами. Згортковий рівень використовує принаймні одне ядро для входу, виконуючи операцію згортки між кожною областю введення та ядром. Результати зберігаються в картах активації, які можна розглядати як вихід згорткового шару. Важливо, що карти активації можуть містити функції, отримані різними ядрами. Кожне ядро може діяти як екстрактор функцій і ділитиметься своїми вагами з усіма нейронами.

Глибинна мережа переконань

Це модель, яка поєднує різні типи нейронних мереж одна з одною, щоб сформувати нову модель нейронної мережі. Зокрема, така мережа інтегрує обмежені машини Больцмана із глибинними нейронними мережами прямого зв’язку. Мережа Больцмана утворює вхідний блок, тоді як глибинна мережа прямого зв’язку утворює вихідний блок. Часто мережі Больцмана накладають один на одного, декілька таких мереж використовуються послідовно. Це додає глибини мережі переконань. Через різний характер мереж вони використовуються як два різних типи алгоритмів навчання. Практично обмежені машини Больцмана використовуються для ініціалізації моделі без вчителя. Після цього для точного налаштування параметрів використовується метод з вчителем. На рис. 1.15 зображена глибинна мережа переконань.

Мережі глибоких переконань можуть замінити глибоку мережу прямого поширення або, за більш складних механізмів, навіть згорткові нейронні мережі. Вони мають переваги в тому, що вони менш дорогі з точки зору обчислень (їхня лінійна складність зростає зі збільшенням кількості шарів, а не експоненціально, як у нейронних мереж прямого поширення); і що вони значно менш вразливі до проблеми зникаючих градієнтів.

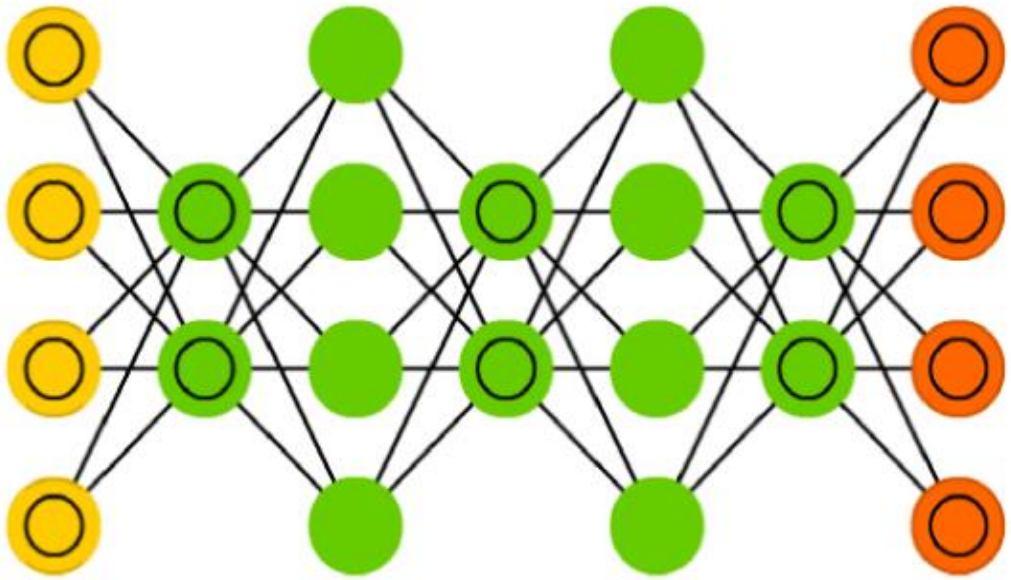

Рисунок 1.15 - Глибинна мережа переконань

Однак, оскільки мережі глибоких переконань накладають значні обмеження на їхні вагові зв’язки, вони також значно менш виразні, ніж глибокі нейронні мережі, які перевершують їх у завданнях, для яких доступно достатньо вхідних даних. Навіть на початку свого розквіту такі мережі рідко використовувалися. Натомість їх використовували як етап попереднього навчання: визначається та навчається мережа глибоких переконань із тією ж загальною архітектурою, що й відповідна глибинна нейронна мережа. Його ваги потім беруться та поміщаються у відповідну глибинну нейронну мережу для покращення результатів, яка потім налаштовується та використовується.

Мережі глибинних переконань згодом також втратили прихильність навчанні інших мереж. З одного боку, машина Больцмана є лише окремим випадком автокодерів, які виявилися більш гнучкими та корисними як для попереднього навчання, так і для інших програм. З іншого боку, запровадження активації ReLU та її подальше вдосконалення разом із запровадженням більш складних оптимізаторів, вивчення пізніх планувальників і методів відключення спрацювали, щоб значно полегшити проблему зникнення градієнта на практиці.

Автоенкодери (автокодер)

Автокодер - це модель нейронної мережі без вчителя, яка використовується

для навчання вибору функцій або зменшення розмірів. Загальною властивістю автокодерів є те, що розмір вхідного та вихідного рівня є однаковим із симетричною архітектурою. Основна ідея полягає в тому, щоб перетворити відображення вхідного шаблону x та у нове кодування c = h(x), яке в дає вихідний шаблон той самий, що й вхідний шаблон, тобто x ≈ y = g(c). Отже, кодування c, яке зазвичай має нижчу розмірність, ніж x, дозволяє відтворити (або закодувати) x.

Конструкція автокодерів подібна до глибинних мереж переконань. Цікаво, що оригінальна реалізація автокодера попередньо навчала лише першу половину мережі за допомогою обмежених машин Больцмана, а потім розгортала мережу, створюючи таким чином другу частину мережі. Подібно до глибинних мереж переконань, етап попереднього навчання супроводжується етапом тонкого налаштування. На рис. 1.16 показана ілюстрація процесу навчання. Тут рівень кодування відповідає новому кодуванню c, що забезпечує значно зменшений розмір x.

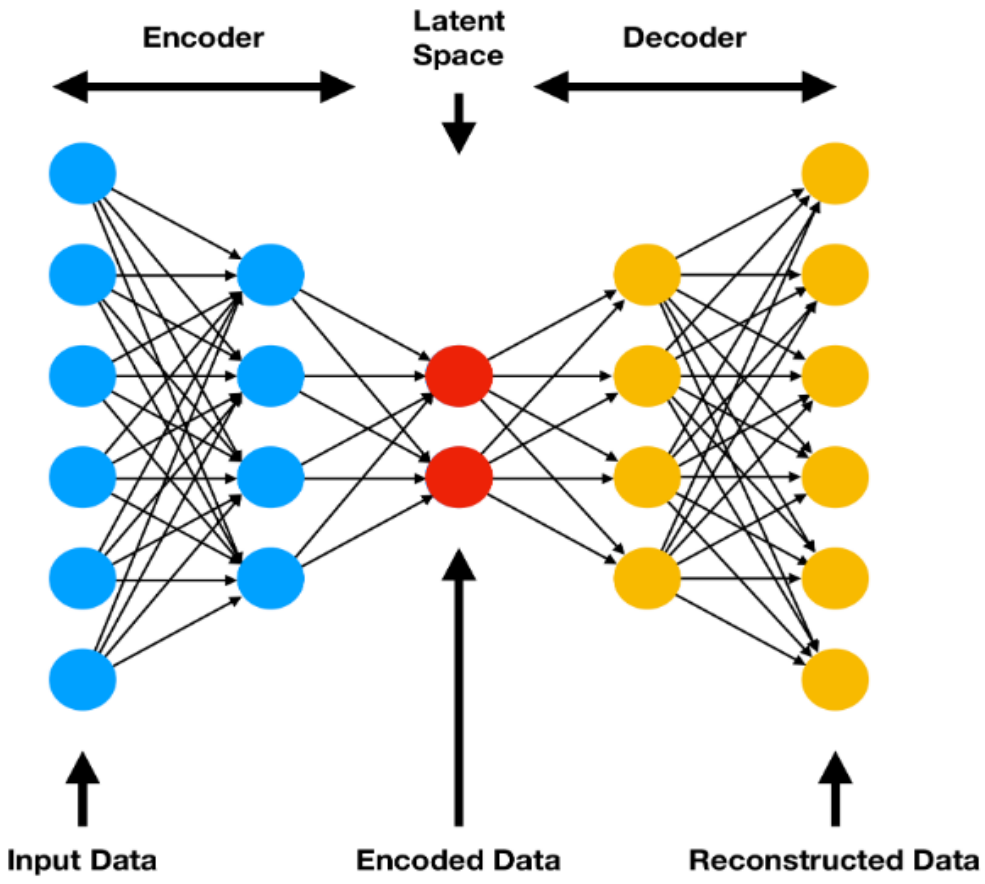

Рисунок 1.16 – Автоенкодер

У додатках модель успішно використовується для зменшення розмірності. Автокодувальники можуть досягти набагато кращого двовимірного представлення даних масиву, коли доступна достатня кількість даних. Зазвичай це призводить до кращої продуктивності. Важливо підкреслити, що існує багато розширень цих моделей, наприклад, розріджений автокодер, автокодер із зменшенням шумів або варіаційний автокодер.

Мережа довгої короткочасної пам’яті.

Були введені Хохрейтером і Шмідхубером у 1997 році. Такі мережі - це варіант рекурсивних нейронних мереж, який має здатність усувати недоліки рекурентних мереж, а саме - при обробці довгострокових залежностей. Крім того, дані мережі не мають проблеми зникнення або вибуху градієнта. У 1999 році була представлена мережа із захисним затвором, яка могла обнулити пам’ять комірки. Це покращило початкові алгоритми і стало стандартною структурою мереж довгої короткочасної пам’яті. На відміну від нейронних мереж прямого поширення, мережі довгої короткочасної пам’яті містять з’єднання зі зворотним зв’язком. Крім того, вони можуть обробляти не лише окремі точки даних, наприклад вектори чи масиви, але й послідовності даних. З цієї причини такі мережі особливо корисні для аналізу мовних або відеоданих. На рис. 1.17 зображена схема шматочку мережі.

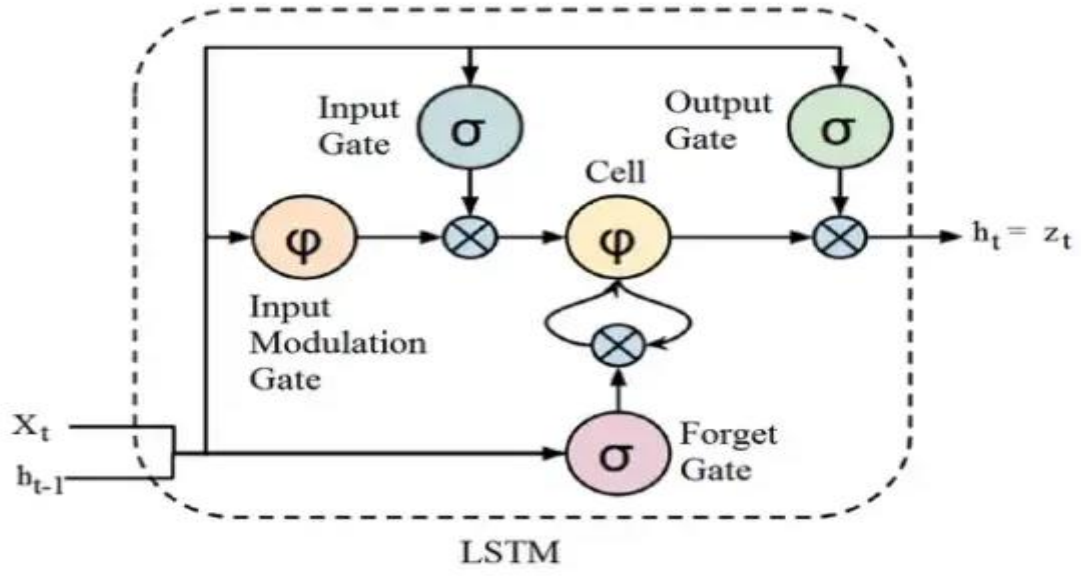

Рисунок 1.17 - Мережа довгої короткочасної пам’яті

1.5 Алгоритми для навчання ШНМ

Алгоритм оптимізації (або оптимізатор) здійснює процес навчання в нейронній мережі. Існує багато різних алгоритмів оптимізації. Вони відрізняються вимогами до пам’яті, швидкістю обробки та точністю чисел. В даній роботі мною були досліджені та проаналізовано декілька найпопулярніших алгоритмів для навчання нейронних мереж. Описані їх переваги та недоліки та визначені їх конкретні цілі. Визначена сама задача навчання.

Задача навчання - мінімізація індексу втрат f. Це функція, яка вимірює продуктивність нейронної мережі з роботою на наборі даних. Індекс втрат включає, загалом, помилку та умови регулярізації. Термін помилки оцінює, як нейронна мережа відповідає набору даних. Термін регулярізації запобігає перенавчанню, контролюючи складність моделі. Функція втрат залежить від адаптивних параметрів (зміщення та синаптичну вагу) у нейронній мережі. Їх можна згрупувати в один n-вимірний ваговий вектор w. На рис. 1.18 представлена функція втрат f(w).

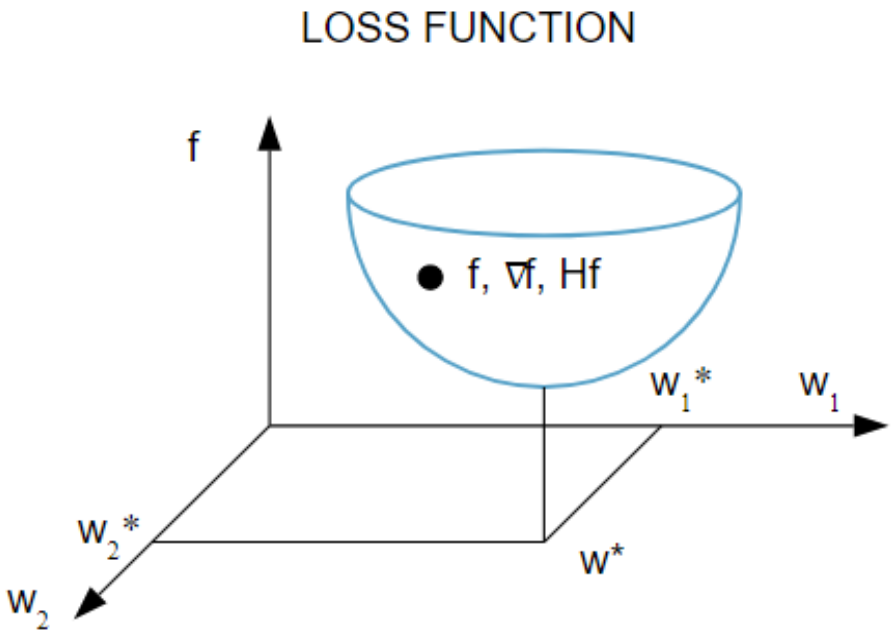

Рисунок 1.18 - Функція втрат

Мінімум втрат відбувається в точці w∗. У будь-якій точці A ми можемо

обчислити першу та другу похідні функції втрат.

Хоча функція втрат залежить від багатьох параметрів, велике значення тут мають одновимірні методи оптимізації. Дійсно, вони дуже часто використовуються в процесі навчання нейронної мережі. На рис. 1.19 зображено графічне представлення функції одновимірної оптимізації.

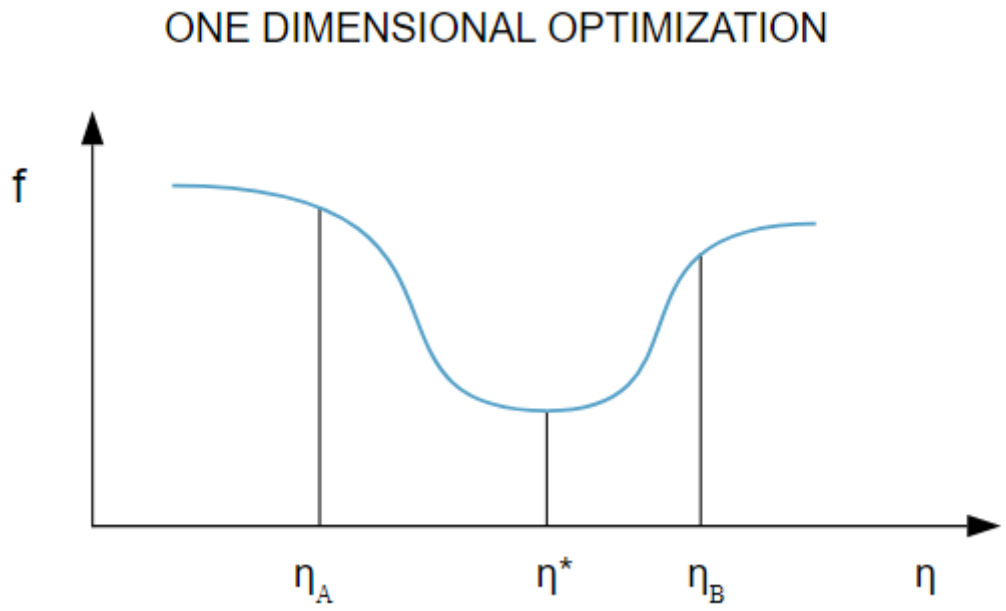

Рисунок 1.19 - Функція одновимірної оптимізації

Точки η1 і η2 визначають інтервал, який містить мінімум f, η∗. У зв’язку з цим одновимірні методи оптимізації здійснюють пошук мінімуму одновимірних функцій. Одними з найбільш використовуваних є золотий переріз і метод Брента. Обидва зменшують мінімальну дужку, доки відстань між зовнішніми точками не стане меншою за визначений допуск.

Функція втрат це нелінійна функція параметрів. Отже, неможливо знайти закриті алгоритми навчання для мінімумів. Замість цього ми розглядаємо пошук у просторі параметрів, що складається з послідовності кроків. На кожному кроці втрати будуть зменшуватися шляхом коригування параметрів нейронної мережі. Таким чином, щоб навчити нейронну мережу, ми починаємо з деякого вектора параметрів (часто вибирається навмання). Потім ми генеруємо послідовність параметрів, щоб функція втрат зменшувалася на кожній ітерації алгоритму.

Алгоритм навчання зупиняється, коли задовольняється задана умова або критерій зупинки.

Метод градієнтного спуску

Градієнтний спуск - найпростіший, але найпопулярніший алгоритм оптимізації. Він активно використовується в алгоритмах лінійної регресії та класифікації. Зворотне поширення в нейронних мережах також використовує алгоритм градієнтного спуску.

Градієнтний спуск — це алгоритм оптимізації першого порядку, який залежить від похідної першого порядку функції втрат. Він обчислює, у який бік слід змінити вагові коефіцієнти, щоб функція досягла мінімуму. Завдяки зворотному поширенню, втрати переносяться з одного шару на інший, а параметри моделі, також відомі як ваги, змінюються залежно від втрат, щоб втрати можна було мінімізувати. Алгоритм навчання градієнтного спуску має серйозний недолік, оскільки вимагає багатьох ітерацій для функцій, які мають довгі, вузькі долинні структури. Дійсно, низхідний градієнт - це напрямок, у якому функція втрат спадає найшвидше, але це не обов’язково призводить до найшвидшої конвергенції.

Переваги:

1. Легкість обчислень;
2. Простота впровадження;
3. Легкість у використанні.

Недоліки:

1. Може «зациклити» обчислення локального мінімуму;
2. Потрібен великий об’єм пам’яті для обчислення градієнта на всьому наборі даних;
3. При великій кількості вхідних даних і точок обчислення, розрахунки можуть іти роками.

Метод стохастичного градієнтного спуску

Це варіант градієнтного спуску. Він намагається частіше оновлювати параметри моделі. При цьому параметри моделі змінюються після обчислення втрат на кожному прикладі навчання. Отже, якщо набір даних містить 1000 рядків,

даний метод оновить параметри моделі 1000 разів за один цикл набору даних замість одного разу, як у простому градієнтному спуску. На рис. 1.20 зображене графічне представлення стохастичного спуску.

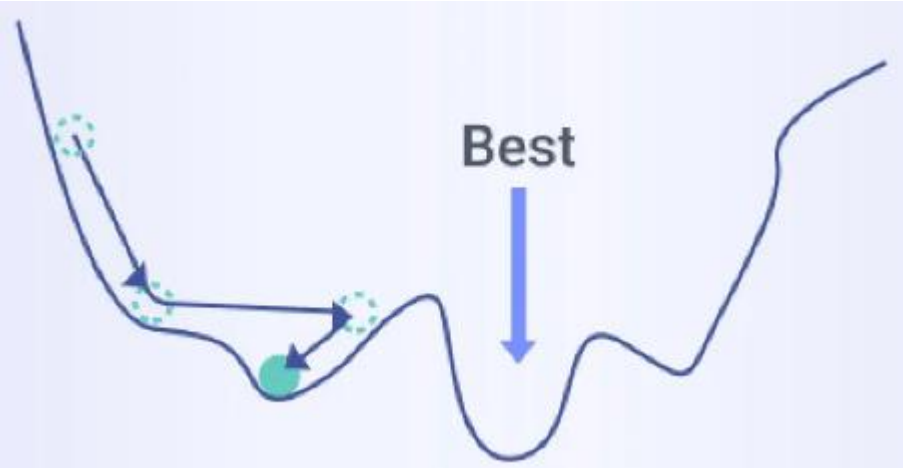

Рисунок 1.20 - Функція стохастичного спуску

Переваги:

1. Часте оновлення параметрів моделі, що дозволяє значно скоротити час;
2. Використовує менше пам’яті;
3. Може досягти нових мінімумів.

Недоліки:

1. Велика варіативність параметрів;
2. Може видавати певні значення навіть після досягнення мінімуму;
3. Для точніших результатів потрібно зменшити швидкість навчання

Метод Ньютона

Метод Ньютона є алгоритмом другого порядку, оскільки він використовує матрицю Гессе. Метою цього методу є пошук кращих напрямків навчання за допомогою других похідних функції втрат.

На рис. 1.21 зображений ілюстрований результат виконання даного методу, який вимагає менше кроків ніж градієнтний спуск, щоб знайти мінімальне значення функції втрат.

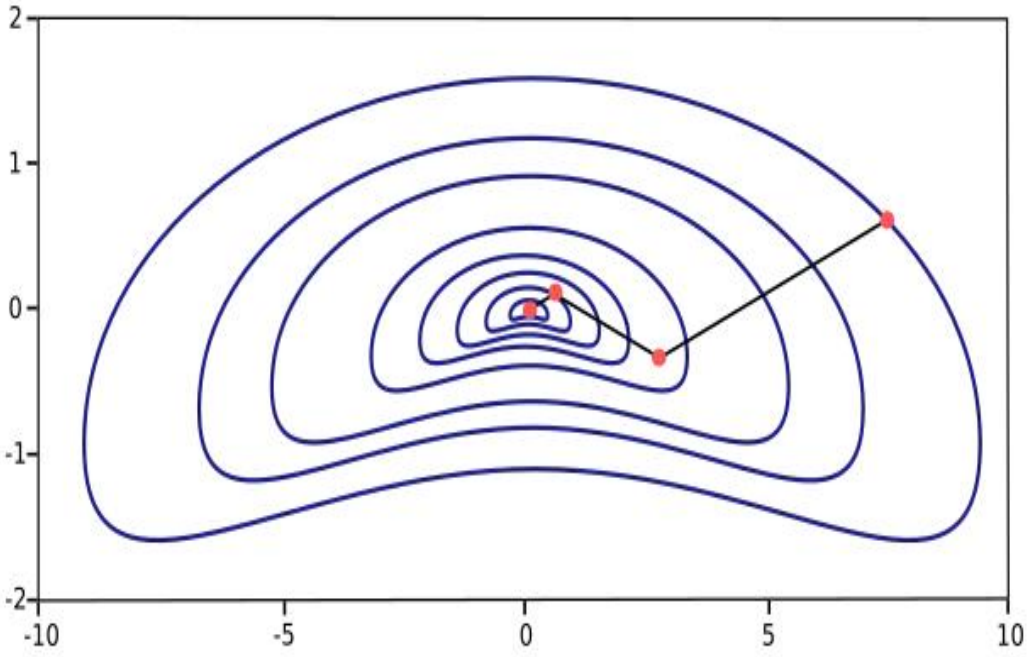

Рисунок 1.21 - Результат алгоритму Ньютона

З недоліків – алгоритм має слабке місце, зумовлене складністю обчислень та необхідним на це часом.

Прискорений градієнт Нестерова

Якщо імпульс занадто високий, алгоритм може пропустити локальні мінімуми та може продовжувати зростати. Отже, для вирішення цієї проблеми був розроблений алгоритм Нестерова. Це метод передбачення. Будемо використовувати основу метода імпульсу - $\gamma V(t-1)$ для модифікації ваг, тому $\theta - \gamma V(t-1)$ приблизно повідомляє нам про майбутнє розташування наступної точки. На рис. 1.22 зображене графічне представлення методу Нестерова.

Переваги:

1. Не пропускає локальний мінімум;
2. Сповільнюється якщо приближається до мінімума;

Недоліком можна вважати гіперпараметр, який необхідно виставляти власноруч.

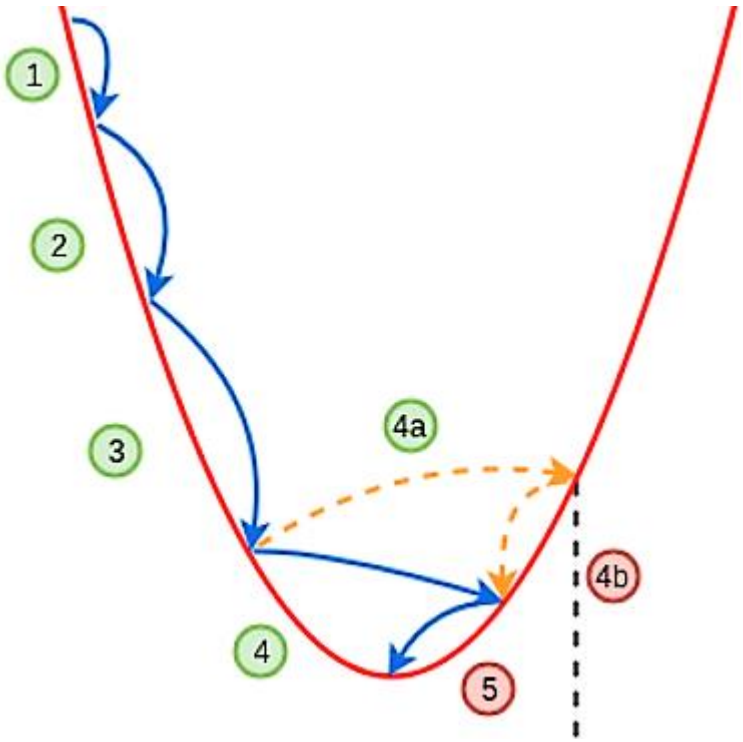

Рисунок 1.22 - Прискорений градієнт Нестерова

2 ДОСЛІДЖЕННЯ СЕРЕДОВИЩА ТА МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗРОБКИ НЕЙРОННИХ МЕРЕЖ

2.1 Порівняльна характеристика мов програмування

Дослідивши всі наявні мови програмування для створення штучного інтелекту та нейронних мереж, я виділив основні, які на мою думку найбільше відповідають сучасним тенденціям розвитку нейроінформаційних технологій. Хороша мова програмування має відповідати трьом простим параметрам: легкість у навчанні, читанні коду та розгортанні на різних платформах. Мова програмування має:

1. Мати усі необхідні засоби та інструменти для розробки штучних нейронних мереж та штучного інтелекту;
2. Мати відкриті бібліотеки та фреймворки;
3. Мати велику спільноту користувачів для підтримки;
4. Підтримувати усі нововведення у сфері штучних нейронних мереж;
5. Бути підкріплена документацією для користувачів.

Java - це надійна об’єктно-орієнтована мова програмування, яка пропонує простий синтаксис і легке впровадження. Java стала основною мовою для розробки мобільних додатків, яка значною мірою покладається на штучний інтелект. Java переважаю у розробці завдяки швидшому виконанню та швидшому часу навчання мереж, ніж Python, що робить цю мову штучного інтелекту ідеальним вибором для проектів машинного навчання, яким потрібна швидкість. Java також полегшує масштабування великих або складних програм штучного інтелекту.

Java має велику кількість бібліотек машинного навчання, таких як Weka, яка використовується для алгоритмів машинного навчання, моделювання різних прогнозів тощо, і Massive Online Analysis, програмним забезпеченням для аналізу даних з відкритим кодом. Багато популярних інструментів обробки великих даних, таких як Apache Hive, Apache Hadoop і Apache Spark, написані на Java, що забезпечує плавну інтеграцію з цими ключовими аналітичними фреймворками.

Технологія віртуальної машини Java також дозволяє розробникам писати та запускати послідовний код на всіх підтримуваних платформах і швидко виготовляти спеціальні інструменти. Java широко використовується для аналізу даних, глибокого навчання та обробки природної мови, навколо якої розквітла багата спільнота підтримки. Ключовим особливостями мови є:

- Об’єктно-орієнтованість;
- Її багатоплатформність;
- Має статичний синтаксис;
- Доступна велика стандартизована бібліотека;
- Підтримує декілька потоків та має механізм переривань.

Java використовується в багатьох критично важливих програмах, тому вона перевірена часом. Її можна використовувати як для комп’ютерних, так і для мобільних програм (за допомогою Android Studio). До недоліків можна віднести досить обмежену кількість бібліотек, і програмістам доводиться багато чого писати самостійно.

Julia

Ця динамічна мова програмування розроблена для досягнення успіху в числовому аналізі та обчислювальній науці. Розроблена Масачусетським технологічним інститутом у 2012 році, мова Julia є відносно новою, але її популярність зростає частково завдяки її швидкості, потужній обчислювальній здатності та синтаксису, схожому на скрипти. Навколо цієї мови з’явилася зростаюча спільнота підтримки, а також велика кількість бібліотек машинного навчання, таких як TensorFlow.jl, Scikitlearn.jl, Mocha.jl, Flux тощо.

Julia може безперешкодно перетворювати алгоритми з наукових статей у код, зменшуючи ризик моделі та підвищуючи безпеку. Крім того, Julia дає змогу інженерам машинного навчання оцінити модель і розгорнути її у виробництві, використовуючи ту саму мову. Julia - це найкраща мова програмування для додатків штучного інтелекту, які вимагають високопотужних обчислень, і ідеально підходить для штучного інтелекту, які вже мають досвід роботи з такими мовами, як Python або R. Ключовими особливостями даної мови вважаю:

- Вона створена для складних наукових обчислень;
- Досить легка в навчанні;
- Може використовувати бібліотеки C та Fortran;
- Підтримує паралельні обчислення;
- Базова версія іде з розширеними бібліотеками що включають в себе функції машинного навчання та оптимізації.

Окрім швидкості, Джулія також дуже гнучка (що дозволяє швидко експериментувати з різними моделями), а також має низку добре розроблених бібліотек машинного навчання, таких як Flux, MLJ і KNet. Недоліками є те, що мова відносно молода та не має ще такої спільноти як C++, що робить навчання трохи складнішим.

Haskell

Haskell - функціональна мова програмування, заснована на семантиці мови програмування Miranda. Перш за все, Haskell забезпечує безпеку та швидкість у контексті машинного навчання. Завдяки підтримці вбудованих спеціальних мов, життєво важливих для досліджень штучного інтелекту, Haskell знайшов собі місце в академічних колах, але такі технічні гіганти, як Microsoft і Facebook, спрямували Haskell на створення фреймворків, які оперують схематизованими даними та борються зі зловмисним програмним забезпеченням відповідно.

Бібліотека HLearn від Haskell пропонує алгоритмічні реалізації для машинного навчання, а її прив’язка до Tensorflow підтримує глибоке навчання. Haskell дозволяє користувачам кодувати високовиразні алгоритми без шкоди для продуктивності, і мова ідеально підходить для проектів, які включають абстрактну математику та ймовірнісне програмування. За допомогою Haskell користувачі можуть представити модель лише за допомогою кількох рядків коду і читати написані ними рядки як математичні рівняння. Таким чином, Haskell може влучно передати складність моделі глибокого навчання за допомогою чистого коду, який нагадує справжню математику моделі. До переваг можна віднести її функціонально – орієнтовану частину, що добре підходить для програмування штучних нейронних мереж. Недоліки – мова дуже складна для вивчення.

Lisp

Lisp це друга найстаріша мова програмування після Fortran, але вона залишається корисною для проектів, пов’язаних з машинним навчанням, завдяки своїй адаптивності, здатності до швидкого створення прототипів, автоматичному збиранню сміття, здатності створювати динамічні об’єкти та підтримці символічних виразів. Здатність LISP обробляти символьну інформацію створює мову для програмування штучного інтелекту, і вона чудово працює в контекстах, які включають обчислення за допомогою символів і символічних виразів. Символічний штучний інтелект є основним методом, який використовується для вирішення проблем, які вимагають логічного мислення та представлення знань. Згодом багато унікальних функцій LISP було використано в інших популярних мовах програмування, наприклад, розуміння списків Python і LINQ у C#. Переваги: дану мову використовують для розробки штучних мереж роками. Вона гнучка, логічна та має символьний характер. Недоліками є складність написання та читання коду. Також спільнота розробників дуже мала.

Scala

Ця мова програмування загального призначення підтримує як об’єктно-орієнтоване, так і функціональне програмування. Scala розроблена в 2004 році як більш стисла альтернатива, створена для усунення очевидних недоліків у дизайні Java. Вихідний код Scala був створений для роботи на віртуальній машині Java, тобто стеки Java і Scala можуть бути взаємозамінними. Scala підтримує багато бібліотек JVM, а також має доступні для читання функції синтаксису з іншими популярними мовами програмування. Оскільки додатки для Android часто пишуться на Java, сумісність Scala з Java робить цю мову корисною для розробки додатків для Android з інтенсивним використанням штучного інтелекту. Ця адаптивна мова є цікавим вибором для програмування нейронних мереж, оскільки вона може обробляти складні алгоритми та передавати дані в великому об’ємі. Scala є популярним вибором для взаємодії з механізмами обробки великих даних, такими як Apache Spark, який написаний на Scala.

Популярність Scala для створення моделей машинного навчання зростає

частково завдяки Spark і створеним на його основі бібліотекам машинного навчання Scala. Бібліотека Apache Spark MLlib та ML пропонує алгоритми кластеризації, класифікації та навчання з вчителем, а також модулі, які можна використовувати для створення конвеєрів даних. Високопродуктивна бібліотека BigDL інтегрується з Apache Spark як і Apache PredictionIO, який пропонує стек, для полегшення створення та розгортання алгоритмів машинного навчання. Основними перевагами є інтеграція з Java. Недоліки – потрібно витратити значну кількість часу на вивчення синтаксису та бібліотек.

Python

Python - це мова загального призначення з різноманітними модулями, починаючи від внутрішніх розробок та закінчуючи наукою про дані та машинним навчанням. Інтуїтивно зрозумілий синтаксис Python спрямований на зручність сприйняття, що спрощує кодування та полегшує вивчення мови. Завдяки своїй універсальності та простоті використання Python є чудовим вибором як для новачків, так і для досвідчених інженерів машинного навчання та спеціалістів із обробки даних, незалежно від досвіду програмування.

Python є однією з найпопулярніших мов програмування для нейромереж, завдяки широкому спектру перевірених, попередньо розроблених бібліотек, які оптимізують процес розробки нейронних мереж. Scikit-learn підтримує фундаментальні алгоритми машинного навчання, такі як класифікація та регресія, а Keras, Caffe та TensorFlow сприяють глибокому навчанню. Завдяки простій структурі та інструментам обробки тексту, таким як NTLK і SpaCy, Python є найкращим вибором мови програмування для обробки природної мови. Python також може має найповнішу документацією та підтримку спільноти, легко інтегрується з іншими мовами програмування. Ключові особливості:

- Дуже простий синтаксис;
- Підтримує купу модулів та бібліотек;
- Має найбільшу бібліотеку функцій серед усіх мов програмування.

Python має багатий набір бібліотек для аналізу та обробки даних, наприклад Pandas, що полегшує роботу з даними. Він має низку бібліотек спеціально для

машинного навчання, таких як TensorFlow і Keras, PyTourch. Він має надійні наукові та обчислювальні бібліотеки, такі як scikit-learn і NumPy. Його навіть можна використовувати для програмування мікроконтролерів із такими проектами, як MicroPython, CircuitPython і Raspberry Pi. З недоліків можна виділити повільну швидкість виконання програм.

Мова програмування R

R була розроблена групою статистиків для проведення статистичних обчислень. Оскільки R може з легкістю обробляти величезні набори даних, ця мова програмування широко використовується для розробки статистичного програмного забезпечення, аналізу та візуалізації даних.

Ця мова програмування є найкращим вибором для проектів машинного навчання, які включають масивний аналіз даних, і пропонує різноманітні методи навчання та оцінки моделі. R - потужна мова для машинного навчання, частково завдяки численним пакетам, зокрема CARAT для навчання класифікації та регресії, randomForest для створення дерев рішень тощо. Інтерактивне середовище R також ідеально підходить для швидкого створення прототипів і експериментування з новими проблемами. Хоча R не є мовою програмування для розгортання моделей машинного навчання, вона є найкращим інструментом для дослідницької роботи в процесі вибору моделі. Ті, хто має досвід роботи з Java або Python, швидко адаптуються до основного синтаксису R. Особливості мови R:

- Має інтегрований набір програмних засобів для обробки даних, обчислення та графічного відображення;
- Має ефективну обробку та зберігання даних, надаючи широкий спектр операторів для обчислень масивів, списків, векторів і матриць;
- Пропонує варіативність у питаннях аналізу даних та їх відображення;
- Мова включає в себе рекурентні функції;

З недоліків – наразі мова майже не підтримується, також програми працюють досить повільно.

C++

Ця високопродуктивна об’єктно-орієнтована мова програмування відома

високою швидкістю обробки, що дозволяє працювати складним моделям машинного навчання з високою ефективністю, частково завдяки компактному коду, який генерує C++. Оскільки C++ є статично типізованою мовою, помилки типу не з’являються під час виконання. C++ також чудово справляється з динамічним балансуванням навантаження, адаптивним кешуванням і керуванням пам’яттю, тому ця мова є найкращим вибором для створення масштабованих інфраструктур великих даних.

Багато бібліотек машинного та глибокого навчання написані мовою C++. Відомі бібліотеки C++ включають SHARK, яка підтримує контрольовані алгоритми навчання, такі як лінійна регресія, і MLPACK, яка пропонує розширювані алгоритми котрі користувачі можуть інтегрувати в масштабовані рішення. C++ зазвичай використовується в контексті ресурсомістких програм штучного інтнелекту, які вимагають швидкого виконання. C++ це компільована мова, тобто вона перетворюється безпосередньо в машинний код, який можна запускати на комп’ютері. C++ пропонує більше контролю над керуванням пам’яттю, ніж решта, але це також означає, що є більше помилок.

Проаналізувавши всі наявні мови програмування, найкращою є Python. Він відповідає таким критеріям:

1. Легкість навчання;
2. Чудова інтеграція;
3. Велика кількість супровідної документації;
4. Простота коду (легко читати і писати код);
5. Не залежить від платформи (може запускатись на будь-якій операційній системі);
6. Прекрасні інструменти візуалізації.

У табл. 2.1 наведена порівняльна характеристика найкращих мов програмування для штучних нейронних мереж.

Таблиця 2.1 – Порівняння найкращих мов програмування для штучних мереж

Параметр	Python	Java	JavaScript	C++
Код	короткий	довгий	гнучкий	довгий
Швидкодія	повільна	швидка	повільна	швидка
Тип даних	динамічний	статичний	динамічний	статичний
Складність	легко	середньо	середньо	важко
Бібліотеки	дуже багато	кілька	кілька	кілька

2.2 Аналіз та вибір інтегрованого середовища розробки

Інтегроване середовище розробки (IDE) відноситься до програмного забезпечення, яке пропонує програмістам широкі можливості розробки програмного забезпечення. IDE найчастіше складаються з редактора вихідного коду, засобів автоматизації збірки та налагоджувача. Більшість сучасних IDE мають інтелектуальне завершення коду. В даній роботі, мною були проаналізовані усі доступні IDE для мови програмування Python.

Основні вимоги до середовища розробки такі:

1. Корекція синтаксу, що включає в себе достатньо велику базу мов програмування з можливістю їх підсвічування;
2. Автозавершення коду, що друкується;
3. Компіляція коду і створення файлу;
4. Налагодження коду;
5. Контроль версій середовища.

Wing

Це середовище, розроблений компанією Wingware спеціально для мови програмування Python. Wing пропонує нам безліч функцій, таких як автозаповнення, автоматичне редагування, браузер вихідного коду, перегляд коду, а також локальне та віддалене налагодження, щоб ми могли розробляти наші програми. У безкоштовних версіях ми не знайдемо всіх опцій, хоча там їх багато. Це інтегроване середовище розробки було розроблене, щоб скоротити час розробки

та налагодження. Воно забезпечує хорошу допомогу в кодуванні чи пошуку помилок. Полегшує навігацію та розуміння коду Python. Редактор Wing прискорює розробку на мові Python, надаючи автозаповнення та контекстно-залежну документацію. Це також дозволить нам мати автоматичне згортання коду, множинний вибір, закладки та багато іншого. Wing може емулювати vi, emacs, Eclipse, Visual Studio і Xcode. Приклад інтерфейсу зображений на рис. 2.1.

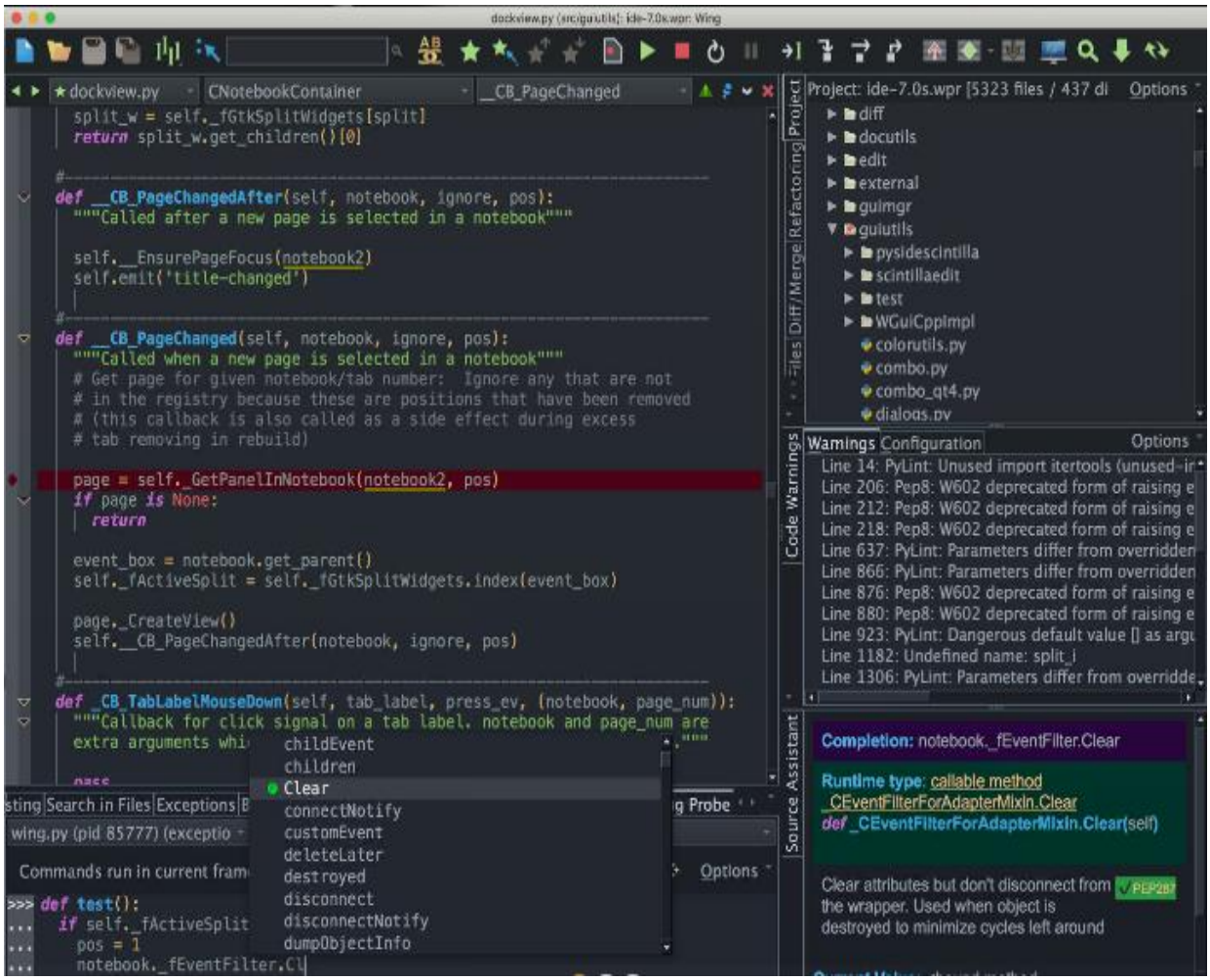

Рисунок 2.1 – Інтерфейс програми Wing

RStudio

Це безкоштовне програмне забезпечення з відкритим вихідним кодом, яке використовується як інтегроване середовище розробки. Дане IDE підтримується усіма операційними системами. RStudio гнучка та багатофункціональна і широко використовується як графічний інтерфейс для роботи з R. Крім того, адаптована до багатьох інших мов програмування, таких як Python або SQL. Інтерфейс зображений на рис. 2.2.

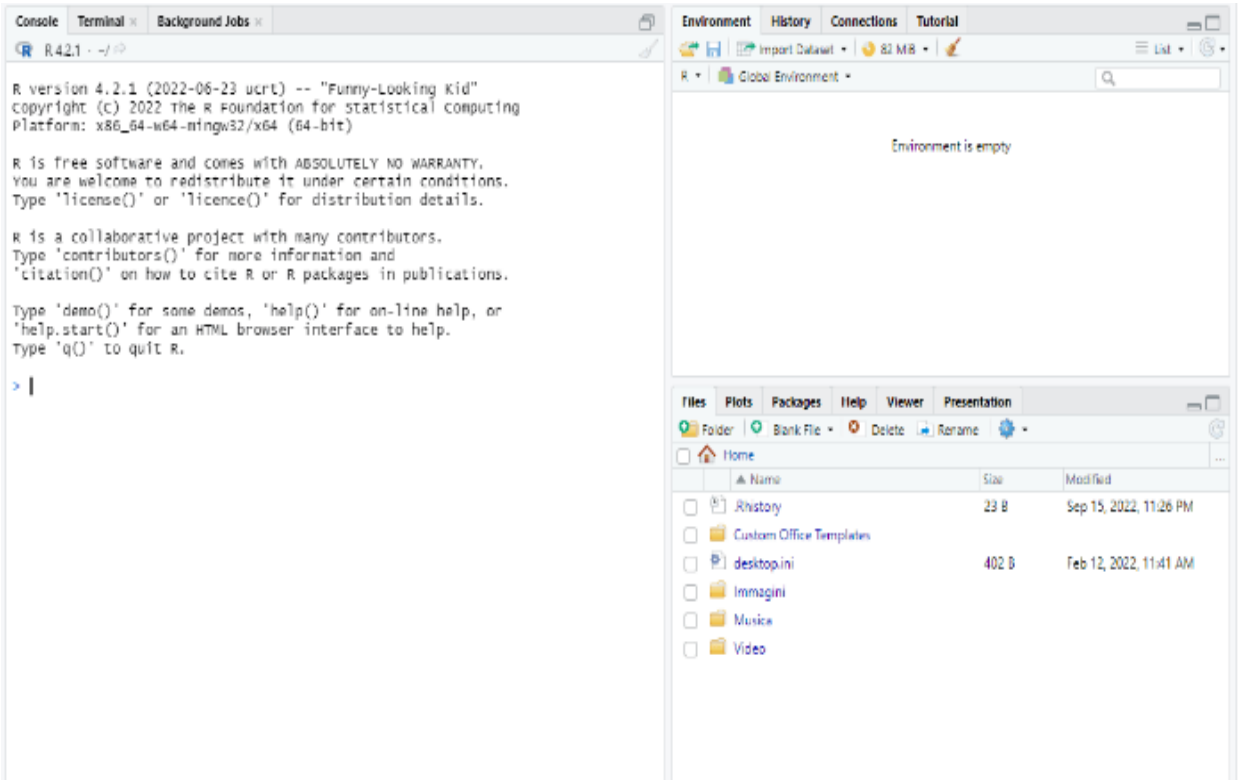

Рисунок 2.2 - Інтерфейс RStudio

Emacs

Emacs є одним із найстаріших доступних текстових редакторів, але відомий своєю динамічністю та потужними утилітами редагування. До появи Emacs текстові редактори були надзвичайно простими, і їх можна було використовувати лише для показу та створення слів або рядків із файлу. З документами, які були досить великими та містили багато даних, було важко працювати, і редагування займало багато часу. Приблизно в цей час з’явився Emacs і повністю змінив світ редагування тексту. Оскільки Emacs є багатоплатформною програмою з відкритим кодом, не дивно, що Emacs здобув таку величезну популярність у спільноті програмістів. Emacs виглядає як гібрид між vi та nano, що містить сильні сторони обох текстових редакторів. Однак, на відміну від цих двох, Emacs також пропонує версію графічного інтерфейсу (GUI), яка забезпечує дуже зручний інтерфейс для користувачів, який можна налаштувати. Emacs містить підручник, який дозволяє користувачам випробувати інтерфейс користувача та вивчити деякі його команди. Emacs вважається дещо сучасною IDE, оскільки вона надає користувачам функції, подібні до тих, які можна знайти в інших IDE. Вони можуть бути такими простими,

як підсвічування синтаксису або інтеграція з системами контролю версій. Візьмемо останній як приклад, скажімо, якщо хтось зацікавлений у налаштуванні своїх файлів і редагуванні їх у контролі версій, Emacs легко дозволить вам це зробити за допомогою плагінів. На рис. 2.3 зображений інтерфейс Emacs.

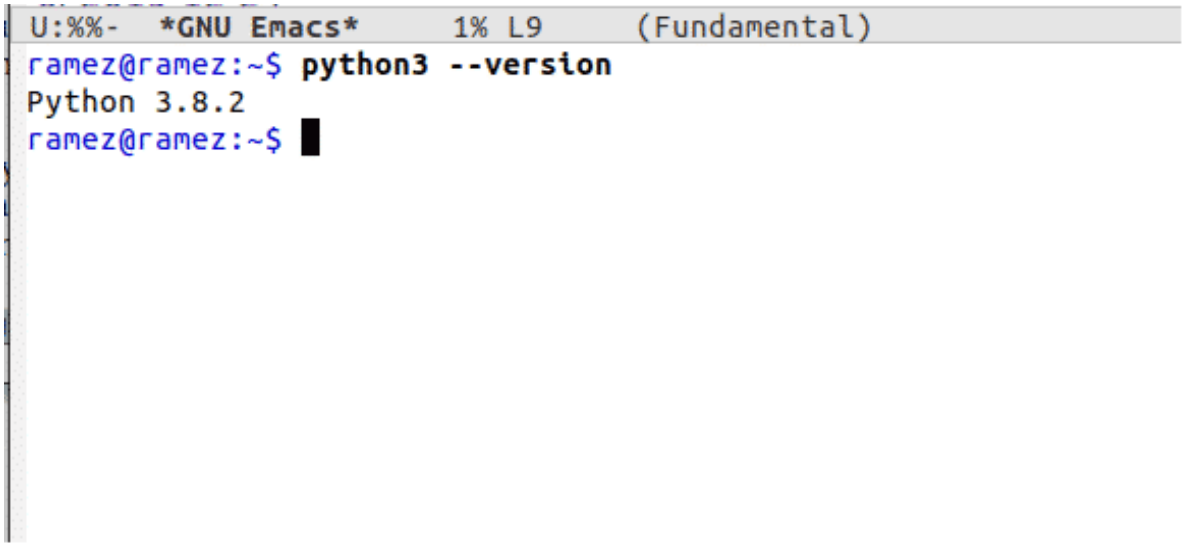

Рисунок 2.3 - Інтерфейс Emacs

Має значний недолік – якщо треба включити розширення, його доведеться писати самому.

Google Colab

Google Colab - це середовище на основі Jupyter, яке повністю працює в хмарі. Він обробляє всі налаштування та конфігурації, необхідні для вашої програми. Colab дозволяє будь-кому писати та виконувати довільний код Python через браузер і особливо добре підходить для машинного навчання, аналізу даних. Технічно кажучи, Colab - це розміщена служба блокнотів Jupyter, яка не потребує налаштування, але забезпечує безкоштовний доступ до обчислювальних ресурсів, включаючи графічні процесори. Ресурси Colab не гарантовані та не необмежені, тож об’єми іноді змінюються. Це необхідно для того, щоб Colab міг надавати ресурси безкоштовно. Ресурси в Colab мають пріоритет для інтерактивних випадків використання. У Colab можна програмувати на мові Python з підтримкою всіх фреймворків, імпортувати/зберігати файли на ГуглДиск, інтегрувати програму з PyTorch та Tensor Flow. На рис. 2.4 зображено інтерфейс Colab.

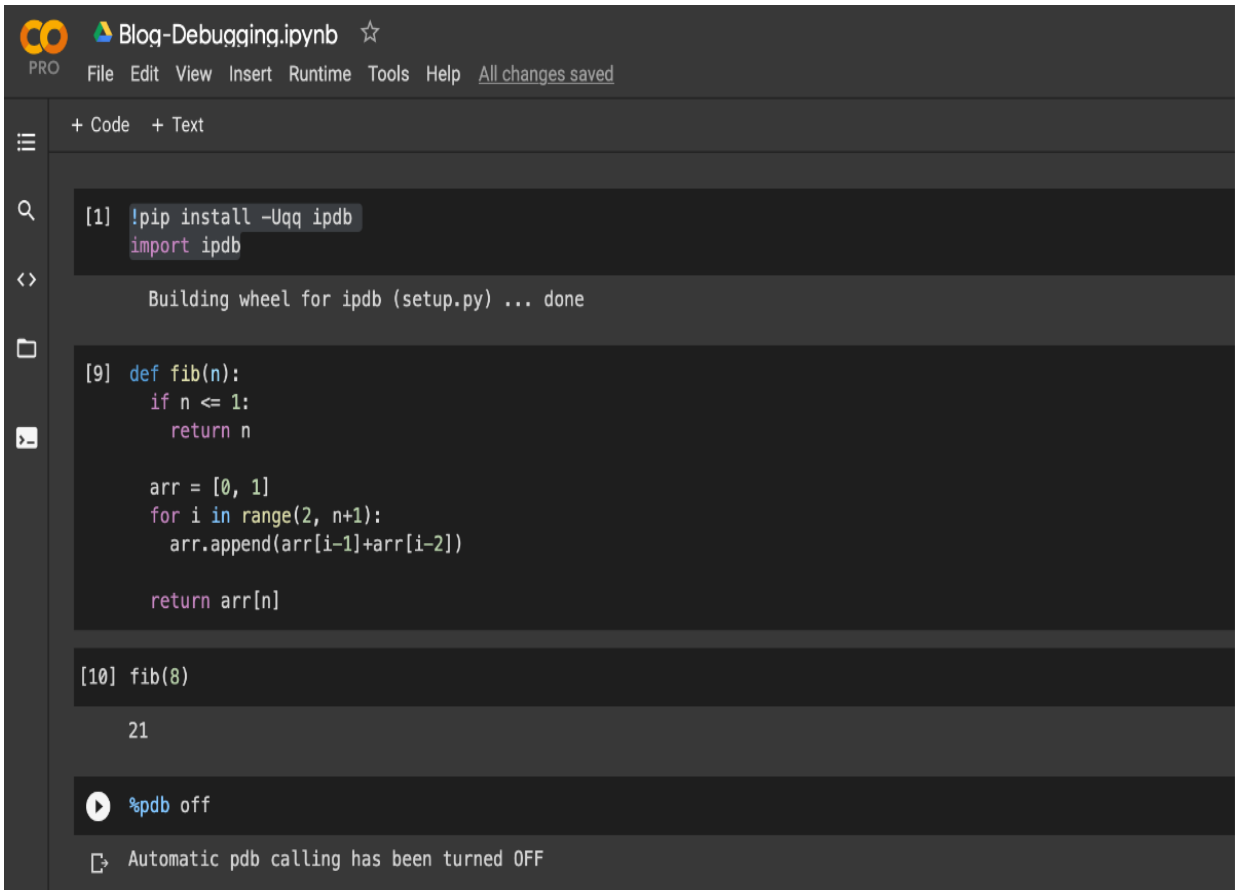

Рисунок 2.4 - Інтерфейс Colab

Thonny

Thonny — це спеціальне середовище розробки Python, яке постачається з вбудованим Python 3. Після встановлення ви можете почати писати код Python. Thonny призначений для новачків. Інтерфейс користувача зберігається простим, тому новачкам буде легко розпочати роботу. Хоча Thonny призначений для початківців, він має кілька корисних функцій, які також роблять його хорошим IDE для повноцінної розробки Python. Деякі з його функцій – підсвічування синтаксичних помилок, налагоджувач, завершення коду, покрокова оцінка виразу тощо.

Переваги: відмінно підходить для початківців.

Недоліки: недостатній функціонал так як дане середовище досить застаріле.

Приклад інтерфейсу зображений на рис. 2.5.

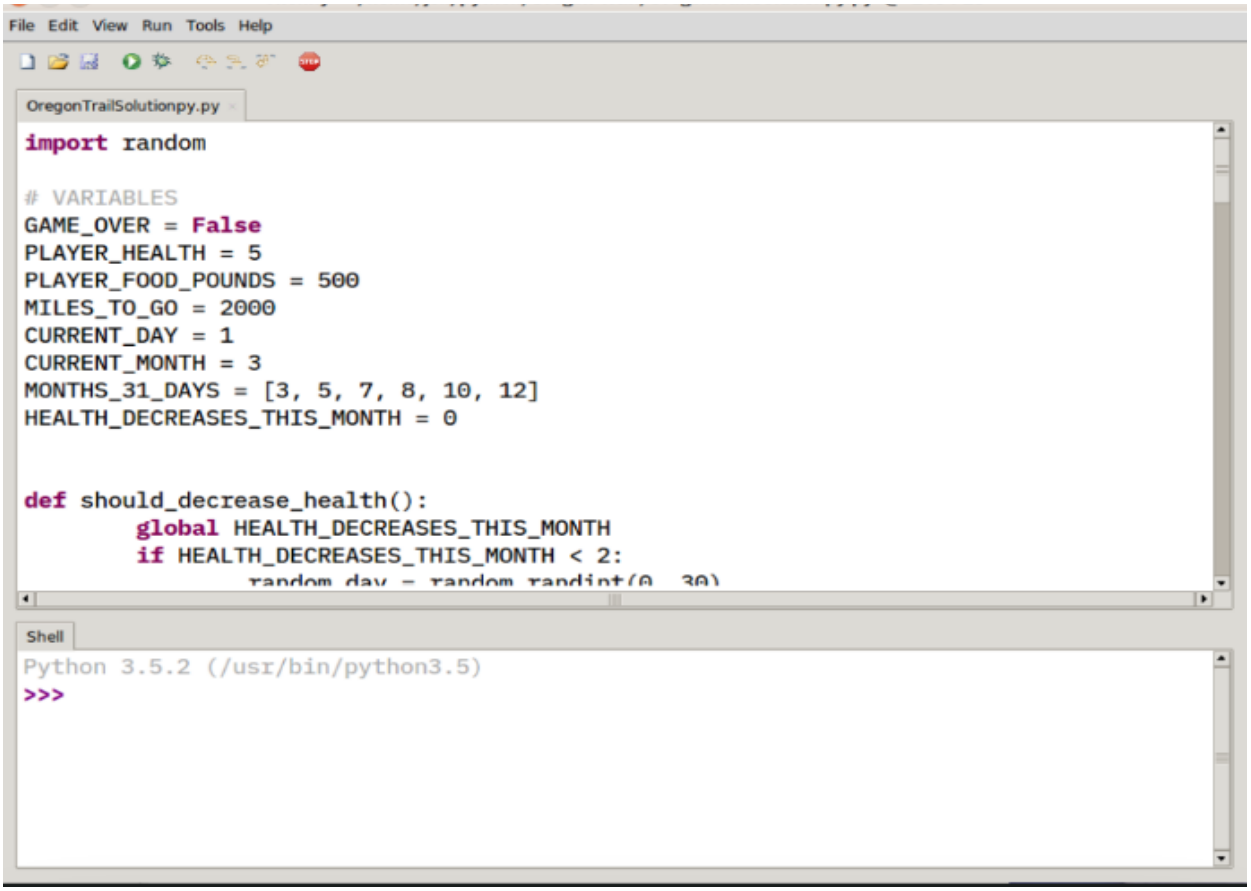

Рисунок 2.5 - Інтерфейс Thonny

PyDev + Eclipse

Eclipse - IDE з відкритим кодом для розробки на Java. Існує безліч розширень і доповнень, які роблять Eclipse корисним для різного роду завдань. Одним із таких розширень є PyDev, що надає інтерактивну консоль Python і можливості для відладки та автодоповнення коду. Із переваг даного середовища - там присутні усі відомі мови програмування. Недоліки – дуже складна для новачків і недосвідчених користувачів. Інтерфейс зображений на рис. 2.6. Eclipse, написаний на Java, тому працює на всіх платформах, виключення - комп’ютерна бібліотека SWT, яка розробляється для всіх розширених платформ. Бібліотека SWT використовується замість стандартної для бібліотеки Java Swing. Вона повністю спирається на операційну систему, що забезпечує швидкість і природний зовнішній вигляд інтерфейсу користувача, але іноді викликає проблеми сумісності та стійкості програм на різних платформах.

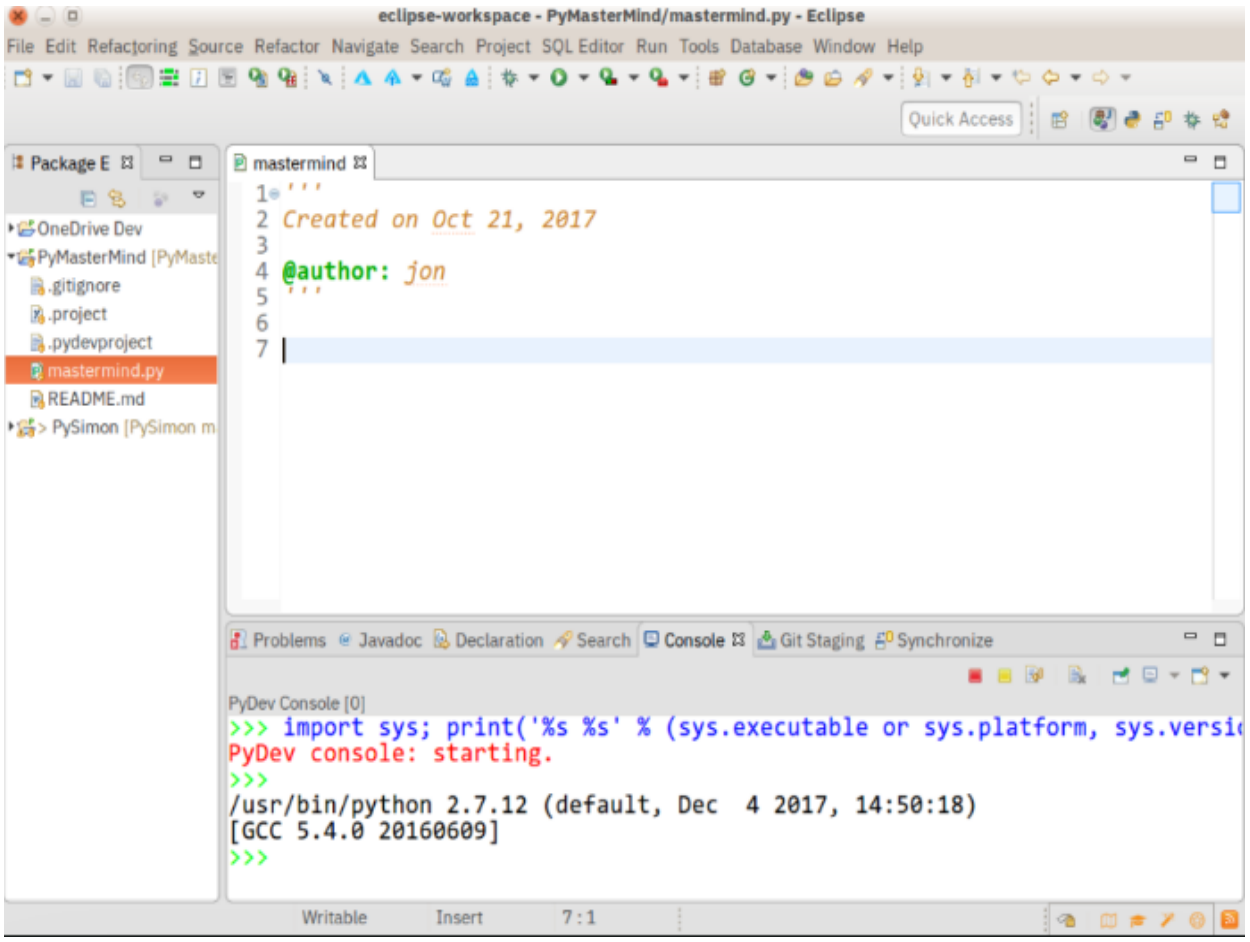

Рисунок 2.6 - Інтерфейс PyDev + Eclipse

Spyder

Spyder - це IDE з відкритим кодом, який зазвичай використовується для наукових розробок. Найпростіший спосіб налагодити роботу зі Spyder - встановити дистрибутив Anaconda. Anaconda - популярний дистрибутив для машинного навчання. Дистрибутив Anaconda включає сотні пакетів, включаючи NumPy, Pandas, scikit-learn, matplotlib тощо. Spyder має кілька чудових функцій, таких як автозавершення, налагодження та оболонка iPython. Однак йому бракує функцій порівняно з PyCharm. Spyder має ту функціональність, яку можна очікувати від стандартної IDE, як редактор коду з підсвічуванням синтаксису, автодоповнення коду і навіть вбудованого оглядача документації. Він дозволяє переглянути значення змінних у вигляді таблиці прямо всередині IDE. Також добре працює інтеграція з IPython/Jupyter. Його можна як інструмент для певної мети, а чи не як основне середовище розробки. Інтерфейс наведений у рис. 2.7.

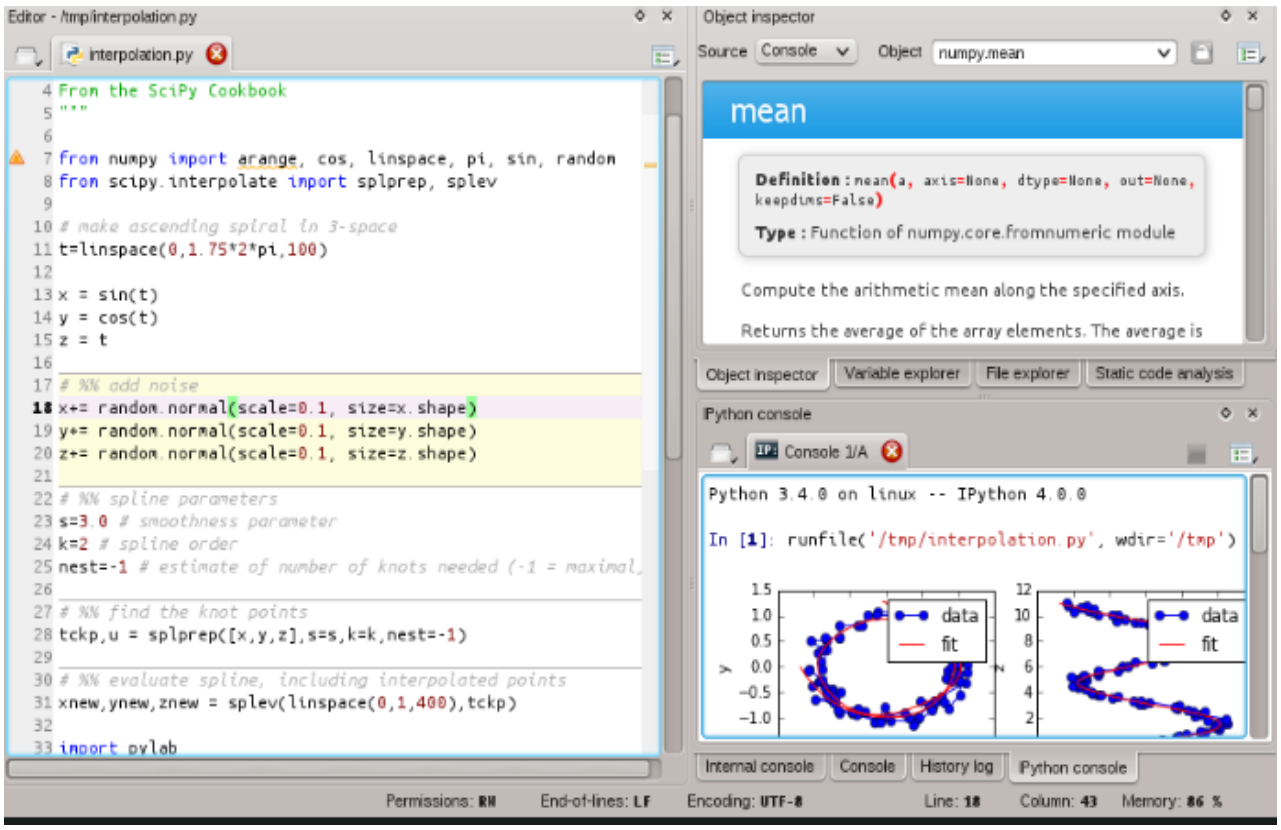

Рисунок 2.7 - Інтерфейс Spyder

Jupyter

Блокнот Jupyter - це безкоштовне середовище із відкритим кодом, яке використовується для створення документів Jupyter. Унікально, що це веб-інтерфейс IDE, який дозволяє вам легко ділитися своєю роботою. Jupyter Notebook дуже популярний серед дослідників даних, оскільки він чудово підходить для машинного навчання, моделювання, візуалізації даних, аналізу даних, а також для обміну та представлення результатів. Jupyter готовий до роботи прямо з коробки, майже не потребуючи налаштування. Це робить його ідеальним інструментом для швидкого створення сценаріїв Python.

Переваги: легкість навчання, підтримка більш ніж 40 мов програмування, можна легко поділитися результатом.

Недоліки: нестача сучасних функцій та розширень, в більшості не поширена серед звичайних користувачів.

Інтерфейс програми наведений на рис. 2.8.

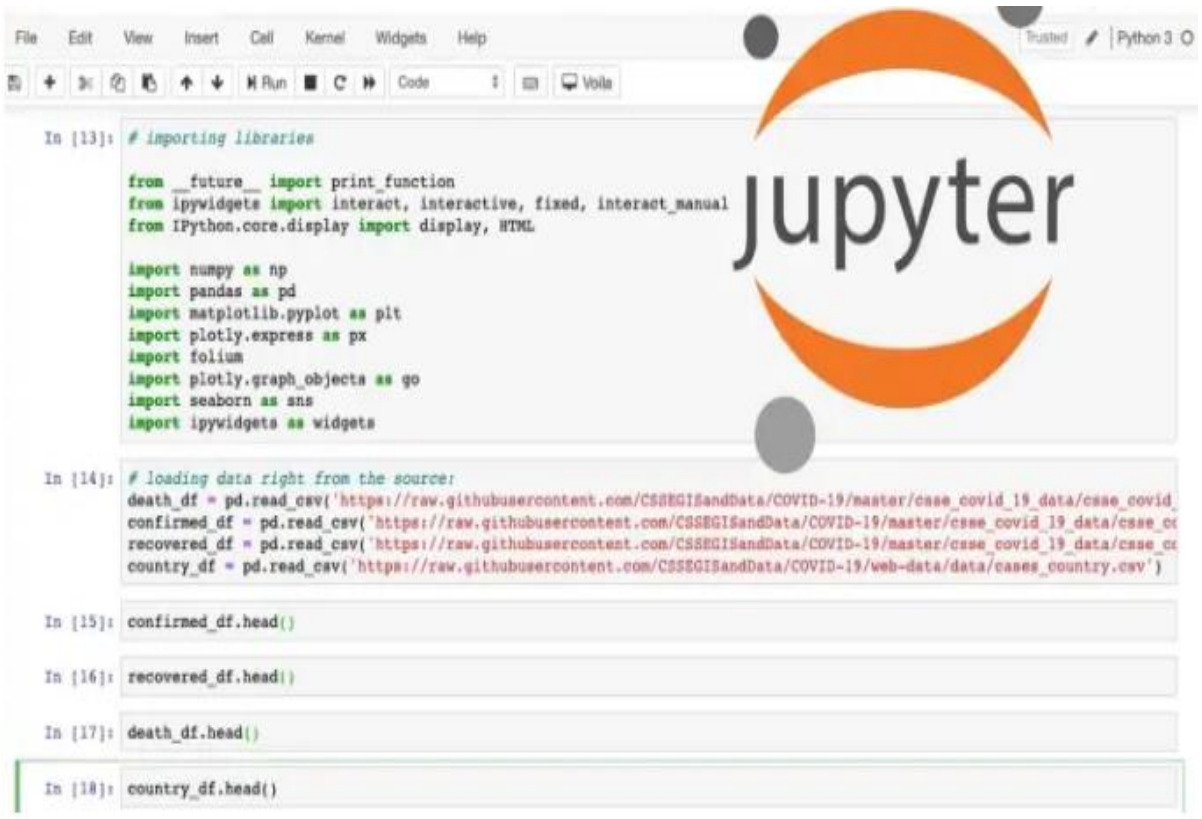

Рисунок 2.8 - Інтерфейс Jupyter

Sublime Text

Sublime Text, написаний інженером з Google, є дуже популярним редактором коду. Доступний на всіх платформах, Sublime Text має вбудовану підтримку редагування Python-коду, а також багатий набір розширень, які називаються пакетами, що надають більше можливостей синтаксису та редагуванню. Встановити додатковий Python-пакет може бути непросто - усі пакети Sublime Text написані на Python, тому для встановлення пакетів спільноти часто може знадобитися виконати Python-скрипт безпосередньо в редакторі.

Переваги: швидкість роботи, гарна підтримка спільноти.

Недоліки: для розширень необхідно писати скрипти, середовище розробки є платним.

Інтерфейс зображений на рис. 2.9.

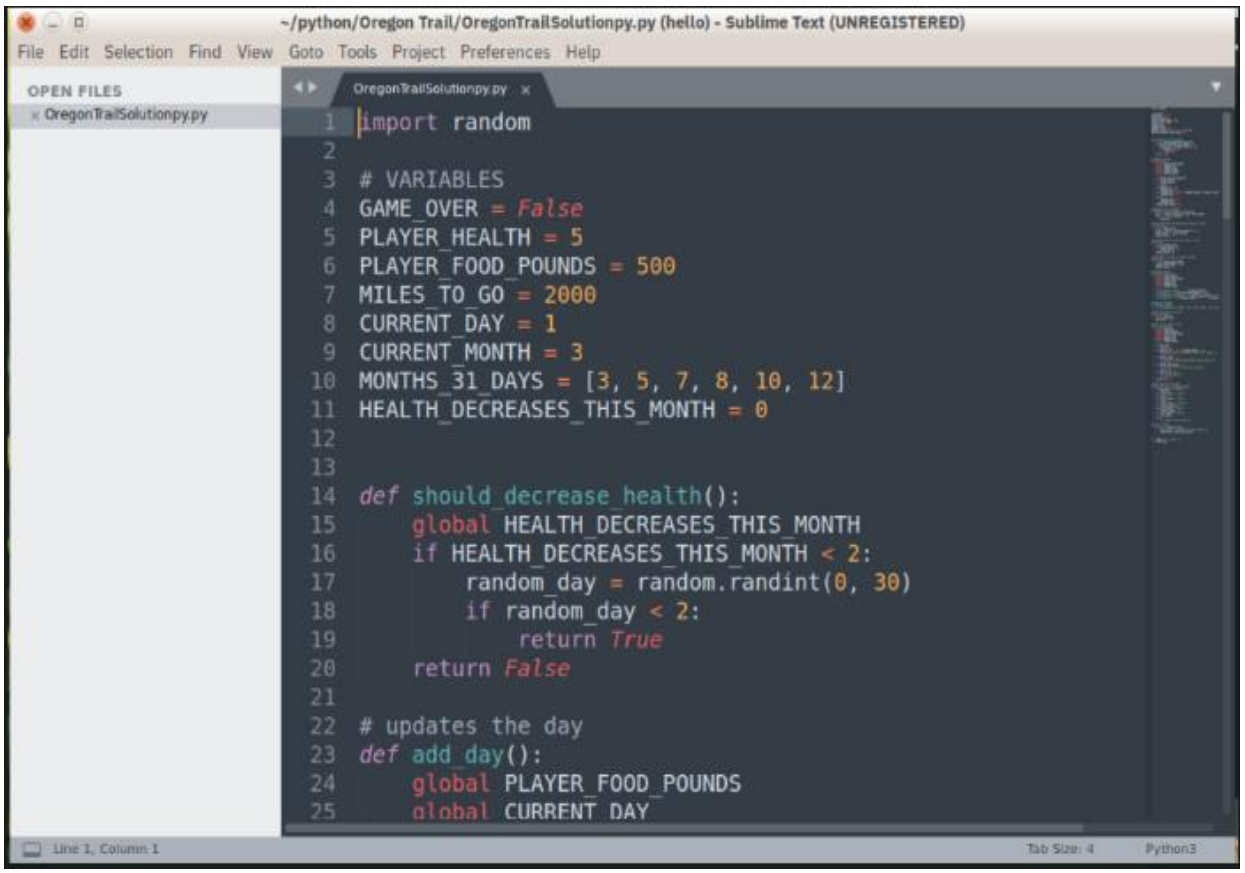

Рисунок 2.9 - Інтерфейс Sublime text

PyCharm

Є однією з найкращих повнофункціональних IDE, призначених саме для Python. Існує як безкоштовний open-source (Community), і платний (Professional) варіанти IDE. PyCharm доступний на Windows, Mac OS X та Linux. PyCharm одразу підтримує розробку на Python - відкрийте новий файл і починайте писати код. Можна запускати та редагувати код прямо з PyCharm. Крім того, у IDE є підтримка проектів та системи управління версіями.

Переваги: підтримка всіх існуючих додатків та розширень, найбільша спільнота користувачів.

Недоліки: повільне завантаження, високі системні вимоги до комп’ютера.

Інтерфейс зображений на рис. 2.10.

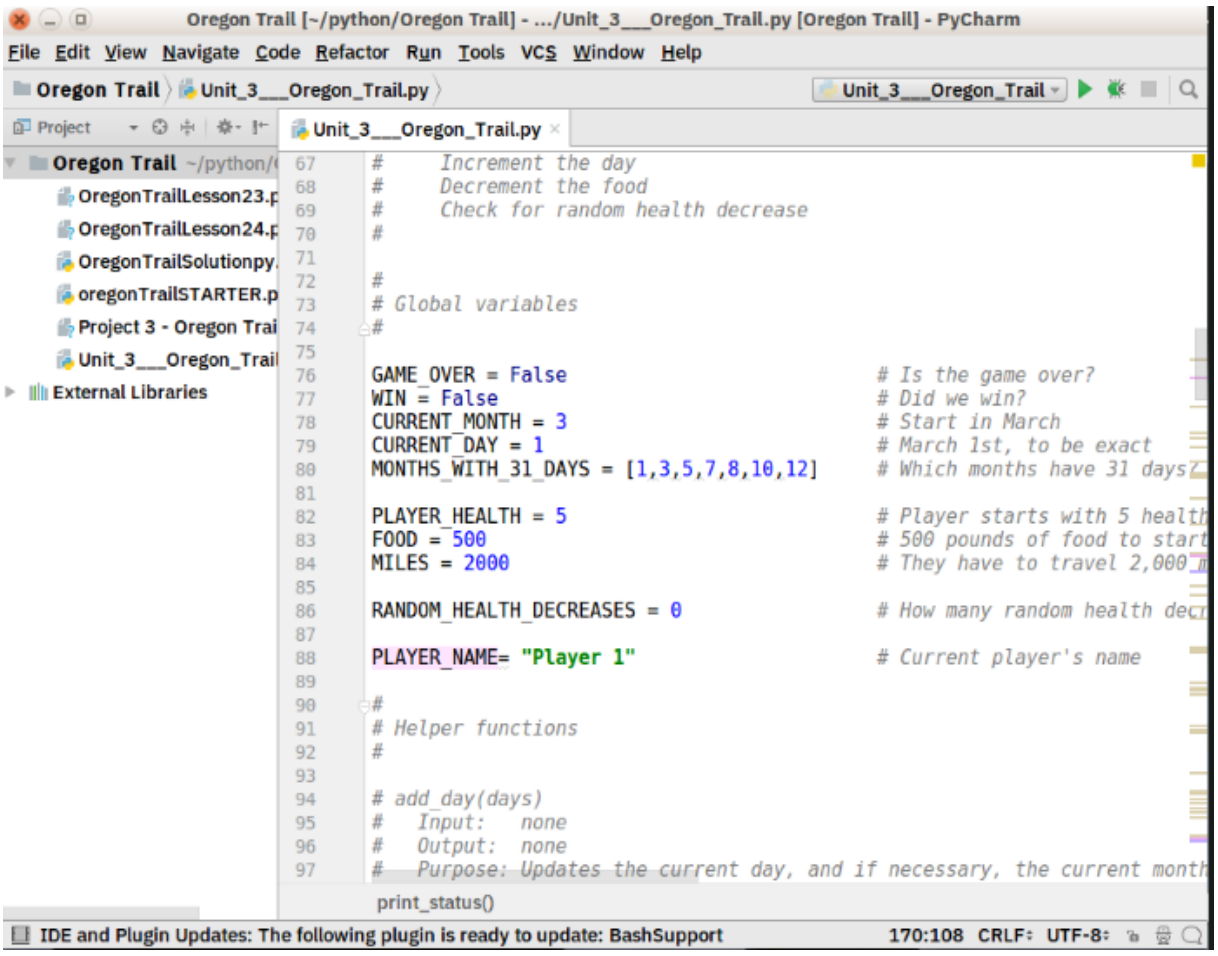

Рисунок 2.10 - Інтерфейс PyCharm

Atom

Atom - це редактор коду з відкритим вихідним кодом, розроблений Github, який можна використовувати для розробки на Python (подібний до Sublime text). Його функції також схожі на Sublime Text. Atom можна налаштувати відповідно до потреб. Деякі з пакетів, які часто використовуються в Atom для розробки на Python, це autocomplete-python, linter-flake8, python-debugger тощо.

Переваги: може запускатись на будь-якому пристрої. Завантажується швидко завдяки архітектурі Electron.

Недоліки: підтримка зборки і відладки не вбудована, потрібно додавати за допомогою спеціальних розширень. Так як Atom написаний на Java, він працює не як додаток, а як процес. На рис. 2.11 зображений інтерфейс програми Atom.

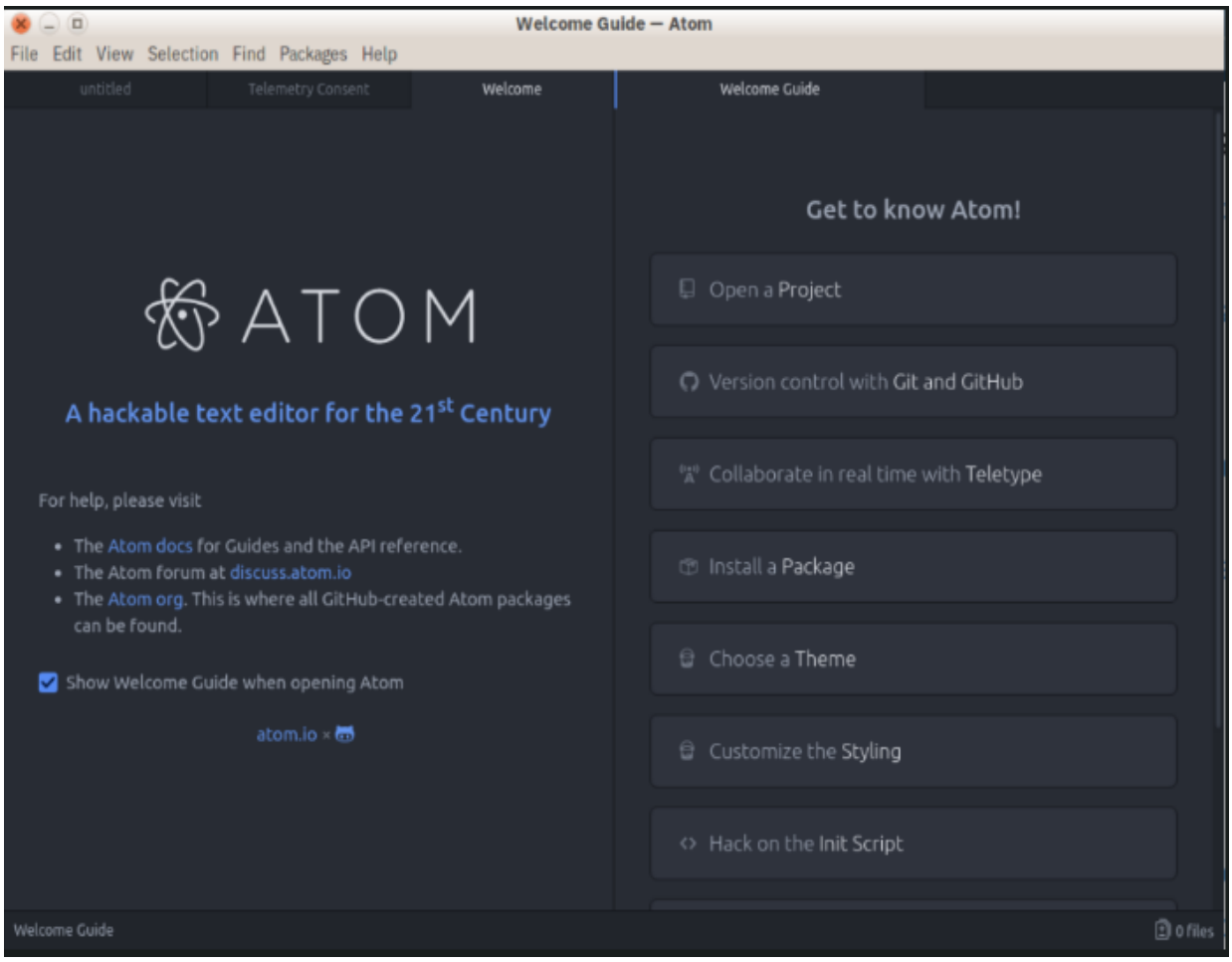

Рисунок 2. 11 - Інтерфейс Atom

Дослідивши найкращі інтегровані середовища розробки для мови програмування Python, мною було обрано Google Colab. Так як даний варіант є безплатним, надає досить великі потужності для обчислень, а також багато місця на віртуальному диску.

Доступ до Google Colab надається по Google акаунту, можна зайти з будь-якого пристрою та розробляти програми для складних обчислень навіть на тензорних ядрах. Тут присутні усі необхідні мені бібліотеки, функції та команди, а також гнучке керування і компіляція «на ходу».

3 СТВОРЕННЯ НЕЙРОННОЇ МЕРЕЖІ

3.1 Вибір та завантаження бібліотек і програмних структур

Бібліотека програмування - це набір попередньо написаного коду, який програмісти можуть використовувати для оптимізації завдань. Ця колекція багаторазового коду зазвичай призначена для вирішення конкретних типових проблем. Бібліотека зазвичай включає кілька різних попередньо закодованих компонентів.

Розробники використовують бібліотеки для більш ефективного створення програм і веб сайтів. Кожна бібліотека розроблена, щоб забезпечити вирішення певної функції. Це може включати автентифікацію користувача, підключення до сервера, інтерфейс користувача, керування даними, алгоритми, анімацію тощо. Розробники часто шукають бібліотеки, щоб допомогти з певним компонентом, який вони хочуть швидко створити або з яким виникають проблеми. Потім вони виберуть усі компоненти, які вони бажають використати, з однієї бібліотеки, щоб їхній додаток був якомога більш цілісним. Іноді розробники також використовують бібліотеки, щоб переглянути те, над чим вони працюють, з іншої точки зору. Кожен програміст робить щось по-різному. Розробники можуть звернутися до бібліотек, щоб побачити, як вони могли б зробити щось іншим способом.

Слова «фреймворк» і «бібліотека» часто використовуються як синоніми. Але більшість програмістів стверджують, що насправді це дві різні речі. Ви можете подумати про це так: фреймворк - це модель будинку, а бібліотека - це меблі та декор, що входять всередину. У більшості випадків бібліотека - це набір об’єктів і функцій, які можна використовувати окремо та які потрібно налаштувати для спільної роботи. Бібліотеки зосереджені на розв’язанні конкретної проблеми в межах сфери розвитку. Бібліотеки дозволяють вам диктувати потік програми, додавати спеціальний код і швидко додавати лише ті компоненти, які вам потрібні. Фреймворки часто більше нагадують креслення, це набори закодованих компонентів, які вже налаштовані для спільної роботи. Фреймворки, як правило,

містять попередньо встановлену архітектуру або дизайн, з якими мають працювати розробники, часто зосереджені на функціонуванні як комплексного рішення для певної методології. Модель будинку не має багато індивідуальних варіантів. Він базується на попередньо визначеному плані та стандартних параметрах.

Я використав наступні бібліотеки: matplotlib, time, json, copy, numpy, PIL, collections, os.

Matplotlib - бібліотека мовою програмування Python для візуалізації даних двовимірною та тривимірною графікою. Отримані зображення можуть бути використані як ілюстрації в публікаціях. Matplotlib поширюється на умовах BSD ліцензії. Зображення, що генеруються в різних форматах, можуть бути використані в інтерактивній графіці, в наукових публікаціях, графічному інтерфейсі користувача, веб-додатках, де потрібна побудова діаграм. Matplotlib побудована за принципами ООП, але має процедурний інтерфейс pylab, який надає аналоги команд MATLAB.

Модуль time - забезпечує різні функції, пов’язані з часом. Хоча цей модуль завжди доступний, не всі функції доступні на всіх платформах. Більшість функцій, визначених у цьому модулі, викликають функції бібліотеки платформи C з однаковою назвою.

JSON - простий формат обміну даними, що базується на підмножині синтаксису JavaScript. Модуль json дозволяє кодувати та декодувати дані у зручному форматі.

Модуль copy являє собою поверхневу чи глибоку операцію копіювання.

NumPy - бібліотека для роботи з масивами. Має функції для роботи в області лінійної алгебри, перетворень Фур’є та матриць.

PIL - бібліотека для роботи з зображеннями, а точніше з растровою графікою. Підтримує всі формати фотозображень.

За основу був узятий фреймворк PyTourch. PyTorch - фреймворк машинного навчання мови Python з відкритим вихідним кодом, створений з урахуванням бібліотеки Torch. Використовується для вирішення різних завдань: комп’ютерний зір, обробка природної мови. Розробляється переважно групою штучного інтелекту

Facebook. Також навколо цього фреймворку вибудовано «екосистему», що складається з різних бібліотек, якы розробляються сторонніми командами: PyTorch Lightning і Fast.ai, що спрощують процес навчання моделей, Pyro, модуль для ймовірнісного програмування, від Uber, Flair, для обробки природної мови та Catalyst.

3.2 Розробка програми

Для створення проекту заходимо на Google Colab, і натискаємо на «Файл» - «Створити блокнот». Відкривається наш проект. Приклад зображено на рис. 3.1.

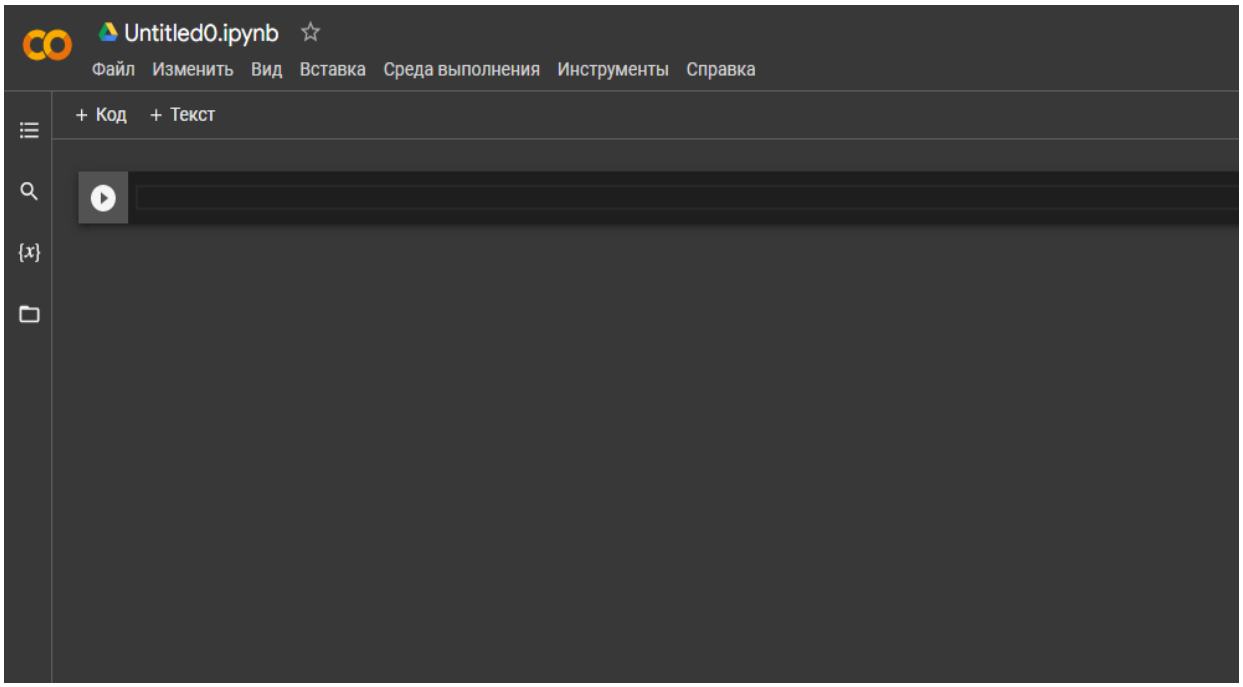

Рисунок 3.1 – Новостворений проект

Далі необхідно вибрати де саме буде обчислюватись наша мережа, на графічному процесорі, чи на центральному. Для цього обираємо вкладку «Середовище виконання» - «Апаратний прискорювач», обираємо GPU. Приклад наведений на рис. 3.2.

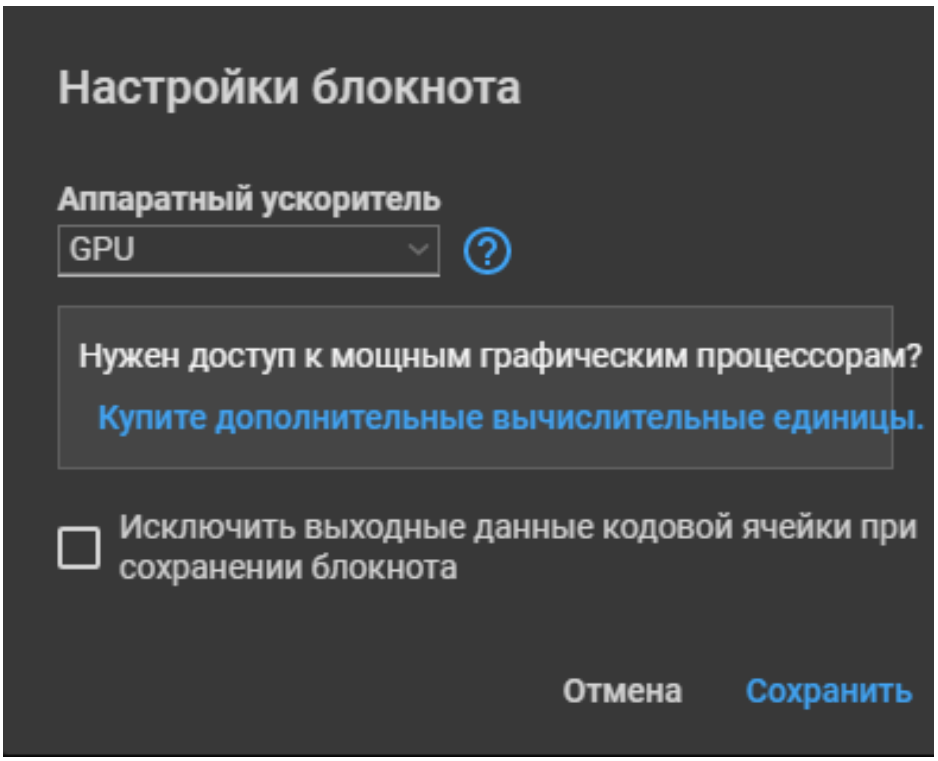

Рисунок 3.2 - Вибір апаратного прискорювача

Перед початком роботи також необхідно завантажити дані з репозиторію вільного доступу для навчання нейронної мережі. Виконуємо команди у вкладці «Код», що зображені на рис. 3.3.

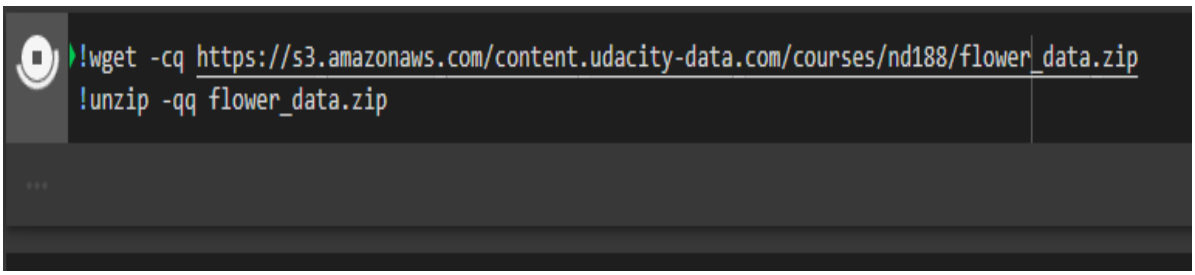

Рисунок 3.3 - Завантаження додаткових даних

Завантажені файли будуть знаходитись у вкладці «Файли». Приклад наведений у рис. 3.4.

Файли що були завантажені мають назву «flower_data» та «cat_to_name.json». Всередині папки знаходиться 2 пули зображень: вибірка на навчання та вибірка на тренування.

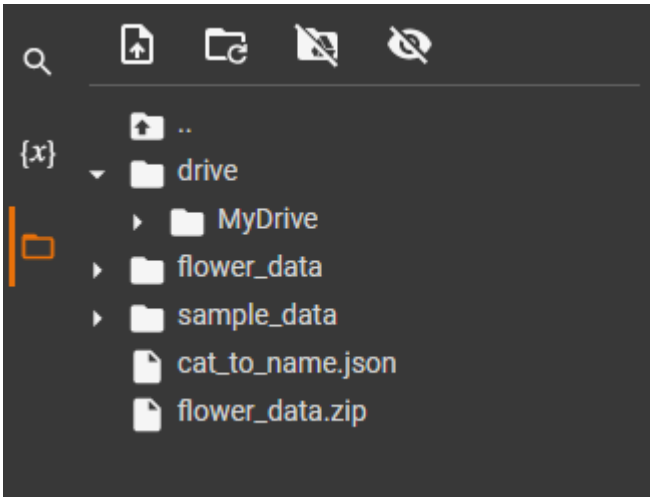

Рисунок 3.4 - Завантажені файли

Першим кроком буде імпорт усіх фреймворків і бібліотек, які були описані вище. На рис. 3.5 зображені усі бібліотеки, необхідні для побудови нейронної мережі.

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import time
import json
import copy

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import PIL

from PIL import Image
from collections import OrderedDict


import torch
from torch import nn, optim
from torch.optim import lr_scheduler
from torch.autograd import Variable
import torchvision
from torchvision import datasets, models, transforms
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn as nn
import torch.nn.functional as F

import os
```

Рисунок 3.5 – Імпорт усіх бібліотек і фреймворків

Для додавання бібліотеки використовується команда «import». Визначаємо тип пристрою, на якому проводимо розрахунки за допомогою функції «tain_on_gpu». Приклад коду наведений на рис. 3.6.

```
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('...Training on CPU ...')
else:
    print('You are good to go!  Training on GPU ...')
```

Рисунок 3.6 - Визначення типу пристрою для обчислення

Наступним кроком буде перетворення даних. Потрібно переконатися, що використовується декілька різних типів перетворень, щоб допомогти мережі навчитися якомога більше. Для цього зображення можна перегорнути, повернути або обрізати.

Це означає, що відхилення надаються для нормалізації значень зображення перед передачею їх в мережу, але їх можна знайти через тензор відхилення різних розмірів. На рис. 3.7 показана частина коду, що відповідає за перетворення даних.

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomRotation(30),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])
}

# Load the datasets with ImageFolder
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'valid']}

# Using the image datasets and the trainforms, define the dataloaders
batch_size = 64
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                              shuffle=True, num_workers=4)
               for x in ['train', 'valid']}

class_names = image_datasets['train'].classes
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid']}
class_names = image_datasets['train'].classes
print(dataset_sizes)
print(device)
```

Рисунок 3.7 - Перетворення вхідних даних

Як видно на рис. 3.7, також були визначені розмір вхідного пакету та класи. Для перевірки коректності роботи мережі, роблю перевірку командою «print (dataset_sizes), print (device)».

Наступним кроком буде маркування папок і зображень, що в них. На рис. 3.8 зображений вміст файлу .json.

{"21": "fire lily", "3": "canterbury bells", "45": "bolero deep blue", "1": "pink primrose", "34": "mexican aster", "27": "prince of wales feathers", "7": "moon orchid", "16": "globe-flower", "25": "grape hyacinth", "26": "corn poppy", "79": "toad lily", "39": "siam tulip", "24": "red ginger", "67": "spring crocus", "35": "alpine sea holly", "32": "garden phlox", "10": "globe thistle", "6": "tiger lily", "93": "ball moss", "33": "love in the mist", "9": "monkshood", "102": "blackberry lily", "14": "spear thistle", "19": "balloon flower", "100": "blanket flower", "13": "king protea", "49": "oxeye daisy", "15": "yellow iris", "61": "cautleya spicata", "31": "carnation", "64": "silverbush", "68": "bearded iris", "63": "black-eyed susan", "69": "windflower", "62": "japanese anemone", "20": "giant white arum lily", "38": "great masterwort", "4": "sweet pea", "86": "tree mallow", "101": "trumpet creeper", "42": "daffodil", "22": "pincushion flower", "2": "hard-leaved pocket orchid", "54": "sunflower", "66": "osteospermum", "70": "tree poppy", "85": "desert-rose", "99": "bromelia", "87": "magnolia", "5": "english marigold", "92": "bee balm", "28": "stemless gentian", "97": "mallow", "57": "gaura", "40": "lenten rose", "47": "marigold", "59": "orange dahlia", "48": "buttercup", "55": "pelargonium", "36": "ruby-lipped cattleya", "91": "hippeastrum", "29": "artichoke", "71": "gazania", "90": "canna lily", "18": "peruvian lily", "98": "mexican petunia", "8": "bird of paradise", "30": "sweet william", "17": "purple coneflower", "52": "wild pansy", "84": "columbine", "12": "colt's foot", "11": "snapdragon", "96": "camellia", "23": "fritillary", "50": "common dandelion", "44": "poinsettia", "53": "primula", "72": "azalea", "65": "californian poppy", "80": "anthurium", "76": "morning glory", "37": "cape flower", "56": "bishop of llandaff", "60": "pink-yellow dahlia", "82": "clematis", "58": "geranium", "75": "thorn apple", "41": "barbeton daisy", "95": "bougainvillea", "43": "sword lily", "83": "hibiscus", "78": "lotus lotus", "88": "cyclamen", "94": "foxglove", "81": "frangipani", "74": "rose", "89": "watercress", "73": "water lily", "46": "wallflower", "77": "passion flower", "51": "petunia"}

Рисунок 3.8 - Назви зображень рослин у папках

PyTorch дозволяє легко завантажувати попередньо навчені алгоритми та використовувати їх для побудови мережі. Деякі з найпопулярніших попередньо навчених моделей, як-от ResNet, AlexNet і VGG, походять від ImageNet Challenge. Ці попередньо навчені моделі дозволяють швидко отримувати передові результати в області комп’ютерного зору, не потребуючи такої великої кількості обчислювальної потужності, терпіння та часу. DenseNet, як на мене, показує найкращі результати і порівняно швидко видає дуже хороші результати. На рис. 3.9 я впроваджую даний алгоритм в нейромережу.

```
model_name = 'densenet' #vgg
if model_name == 'densenet':
    model = models.densenet161(pretrained=True)
    num_in_features = 2208
    print(model)
elif model_name == 'vgg':
    model = models.vgg19(pretrained=True)
    num_in_features = 25088
    print(model.classifier)
else:
    print("Unknown model, please choose 'densenet' or 'vgg'")
```

Рисунок 3.9 - Впровадження алгоритму

Я також додав деяку варіативність у виборі алгоритму, а саме мережа Densenet або vgg, на випадок якщо потрібно буде змінити алгоритм.

Тепер необхідно створити класифікатор. Мною було впроваджено модель, в якій можна легко змінювати кількість прихованих шарів, а також регулювати відсоток фільтрації. З оптимізатором, критерієм і планувальником можна проявити будь-яку творчість. Критерій - це метод, який використовується для оцінки відповідності моделі, оптимізатор - це метод оптимізації, який використовується для оновлення ваг, а планувальник надає різні методи для налаштування швидкості навчання та розміру кроку, що використовується під час оптимізації. На рис. 3.10 зображена частина коду, на якій відображений класифікатор і приховані шари.

Нейронні мережі відмінно працюють як класифікатори, так як при застосуванні правильних алгоритмів навчання і кількості зв’язків отримуємо мережу з точністю більше як 95%.

Наразі алгоритм класифікації використовує функцію активації ReLU, яка показує дуже вражаючі результати при класифікації зображень.

```
# Create classifier
for param in model.parameters():
    param.requires_grad = False

def build_classifier(num_in_features, hidden_layers, num_out_features):

    classifier = nn.Sequential()
    if hidden_layers == None:
        classifier.add_module('fc0', nn.Linear(num_in_features, 102))
    else:
        layer_sizes = zip(hidden_layers[:-1], hidden_layers[1:])
        classifier.add_module('fc0', nn.Linear(num_in_features, hidden_layers[0]))
        classifier.add_module('relu0', nn.ReLU())
        classifier.add_module('drop0', nn.Dropout(.6))
        classifier.add_module('relu1', nn.ReLU())
        classifier.add_module('drop1', nn.Dropout(.5))
        for i, (h1, h2) in enumerate(layer_sizes):
            classifier.add_module('fc'+str(i+1), nn.Linear(h1, h2))
            classifier.add_module('relu'+str(i+1), nn.ReLU())
            classifier.add_module('drop'+str(i+1), nn.Dropout(.5))
        classifier.add_module('output', nn.Linear(hidden_layers[-1], num_out_features))

    return classifier
hidden_layers = None#[4096, 1024, 256][512, 256, 128]

classifier = build_classifier(num_in_features, hidden_layers, 102)
print(classifier)
```

Рисунок 3.10 - Побудова класифікатора

На рис. 3.11 зображено програмування оптимізатора, описаного вище.

```
 # Only train the classifier parameters, feature parameters are frozen
if model_name == 'densenet':
    model.classifier = classifier
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adadelta(model.parameters()) # Adadelta #weight optim.Adam(model.parameters(), lr=0.001, momentum=0.9)
    #optimizer_conv = optim.SGD(model.parameters(), lr=0.0001, weight_decay=0.001, momentum=0.9)
    sched = optim.lr_scheduler.StepLR(optimizer, step_size=4)
elif model_name == 'vgg':
    model.classifier = classifier
    criterion = nn.NLLLoss()
    optimizer = optim.Adam(model.classifier.parameters(), lr=0.0001)
    sched = lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)
else:
    pass
```

Рисунок 3.11 - Оптимізатор мережі

Наступним кроком безпосередньо переходжу до алгоритму навчання

нейронної мережі.

На рис. 3.12 зображені фрагменти коду, які описують кількість епох при тренуванні мережі.

```
def train_model(model, criterion, optimizer, sched, num_epochs=5):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch+1, num_epochs))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'valid']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        #sched.step()
                        loss.backward()

                        optimizer.step()
```

Рисунок 3.12 - Алгоритм навчання мережі

Кожна епоха тренування має фазу тренування та перевірки результатів, також присутнє алгоритм зворотного поширення похибки для навчання на тренувальній стадії.

Також було реалізовано відслідковування результатів тренування мережі в режимі реального часу. На рис. 3.13 відображений результат такого відслідковування.

Під час роботи нейромережі, виводяться дані про якість її навчання. Маю зауважити, що модель напрочуд швидко досягла результатів у 95%.

```
Epoch 1/30
----------
train Loss: 2.5175 Acc: 0.4652
valid Loss: 0.9789 Acc: 0.8240

Epoch 2/30
----------
train Loss: 0.8339 Acc: 0.8405
valid Loss: 0.4596 Acc: 0.9059

Epoch 3/30
----------
train Loss: 0.5297 Acc: 0.8900
valid Loss: 0.3249 Acc: 0.9315

Epoch 4/30
----------
train Loss: 0.4010 Acc: 0.9183
valid Loss: 0.2483 Acc: 0.9450
```

Рисунок 3. 13 - Результати роботи мережі в режимі рельного часу

Тренування мережі зайняло більше ніж пів години часу. Наразі необхідно запровадити функцію порівняння, а також для визначення кількості правильних суджень мережі. Алгоритм зображений на рис. 3.14.

```
# Evaluation

model.eval()

accuracy = 0

for inputs, labels in dataloaders['valid']:
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)

    # Class with the highest probability is our predicted class
    equality = (labels.data == outputs.max(1)[1])

    # Accuracy is number of correct predictions divided by all predictions
    accuracy += equality.type_as(torch.FloatTensor()).mean()

print("Test accuracy: {:.3f}".format(accuracy/len(dataloaders['valid'])))
```

Рисунок 3.14 - Алгоритм розрахунку кількості правдивих суджень

Однією з особливостей мереж Densnet є можливість створення контрольної точки, до якої можна повернутись. Точка зберігає вже навчену мережу. Алгоритм для збереження стану мережі наведений на рис. 3.15. На тому ж рисунку зображений варіант завантаження мережі до контрольної точки.

```
# Saving the checkpoint
model.class_to_idx = image_datasets['train'].class_to_idx
checkpoint = {'input_size': 2208,
              'output_size': 102,
              'epochs': epochs,
              'batch_size': 64,
              'model': models.densenet161(pretrained=True),
              'classifier': classifier,
              'scheduler': sched,
              'optimizer': optimizer.state_dict(),
              'state_dict': model.state_dict(),
              'class_to_idx': model.class_to_idx
             }

torch.save(checkpoint, 'checkpoint_ic_d161.pth')
# Loading the checkpoint
ckpt = torch.load('checkpoint_ic_d161.pth')
ckpt.keys()
# Load a checkpoint and rebuild the model
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = checkpoint['model']
    model.classifier = checkpoint['classifier']
    model.load_state_dict(checkpoint['state_dict'])
    model.class_to_idx = checkpoint['class_to_idx']
    optimizer = checkpoint['optimizer']
    epochs = checkpoint['epochs']

    for param in model.parameters():
        param.requires_grad = False

    return model, checkpoint['class_to_idx']
model, class_to_idx = load_checkpoint('checkpoint_ic_d161.pth')
model
idx_to_class = { v : k for k,v in class_to_idx.items()}
image_path = 'flower_data/valid/102/image_08006.jpg'
img = Image.open(image_path)
def process_image(image):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns an Numpy array
    '''
```

Рисунок 3.15 - Реалізація контрольних точок нейронної мережі.

Тепер виведемо результати роботи нейромережі за допомогою бібліотеки PIL. На рис. 3.16 відображений графічний вивід певної фотографії.

```
image_path = 'flower_data/valid/102/image_08006.jpg'
img = Image.open(image_path)
def process_image(image):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns an Numpy array
    '''
    # Process a PIL image for use in a PyTorch model
    # tensor.numpy().transpose(1, 2, 0)
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])
    image = preprocess(image)
    return image
def imshow(image, ax=None, title=None):
```

Рисунок 3.16 - Функція графічного виводу інформації

Результатом буде вивід вказаної фотографії, як на рис. 3.17.

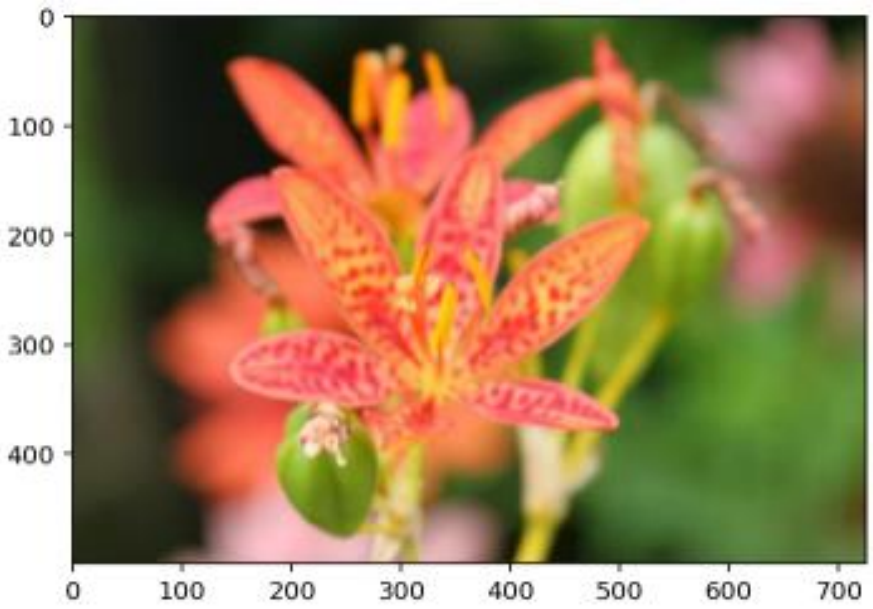

Рисунок 3.17 - результат роботи алгоритму виводу зображень нейромережі

Отже, первинним результатом роботи нейромережі є досягнута точність у 96%, що відображено на рис. 3.18, а також можливість графічного виводу фотографії в режимі реального виконання програми.

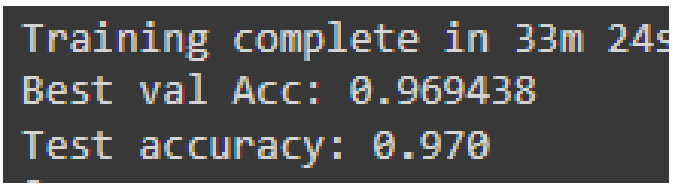

Рисунок 3.18 - Фінальний результат роботи мережі

3.2 Механізм передбачення

Коли зображення мають правильний формат, можна написати функцію для прогнозування за допомогою даної моделі. Одна з поширених практик полягає в тому, щоб передбачити 5 чи близько того найімовірніших класів. Потрібно обчислити ймовірності класу, а потім знайти найбільші значення цих ймовірностей. Щоб отримати найбільші значення верхньої більшої у тензорі, використовую k.topk(). Цей метод повертає як найвищі k ймовірностей, так і індекси цих ймовірностей, що відповідають класам. Потрібно перетворити ці індекси на фактичні мітки класу за допомогою class_to_idx, який був доданий до моделі. Цей метод має визначити шлях до зображення та контрольної точки моделі, а потім повернути ймовірності та класи.

Інструменти прогнозної аналітики базуються на кількох різних моделях і алгоритмах, які можна застосовувати в широкому діапазоні випадків використання. Визначення того, які методи прогнозного моделювання найкращі для вашої компанії, є ключовим для отримання максимальної віддачі від рішення прогнозної аналітики та використання даних для прийняття глибоких рішень.

На рис. 3.19 зображений код ймовірнісного прогнозування наступних 5 зображень.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.class_to_idx = image_datasets['train'].class_to_idx
def predict2(image_path, model, topk=5):
    ''' Predict the class (or classes) of an image using a trained deep learning model.
    '''

    # Implement the code to predict the class from an image file
    img = Image.open(image_path)
    img = process_image(img)

    # Convert 2D image to 1D vector
    img = np.expand_dims(img, 0)


    img = torch.from_numpy(img)

    model.eval()
    inputs = Variable(img).to(device)
    logits = model.forward(inputs)

    ps = F.softmax(logits,dim=1)
    topk = ps.cpu().topk(topk)

    return (e.data.numpy().squeeze().tolist() for e in topk)
img_path = 'flower_data/valid/18/image_04252.jpg'
probs, classes = predict2(img_path, model.to(device))
print(probs)
print(classes)
flower_names = [cat_to_name[class_names[e]] for e in classes]
print(flower_names)
def view_classify(img_path, prob, classes, mapping):
    ''' Function for viewing an image and it's predicted classes.
    '''
    image = Image.open(img_path)
```

Рисунок 3.19 - Алгоритм прогнозування

На рис. 3.20 та рис. 3.21 зображені кінцеві графічні виводи прогнозованих результатів.

Рисунок 3.20 - Кінцевий результат роботи алгоритму прогнозування

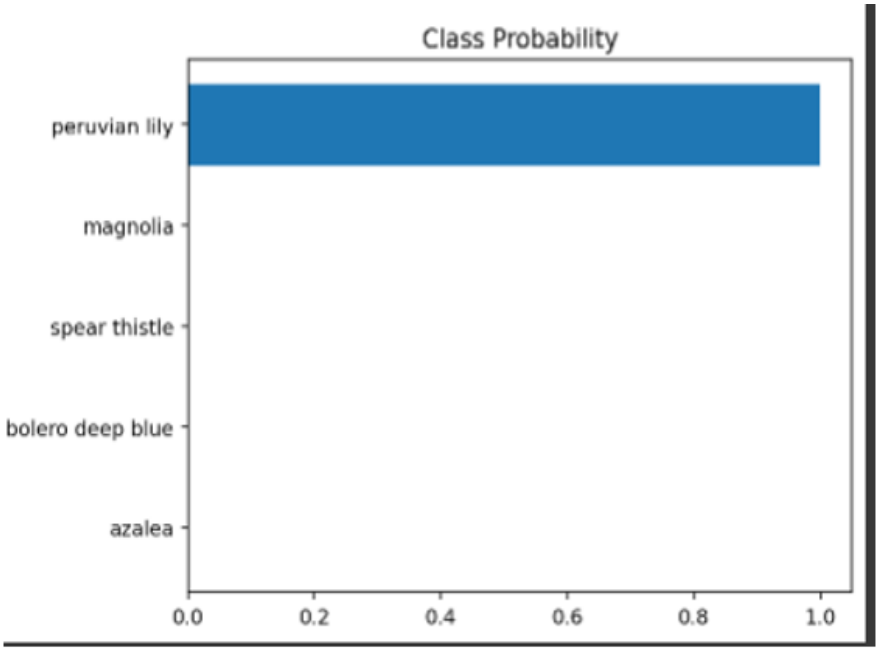

Рисунок 3.21 - Графічний вивід результатів передбачення

ВИСНОВКИ

Провівши дослідження на тему нейронних мереж можна з впевненістю сказати, що подальше їх використання в усіх сферах нашого життя неминуче. Штучні мережі допомагають людям обирати музику, фільми, робити прогнози на майбутнє.

На підприємствах нейронні мережі керують цілими цехами виробництва, так як їх ефективність сягає більше ніж 96%.

Медицина активно використовує ШНМ для прогнозування розвитку смертельних хвороб.

Завдяки останнім оновленням алгоритмів Keras, PyTourch, Tensor майже кожен може зробити свій внесок у розробку ШНМ. Нейромережі, що аналізують природну мову, можуть використовуватися для створення чат-ботів, які дозволяють клієнтам отримати необхідну інформацію про продукти компанії. Безпілотні автомобілі - концепт, над яким працює більшість великих концернів, а також технологічні компанії та стартапи, у своїй роботі спирається на нейромережі. Навіть сільське господарство – майже в усіх сферах нашого життя присутні нейронні мережі, які безсумнівно роблять наше життя кращим.

Звісно вони не ідеальні і мають свої недоліки у вигляді складності реалізації, їх вартості та кваліфікованих кадрів для розробки.

Але прогрес не стоїть на місці і вже через кілька років людство перейде до використання ШНМ у повсякденному житті.

ПЕРЕЛІК ПОСИЛАНЬ

1. Goodfellow I. Deep Learning / Goodfellow I., Bengio Y. and Courville A.; Cambridge MA : MIT Press [2017] – 777 pages.

2. Springenberg, J. T. Striving for Simplicity: The All Convolutional Net / Springenberg, J. T. Dosovitskiy, A.; Brox, T. & Riedmiller, M. – Режим доступа: https://arxiv.org/abs/1412.6806.

3. Ruder S. (2016) An overview of gradient descent optimisation algorithms. – Режим доступу: https://arxiv.org/abs/1609.04747.

4. Robinds H. A stochastic approximation method // Annals of Mathematical Statistics / Robinds H. and Monro S. – 1951 – vol. 22 – pp. 400–407.

5. Darken, C. Learning rate schedules for faster stochastic gradient search / Darken, C., Chang, J. Moody, J. // Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop, September 1–11, 1992.

6. Dauphin, Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization / Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., Bengio, Y. – Режим доступу: http://arxiv.org/abs/1406.2572.

7. Sutton, R. S. Two problems with backpropagation and other steepest-descent learning procedures for networks. Proc. 8th Annual Conf. Cognitive Science Society – 1986.

8. Kingma, D. P. Adam: a Method for Stochastic Optimization / Kingma, D. P., Ba, J. L. // International Conference on Learning Representations, May 7-9, 2015, SanDiego, USA.

9. Nielsen M. Neural Networks And Deep Learning. – Режим доступу: http://neuralnetworksanddeeplearning.com/.

10. Sermanet, P. Traffic Sign Recognition with Multi-Scale Convolutional Networks / Sermanet, P., LeCun, Y. // The 2011 International Joint Conference on Neural Networks, September 2011.

11. Сховище зображень для навчання нейронних мереж [Електронний ресурс]. «The CIFAR-10 dataset». Режим доступу:

https://www.cs.toronto.edu/~kriz/cifar.html.

12. Principles of training multi-layer neural network using backpropagation [Електронний ресурс] – режим доступу: http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html.

13. An overview of gradient descent optimization algorithms [Електронний ресурс]: – режим доступу: https://ruder.io/optimizing-gradientdescent/index.html.

14. Введение в архитектуры нейронных сетей [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://habr.com/ru/company/olegbunin/blog/340184/.

15. Harris Drucker, Yann Le Cun. Improving Generalization Performance Using Backpropagation / Harris Drucker, Yann Le Cun // IEEE Transactions on Neural Networks. - 1992. - Vol.3, №5 - p.991-997.

16. Барський, А. Б. Нейронні мережі: розпізнавання, управління, прийняття рішень - М .: Фінанси і статистика, 2004. - 176 с. 3.

17. Хайкін С. Нейронні мережі. Повний курс. Друге видання. - М .: Вільямс, 2006. - 1104 с. 4.

18. Саттон Р.С. Навчання з підкріпленням. - М .: БИНОМ. Лабораторія знання, 2012. - 399 с

19. Комарцова, Л. Г. Нейрокомп'ютери: навчальний посібник / Л. Г. Комарцова, А. В. Максимов. - М .: МГТУ, 2002. - 318 с.

20. Крісілов, В. А. Методи прискорення навчання нейронних мереж / В. А. Крісілов, Д. Н. Олешко. - М .: Гардарика, 2005. - 1042 с.

21. Рутковська, Д. Нейронні мережі, генетичні алгоритми та нечіткі системи / Д. Рутковська, М. Піліньскій, Л. Рутковський. - М .: Гаряча лінія. - Телеком, 2006. - 147 с.

22. Форсайт, Д. А. Комп'ютерне зір. Сучасний похід / Д. А. Форсайт, П. Джин. - М .: Вільямс, 2004. - 928 с.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (презентація)