Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерних систем та мереж

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

за напрямом 123 «Комп’ютерна інженерія»

на тему: МЕТОДИ ПАРАМЕТРИЗАЦІЇ ТА ОПТИМІЗАЦІЇ ШТУЧНИХ НЕЙРОСТРУКТУР ДЛЯ РОЗПІЗНАВАННЯ АНІМАЦІЙНИХ ОБ'ЄКТІВ

Виконав _____ П. В. Бригінець

Керівник випускної роботи _____ А. І. Купін

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2023

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ФН – формальний нейрон
ШНМ – штучні нейроні мережі
API – Application Programming Interface
CNN – Convolutional Neural Networks
RBF – Radial Basis Function
RNN – Recurrent Neural Networks
SGD – Stochastic gradient descent
LSTM – long short-term memory
РНМ – Рекурентні Нейроні Мережі

ВСТУП

Порівняно зі звичайним алгоритмічним програмуванням нейромежеві комп’ютерні системи дають можливість реалізувати такі функції, що складно піддаються формалізації, наприклад: розпізнавання зображень, визначення ризиків кардіологічних захворювань, метеорологічні прогнози тощо. Широкий діапазон застосування нейромереж та суміжних рішень на їхній основі за останнє десятиліття робить тему досить актуальною.

Паралельно з розвитком комп’ютерних технологій в 50–х роках, розпочалися дослідження функціонування та структури людського мозку, саме ці дані становлять основу нейронних мереж. Нейронні мережі представляють собою програмну або апаратну реалізацію за принципом біологічних нейронних мереж – з’єднань біологічних нервових клітин. Нервові клітини мають складні процеси збудження, фізіологічного функціонування, передачі сигналу тому для повного моделювання потребується потужний комп’ютер лише для одного нейрона, що й стало необхідністю в спрощеній математичній моделі. Сучасні рішення налічують тисячі формальних нейронів, як наприклад AlphaGo для гри в ґо на основі поглибленої нейронної мережі. Розроблена вона компанією Google DeepMind в 2015 році. AlphaGo стала першою в світі програмою, яка виграла матч без гандикапу у професійного гравця в ґо [1].

Актуальність роботи. Нейронні мережі та глибоке навчання наразі забезпечують найкращі рішення для багатьох проблем розпізнавання образів, мови та обробки природної мови. Особливої уваги заслуговують розроблені платформи та API, які значно спрощують навчання та аналіз системи. Можливість реалізувати складну задачу без точного налаштування значень окремих одиниць вагів штучних нейронів, а глобальний підбір матеріалів для навчання та гнучкість системи загалом без сумніву показує перспективність напряму в майбутньому. Задіяти всі обчислювальні ресурси графічних процесорів без сумніву необхідно для масштабних проектів, тому розглянеться апаратне забезпечення і його зв'язок із швидкістю навчання.

Тлумачення анімації - досить обширне від комп’ютерної 3D графіки до растрових зображень. З апаратної точки зору анімація рисунка представляє собою швидку, понад 16 кадрів в секунду, послідовність зображень що візуалізуються. Та анімацію на основі рендеренгу 3D моделей с постановкою сцени, освітлення і пост ефектів. Незалежно від типу анімації її можна розбити на дискретні зображення, і через геометричні представлення відтворити модель. Прикладом такого рішення є скелетна анімація, що теоретично спрощує навчання нейронної мережі.

Мета і завдання дослідження. Якщо фізичний рендеринг і створення якісно освітлених статичних сцен стають доступні ентузіастам завдяки потужним безкоштовним ігровим движкам і інструментам 3D моделювання, то створення гарної анімації вимагає обладнання для захоплення рухів і тривалої копіткої роботи по їх впровадженню. Одна з найдоступніших систем це костюм neuronmocap вартістю близько $ 2к без урахування доставки.[22]

Розробки в області глибинного навчання нейронних мереж потенційно можуть вирішити цю проблему: вони можуть вчитися на великих наборах даних, і одного разу навчені, вони займають мало пам'яті і швидко виконують поставлені завдання. Залишається відкритим питання про те, як саме нейронні мережі найкраще застосовувати таким чином, щоб отримувати високоякісний результат в режимі реального часу з мінімальною обробкою даних. Дослідники з Единбурзького університету в 2017 розробили систему навчання, яка називається фазово-функціональної нейронною мережею (PFNN), яка використовує машинне навчання для анімації персонажів у відеоіграх і інших додатках.

Підбірка результатів з використанням PFNN для перетину нерівній місцевості: персонаж автоматично пересувається відповідно до призначеним для користувача управлінням в реальному часі і геометрією оточення.[23]

Актуальність роботи. Рішення по розпізнаванню або генерації актуальні при високих обчислювальних можливостях навченої нейронної мережі. Але такі рішення затратні по економічно-фінансовим ресурсам, що і потребує дослідження можливості оптимізації навчання нейронних мереж.

Мета досліджень. Основною метою є огляд математичного представлення нейронних структур, їх типів і принцип функціонування.

Визначити доцільність використання готових платформних середовищ або розробка нових нейронних мереж. Оцінка технічних вимог теоретичної мережі відповідно певної ланки апаратних рішень. Зробити детальний опис коду, певного практичного рішення платформного або програмного в залежності на основі встановлених рішень в розділі огляду платформ.

Об’єкт дослідження – процеси навчання штучних нейронних мереж створених на основі відкритої платформи TensorFlow.

Предмет дослідження – моделі та засоби реалізації штучних нейронних мереж, їх вплив на параметри точності, детермінації, часу навчання розглянутої моделі і в цілому.

Методи досліджень. Використовуючи реалізований приклад, встановити параметри оптимізації нейронній мережі за такими напрямами: кількість навчальних епох, коефіцієнт навчання, алгоритму навчання, змінна функції активації, проведення оцінки швидкості навчання.

Наукова новизна одержаних результатів. Побудова модель дослідження впливу швидкості навчання від різних параметрів, на основі яких можна згрупувати та оптимізувати процес навчання математичних моделей нейронних мереж. Прослідковано закономірності досліджуємих параметрів на розглянутій інтерпретації.

Практичне значення отриманих результатів. Результати досліджень дозволять пришвидшити процеси навчання для моделей нейронних мереж реалізованих на платформах, і підбору алгоритмів у випадку повного циклу розробки моделі.

Апробація роботи. Апробація досліджень відбувалась на XVI Всеукраїнська науково-практична WEB конференції аспірантів, студентів та молодих вчених «Комп'ютерні інтелектуальні системи та мережі» (КІСМ–2023) від Криворізького національного університету (21–23 березня 2023 р., м. Кривий Ріг). Також була участь у «Всеукраїнському науково-практичній конференції здобувачів вищої освіти та молодих учених "Комп'ютерна інженерія і кібербезпека: досягнення та інновації" (27–29 листопада 2022 р., м. Кропивницький).

1. ОСНОВНІ КОНЦЕПЦІЇ НЕЙРОМЕРЕЖ

1.1 Особливості формалізації нейронів

Біологічний нейрон – складна система, математична модель якого досі повністю не побудована. Введено безліч моделей, що розрізняються обчислювальною складністю і схожістю з реальним нейроном. Одна з найважливіших – формальний нейрон, незважаючи на простоту ФН, мережі, побудовані з таких нейронів, можуть сформувати довільну багатомірну функцію на виході наведено на рисунку 1.1.

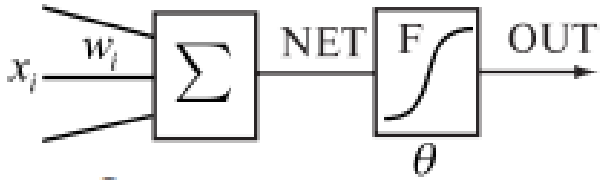

Рисунок 1.1 – Формальний нейрон (структура)

Нейрон складається з зваженого суматора і нелінійного елемента. Функціонування нейрона визначається формулами (1.1–1.2):

$$\text{NET} = \sigma_i \; \square_{\square} \square_{\square} \qquad (1.1)$$

$$\text{OUT} = F(\text{NET} - \theta) \qquad (1.2)$$

де x_i – вхідні сигнали, сукупність всіх вхідних сигналів нейрона утворює вектор x;

w_i – вагові коефіцієнти, сукупність вагових коефіцієнтів утворює вектор ваг w;

NET – зважена сума вхідних сигналів, значення NET передається на нелінійний елемент;

θ – пороговий рівень даного нейрона;

F – нелінійна функція, звана функцією активації.

Нейрон має кілька вхідних сигналів x і один вихідний сигнал OUT. Параметрами нейрона, що визначають його роботу, є: вектор ваг w, пороговий рівень θ і вид функції активації F [3].

Розглянемо основні види функцій активації, які поширені в штучних НС.

1. Жорстка сходинка у формулі (1.3):

$$OUT = \{0, NET < 0 \; 1, \quad NET \geq 0 \qquad (1.3)$$

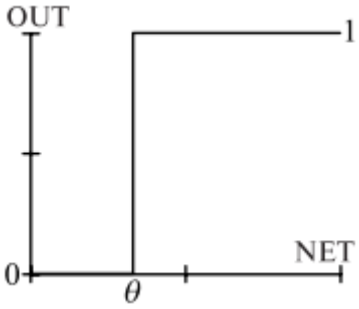

Рисунок 1.2 – Функція жорсткої сходинка

Використовується в класичному формальному нейроні. Розвинена повна теорія, дозволяє синтезувати довільні логічні схеми на основі ФН з такою нелінейностю. Функція обчислюється двома – трьома машинними інструкціями, тому нейрони з такою нелінійністю вимагають малих обчислювальних ресурсів. Ця функція надмірно спрощена і не дозволяє моделювати схеми з безперервними сигналами. Відсутність першої похідної ускладнює застосування градієнтних методів для навчання таких нейронів. Мережі на класичних ФН найчастіше формуються, синтезуються, тобто їх параметри розраховуються за формулами, на противагу навчанню, коли параметри підлаштовуються ітеративно.

1. Логістична функція (сигмоїда, функція Фермі) у формулі 1.4:

$$OUT = \frac{1}{1+e^{-NET}} \quad (1.4)$$

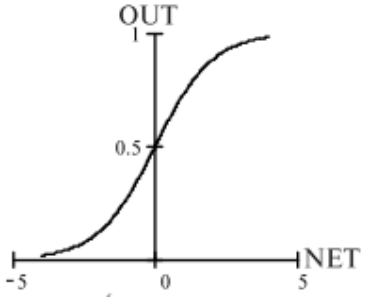

Рисунок 1.3 – Логістична функція

Часто використовується для багатошарових перцептронів та інших мереж з безперервними сигналами. Гладкість, безперервність функції – важливі позитивні якості. Безперервність першої похідної дозволяє навчати мережу градієнтними методами (наприклад, метод зворотного поширення помилки). Функція симетрична щодо точки (NET = 0, OUT = 1/2), це робить рівноправними значення OUT = 0 і OUT = 1, що істотно в роботі мережі. Проте, діапазон вихідних значень від 0 до 1 несиметричний, через це навчання значно сповільнюється.

Дана функція – стискає, тобто для малих значень NET коефіцієнт передачі K = OUT /NET великий, для великих значень він знижується. Тому діапазон сигналів, з якими нейрон працює без насичення, виявляється широким.

Значення похідної легко виражається через саму функцію. Швидкий розрахунок похідної прискорює навчання.

1. Гіперболічний тангенс формула (1.5) на рисунку 1.4 :

$$OUT = th(NET) = \frac{e^{NET} - e^{-NET}}{e^{NET} + e^{-NET}} \quad (1.5)$$

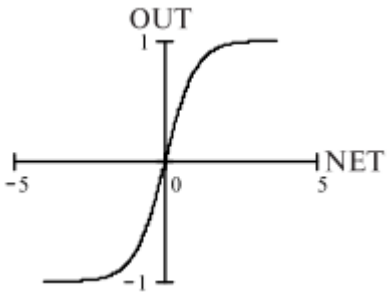

Рисунок 1.4 – Логістична функція (приклад)

Теж застосовується часто для мереж з безперервними сигналами. Функція симетрична щодо відносно точки (0,0), це перевага в порівнянні з сигмоїдою.

Похідна також неперервна і виражається через саму функцію.

4. Положиста сходинка на рисунку 1.5:

$$OUT = \{0, NET \le 0 \ \frac{(NET-\theta)}{\Delta}, \theta \le NET < \theta + \Delta \ 1, NET \ge \theta + \Delta \quad (1.6)$$

Розраховується легко, але має розривну першу похідну в точках NET = θ, NET = + Δ θ, що ускладнює алгоритм навчання наведена формула (1.6).

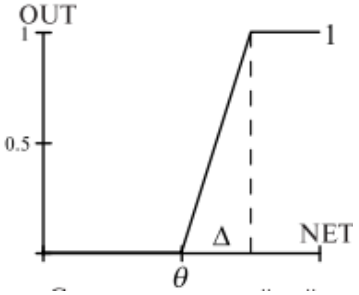

Рисунок 1.5 – Логістична функція

5. Експонента: NET OUT = e^{-NET}. Застосовується в спеціальних випадках.

6. SOFTMAX – функція у формулі (1.7):

$$OUT = \frac{e^{NET}}{\sum_i \ e^{NET}} \quad (1.7)$$

Тут підсумовування проводиться по всіх нейронах даного шару мережі. Такий вибір функції забезпечує суму виходів шару, що дорівнює одиниці при будь–яких значеннях сигналів NETi даного шару. Це дозволяє трактувати OUTi як ймовірності подій, сукупність яких (всі виходи шару) утворює повну групу. Це корисна властивість дозволяє застосувати SOFTMAX – функцію в задачах класифікації, перевірки гіпотез, розпізнавання образів і у всіх інших, де потрібні виходи–ймовірності.

7. Ділянки синусоїди у формулі (1.8):
OUT = sin (NET) для,

$$NET = \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \text{ або } NET = [-\pi, \pi] \quad (1.8)$$

8. Гаусова крива на рисунку 1.6:

$$OUT = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{(NET-m)}{2\sigma^2}} \quad (1.9)$$

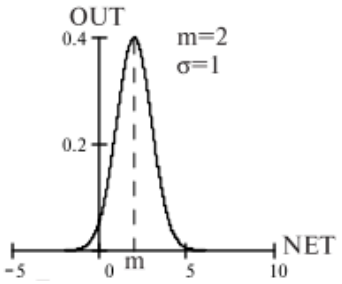

Рисунок 1.6 – Гаусова крива (модель)

Застосовується у випадках, коли реакція нейрона повинна бути максимальною для деякого визначеного значення NET у формулі (1.9).

9. Лінійна функція, OUT = K NET, K = const. Застосовується для тих моделей мереж, для яких не потрібне послідовне з'єднання шарів нейронів один за одним.

1.2 Персептрон

Поняття штучного нейрона і штучної нейронної мережі з'явилися досить давно, ще в 1943 році. Це була чи не перша стаття, в якій робилися спроби змоделювати роботу мозку. Її автором був Уоррен Мак–Каллок.

Ці ідеї продовжив нейрофізіолог Френк Розенблат. Він запропонував схему пристрою, що моделює процес людського сприйняття, і назвав його «персептроном» (від латинського perceptio – сприйняття). У 1960 році Розенблатт представив перший нейрокомп'ютер – «Марк–1», який був здатний розпізнавати деякі букви англійського алфавіту.

Таким чином персептрон є однією з перших моделей нейромереж, а «Марк–1» – першим у світі нейрокомп'ютером.

Персептрони стали дуже активно досліджувати. На них покладали великі надії. Однак, як виявилося, вони мали серйозні обмеження. Був такий вчений Мінський, який був однокурсником Розенблатта. Він написав цілу книгу (1971 рік), в якій провів детальний їх аналіз, попутно показавши, що вони не так вже й багато і вміють, та й взагалі сильно обмежені.

Розенблатт не встиг написати відповідь Мінському, так як загинув в свій 43 день народження під час катастрофи човна.

З тих пір ентузіазм вчених в дослідженні персептронов і штучних мереж вщух. Стали перспективними інші області. Про нейромережі забули. Але потім були відкриті нові види мереж, а також алгоритми їх навчання, що знову відродило інтерес до цієї області.

В основі персептрона лежить математична модель сприйняття інформації мозком. Різні дослідники по–різному його визначають. У найзагальнішому своєму вигляді (як його описував Розенблатт) він представляє систему з елементів трьох різних типів: сенсорів, асоціативних елементів і реагують елементів.

Першими в роботу включаються S–елементи. Вони можуть перебувати або в стані спокою (сигнал дорівнює 0), або в стані збудження (сигнал дорівнює 1).

Далі сигнали від S–елементів передаються A–елементів по так званим S–A зв'язків. Ці зв'язки можуть мати ваги, рівні тільки –1, 0 або 1.

Потім сигнали від сенсорних елементів, які пройшли по S–A зв'язків потрапляють в A–елементи, які ще називають асоціативними елементами. Варто зауважити, що одному A–елементу може відповідати кілька S–елементів. Якщо сигнали, що надійшли на A–елемент, в сукупності перевищують деякий його поріг θ, то цей A–елемент збуджується і видає сигнал, рівний 1. В іншому випадку (сигнал від S–елементів не перевищив порогу A–елемента), генерується нульовий сигнал.

Чому A–елементи назвали асоціативними? Справа в тому, що A–елементи є агрегаторами сигналів від сенсорних елементів. Наприклад, у нас є група сенсорів, кожен з яких розпізнає частину літери «Д» на досліджуваній зображенні. Однак тільки їх сукупність (тобто коли декілька сенсорів видали сигнал, рівний 1) може порушити A–елемент цілком. На інші літери A–елемент не реагує, тільки на букву «Д». Тобто він асоціюється з буквою «Д». Звідси і така назва.

Можна навести й інший приклад. Насправді людські очі складаються з неймовірної кількості S–елементів (сенсорів), що уловлюють падаюче світло (близько 140 000 000). І у вас якийсь A–елемент, який розпізнає конкретну частину обличчя. І ось ви побачили на вулиці людину. Деякі A–елементи, які розпізнали конкретні частини особи, порушуються.

Далі сигнали, які справили порушені A–елементи, направляються до суматору (R–елемент), дія якого вам вже відомо. Однак, щоб дістатися до R–елемента, вони проходять по A–R зв'язків, у яких теж є ваги. Однак тут вони вже можуть приймати будь–які значення (на відміну від S–A зв'язків).

Фінальний акорд. R–елемент складає один з одним зважені сигнали від A–елементів та, якщо перевищено певний поріг, генерує вихідний сигнал, рівний 1. Це означає, що в загальному потоці інформації від очей ми розпізнали обличчя людини. Якщо поріг не перевищено, то вихід персептрона дорівнює –1. Тобто ми не виділили особа із загального потоку інформації.

Персептрон (Perceptron) – найпростіший вид нейронних мереж. В основі лежить математична модель сприйняття інформації мозком, що складається з сенсорів, асоціативних і реагують елементів.

Є кілька підвидів персептронов, на деякі з яких будуть розглянені. Історично виділилися кілька типів персептронов. Їх часто плутають, так як різні автори, які писали статті на цю тему, найчастіше розуміли під персептроном схожі, але все ж таки різні математичні моделі.

Я вже наводив загальне визначення персептрона вище. Тепер розглянемо його підвид – персептрон з одним прихованим шаром (або елементарний персептрон). Саме цей тип персептронов часто мають на увазі, коли говорять про персептрони взагалі.

Шар саме прихований, тому що A–елементи розташовані між шарами S–елементів та R–елементів.

Персептрон з одним прихованим шаром – персептрон, у якого є тільки по одному шару S, A і R елементів.

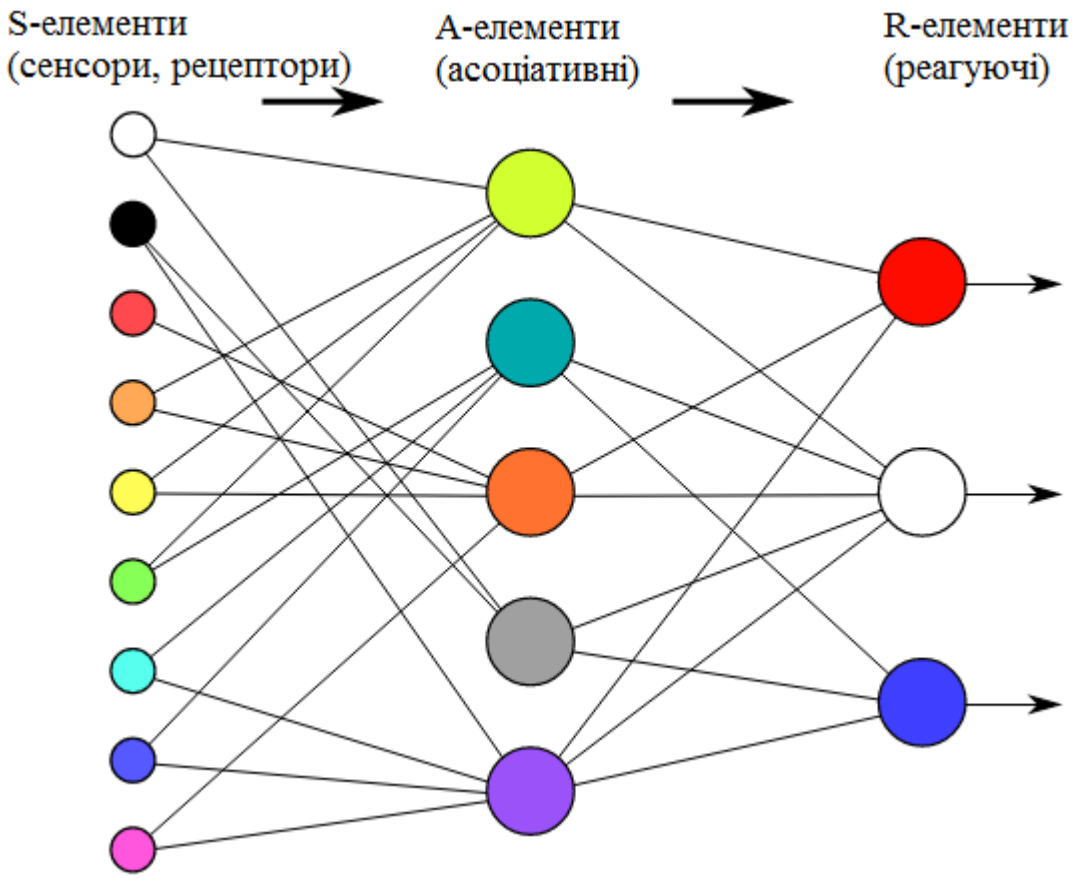

Рисунок 1.7– Зображення персептрона

1.2.1 Одношаровий персептрон

Одношаровий персептрон схожий за назвою, але це не те ж саме, що і персептрон з одним прихованим шаром, хоча так може здатися. Саме цей тип персептронов буде розгянуто докладніше. Цей тип персептронов, як і елементарні персептрони, теж часто мають на увазі, коли говорять про персептрони взагалі.

Його ключова особливість полягає в тому, що кожен S–елемент однозначно відповідає одному A–елементу, все S–A зв'язку мають вагу, рівний +1, а поріг A елементів дорівнює 1.

Виходячи з ключової особливості одношарового персептрона сенсор може бути однозначно пов'язаний тільки з одним асоціативним елементом. Сенсор може передавати сигнал тільки одному A–елементу. Прибираємо зайву зв'язок. Ту ж операцію проводимо і з іншими сенсорами.

Обов'язково кожен S–елемент однозначно відповідає одному A–елементу. Це означає, що кожен сенсор може передавати сигнал тільки одному A–елементу. Однак це твердження зовсім не забороняє ситуації, коли декілька сенсорів передають сигнал на один A–елемент.

Далі, S–A зв'язку завжди мають вагу, що дорівнює одиниці, а поріг A–елементів завжди дорівнює +1. З іншого боку відомо, що сенсори можуть подавати сигнал рівний тільки 0 або 1.

Перший S–елемент, нехай, генерує сигнал, що дорівнює одиниці. Сигнал проходить по S–A зв'язку і не змінюється, так як будь–яке число, помножене на 1 дорівнює самому собі. Поріг будь–якого A–елемента дорівнює 1. Так як сенсор справив сигнал, рівний 1, то A–елемент однозначно збудився. Це означає, що він видав сигнал, рівний 1 (так як він також має здатність генерувати тільки 1 або 0 на своєму виході). Далі цей одиничний сигнал множиться на довільну вагу A–R зв'язку і потрапляє в відповідний R–елемент, який підсумовує підсумки на нього зважені сигнали, і якщо вони перевищують його поріг, видає +1. В іншому випадку вихід даного R–елемента дорівнює –1.

Не рахуючи сенсорних елементів і S–A зв'язків, тільки що описано схему роботи штучного нейрона. Одношаровий персептрон дійсно є штучним нейрон з невеликою відмінністю. На відміну від штучного нейрона, у одношарового персептрона вхідні сигнали можуть приймати фіксовані значення: 0 або 1. У штучного нейрона на вхід можна подавати будь–які значення.

У персептрони R–елементи підсумовують зважені вхідні сигнали і, якщо зважена сума вище деякого порога, видають 1. Інакше виходи R–елементів були б рівні –1.

Така поведінка легко задається функцією активації під назвою функція одиничного стрибка. Відмінність полягає в тому, що функція одиничного стрибка видає 0, якщо поріг не перевищено, а тут видає –1, але це не суттєво.

Таким чином стає ясно, що частина одношарового персептрона можна представити у вигляді штучного нейрона. Оскільки не відмінялися S–елементи, яких в штучному нейроні просто немає і в одношаровому персептрони S–елементи і A–елементи можуть приймати тільки фіксовані значення 0 і 1, тоді як в штучному нейроні таких обмежень немає.

Одношаровий персептрон – персептрон, кожен S–елемент якого однозначно відповідає одному A–елементу, S–A зв'язку завжди рівні 1, а поріг будь–якого A–елемента дорівнює 1.

Частина одношарового персептрона відповідає моделі штучного нейрона. Одношаровий персептрон може бути і елементарним персептроном, у якого тільки по одному шару S, A, R–елементів.

1.2.2 Багатошаровий персептрон

Під багатошаровим персептроном розуміють два різних види: багатошаровий персептрон по Розенблатта і багатошаровий персептрон по Румельхарту.

Багатошаровий персептрон по Розенблатта містить більше 1 шару A–елементів.

Багатошаровий персептрон по Румельхарту є окремим випадком багатошарового персептрона по Розенблатта, з двома особливостями: S–A зв'язку можуть мати довільні ваги і навчатися нарівні з A–R зв'язками.

Навчання проводиться за спеціальним алгоритмом, який називається навчанням за методом зворотного поширення помилки.

Цей метод є фундаментальним каменем навчання всіх багатошарових ШНМ. Багато в чому завдяки йому відновився інтерес до нейронних мереж. Багатошаровий персептрон по Румельхарту – багатошаровий персептрон по Розенблатта, у якого навчання підлягають ще й S–A зв'язку, а також саме навчання проводиться за методом зворотного поширення помилки.

Персептрони дуже добре вирішують завдання класифікації. Це означає якщо є групи об'єктів (наприклад, кішки і собаки), то персептрон після навчання зможе вказувати до якої групи належить об'єкт (до кішок або собак).

Якщо є поле сенсорів (матриця) і якась класифікація, що залежить від нього, то безліч елементарних персептронов, які проводять успішну класифікацію не є порожнім.

Під полем сенсорів розуміється безліч всіх S–елементів. Під класифікацією – придумані нами класи (ті ж кішки і собаки). Під «непустою безліччю елементарних персептронов, які проводять успішну класифікацію» розуміється, що знайдеться хоча б один перспетрон, впорався з класифікацією об'єктів.

На прикладі завдання щоб персептрон навчився розрізняти кішок і собак. Для класифіцації буде 3 сенсора: довжина лап, окрас і форма морди. Так як S–елементи можуть приймати значення 0 або 1, то умовно прийняти, що значення 1 будуть відповідати коротким лапам, змішаного окрасу і округла морда відповідно. Значення 0 означатимуть ознака собаки на даному S–

елементі (довгі лапи, однотонний забарвлення і витягнута морда). Ось ми і отримали сенсорне поле. Якщо хочете, його можна представити у вигляді безлічі можливих значень 0 і 1 у кожного S–елемента. Наприклад, абсолютна кішка повинна викликати спрацьовування всіх S–елементів {1,1,1}.

Ідеальною ж собаці відповідає наступний набір виходів S–елементів: {0,0,0}.

Самі по собі сенсори не грають ролі. Але додавши до набору виходів сенсорів сенс: кішка або собака, ми тим самим поставили деяку класифікацію. Математично це означає, що ми задали деяку функцію, яка приймає набір виходів S–елементів, а її значенням є 0 або 1 (кішка або собака).

Можна вибрати будь–який набір S–елементів та будь–яку класифікацію. І безліч «рішень» все одно не буде порожнім.

Це означає, що теоретично персептрони здатні вирішувати будь–яке завдання на класифікацію.

Але є і друга теорема, доведена Розенблатта:

Якщо є поле сенсорів (матриця) і якась класифікація, що залежить від нього, то процес навчання з корекцією помилок, що починається з довільного початкового стану, завжди приведе до досягнення рішення протягом кінцевого проміжку часу.

Під довільним початковим станом тут розуміється персептрон з довільними S–A і A–R вагами зв'язків. Під рішенням в теоремі розуміється персептрон з певними вагами, успішно вирішальний нашу задачу на класифікацію [4].

1.3 Мережа радіально–базисних функцій(RBF–мережа)

Штучні нейронні мережі на основі радіально–симетричних (радіально–базисних) функцій можуть використовуватися для вирішення широкого кола завдань, серед яких найбільш часті – апроксимація, класифікація та кластеризація даних.

Основна властивість радіально–симетричних функцій – це монотонне і симетричне щодо деякої вертикальної осі симетрії зміна (спадання або зростання) їх відгуків.

Як приклад такої функції може служити вираз функції Гаусса $f_3(s) = e^{-a(s-T)^2}$. Нейронні мережі на основі радіально–симетричних функцій. Саме ця функція найбільш часто використовується в даній архітектурі нейронних мереж, однак головним чином, в її багатовимірному випадку:

$$h \rightarrow (x) = exp\,(-a * \|x - c \rightarrow \|^2) \qquad (1.10)$$

Де $c \rightarrow$ вектор центрів (координат вертикальних осей симетрії) безлічі радіально–симетричних функцій у формулі (1.10); $\|x - c \rightarrow\|$ норма вектора відхилень вхідної змінної від центрів радіально–симетричних функцій. Параметр α пов'язаний з радіусом розсіювання вхідних змінних r і може бути

замінений у виразі (1.11) на відповідне відношення:.

$$a = \frac{1}{2r^2} \tag{1.11}$$

Нейронні мережі на основі радіально–симетричних функцій.

Норма різниці векторів розраховується як евклідова відстань наведено у формулі (1.12):

$$\|x - c \rightarrow \| = \sqrt{(x - c_1)^2 + (x - c_2)^2 + \ldots + (x - c_m)^2} \tag{1.12}$$

На рисунку 1.8 наведена типова структура штучної нейронної мережі на основі радіально–симетричних функцій.

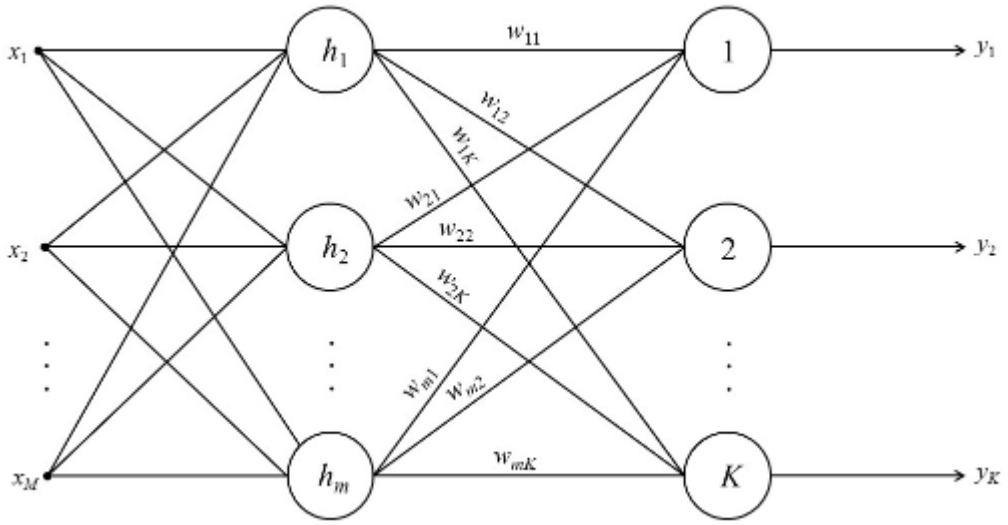

Рисунок 1.8 – Структура нейронної мережі на основі радіально–симетричних функцій

Дана структура містить два шари нейронів. Виходи першого шару активуються безліччю радіально–симетричних функцій (1). Фактично вони обробляють вектор вхідних значень, визначаючи ступінь близькості кожного з них до центрів радіально–симетричних функцій. Виходи нейронів другого шару (т. Е. Виходи всієї нейронної мережі) – це лінійні комбінації виходів першого шару.

Склад і кількість входів і виходів визначаються класом розв'язуваної задачі. При апроксимації даних входи – це аргументи апроксимуючої залежності, а виходи – повертаються нею значення. При кластеризації та класифікації даних входи – це характерні ознаки, за якими розрізняються об'єкти, що відносяться до кластерів або класам, а виходи вказують на відповідний входів кластер або клас.

Кількість прихованих елементів також залежить від розв'язуваної задачі. Якщо це апроксимація даних, воно може бути будь–яким. У разі кластеризації та класифікації даних повинна відповідати кількості кластерів або еталонних образів класів.

Життєвий цикл штучних нейронних мереж на основі радіально–симетричних функцій, як і для більшості інших архітектур, включає дві стадії: навчання і практичного використання. У свою чергу, на стадії навчання можна

виділити також два етапи: настройка нейронної мережі і оптимізація синаптичних коефіцієнтів лінійного вихідного шару.

На етапі налаштування даної нейронної мережі необхідно визначити центри С радіуси r радіальних елементів (нейронів прихованого шару).

При наявності невеликої кількості еталонних зразків для навчання в якості центрів радіально–симетричних функцій слід вибирати відповідні їм вектора. Якщо обсяг навчальної вибірки досить великий, як центри можуть бути використані:

– центри потенційних кластерів, за якими можна розподілити всі приклади навчальної вибірки вручну або з використанням додаткових алгоритмів кластеризації, в тому числі інших архітектур нейронних мереж;

– окремі випадкові приклади навчальної вибірки.

Слід зауважити, що другий варіант краще застосовувати при великій кількості нейронів в прихованому шарі.

Вибір радіусів радіальних елементів визначається необхідним видом радіально–симетричною функції. При великих значеннях параметра а графік функції занадто гострий, а це значить, що мережа не буде коректно інтерполювати дані між відомими точками на досить великій відстані від них, так як втрачає здатність до узагальнення навчальних даних. Навпаки, при надмірно малих значеннях параметра а мережу стає несприйнятливою до окремих деталей.

З урахуванням вищесказаного радіуси можуть задаватися наступними способами:

– користувачем нейронної мережі в явному вигляді на основі евристичного підбирання;

– розраховуватися автоматично по середній відстані до декількох (в залежності від загального обсягу навчальної вибірки і кількості прихованих нейронів) найближчих прикладів.

На етапі оптимізації вагових коефіцієнтів лінійного вихідного шару послідовно виконуються наступні дії.

1. Розраховується характеристична матриця значень радіально–симетричних елементів всіх навчальних прикладів:

$$\bar{H} = [h_1(x_1)\ h_2(x_1)\ \dots\ h_m(x_1)\ \ h_1(x_2)\ h_2(x_2)\ \dots\ h_m(x_2)\ \ \dots\ h_1(x_N)\ \ \dots\ \dots\ \dots\ \ h_2(x_N)\ \dots\ h_m($$

(1.13)

Нейронні мережі на основі радіально–симетричних функцій.

Кількість рядків даної матриці дорівнює кількості прикладів навчальної вибірки. Кількість стовпців – кількості радіальних елементів у формулі (1.13).

2. Методами лінійної алгебри розраховується матриця вагових коефіцієнтів вихідного шару нейронів у формулі (1.14):

$$\bar{W} = \left(\bar{H}^T * \bar{H}\right)\bar{H}^T * \bar{Y} \qquad (1.14)$$

Нейронні мережі на основі радіально–симетричних функцій, де матриця виходів навчальних прикладів містить стовпці в кількості, що дорівнює числу навчальних прикладів, і рядки в кількості, що відповідає числу виходів

нейронної мережі наведена у формулі (1.15):

$$\bar{Y} = [y_{11}\ y_{21}\ \dots\ y_{K1}\ y_{12}\ y_{22}\ \dots\ y_{K2}\ \dots\ y_{1N}\ \dots\ \dots\ \dots\ y_{2N}\ \dots\ y_{KN}\ ] \quad (1.15)$$

Серед переваг даної архітектури нейронних мереж виділяють:

– наявність єдиного прихованого шару, достатнього для моделювання яскраво виражених нелінійних залежностей;

– простота алгоритму оптимізації вагових коефіцієнтів;

– гарантоване знаходження глобального оптимуму функції помилки при знаходженні вагових коефіцієнтів нейронів вихідного шару;

– висока швидкість навчання.

До обмежень або недоліків нейронних мереж на основі радіально–симетричних функцій можна віднести:

– необхідність спеціальної настройки параметрів радіально–симетричних функцій, складність настройки при великій кількості прихованих радіальних елементів;

– неможливість екстраполірованія моделі за межами вихідного інтервалу зміни вхідних значень навчальної вибірки.

Підвищити точність рішення задачі апроксимації з використанням штучної нейронної мережі на основі радіально–симетричних функцій можна за рахунок додавання нових радіальних елементів або зміни налаштувань наявних прихованих нейронів. У межі кількість радіальних елементів може збігатися з кількістю експериментальних точок. У цьому випадку завдання зводиться до інтерполяції експериментальних даних, в результаті чого значення, що розраховуються нейронною мережею, будуть в точності повторювати результати експерименту в відповідних точках.

1.4 Сгорткова нейрона мережа Convolutional Neural Networks, CNN

Як відомо, нейронні мережі отримують вхідні дані (один вектор), після чого перетворюють інформацію, проводячи її через ряд прихованих шарів. Кожен схований шар складається з безлічі нейронів, де всякий нейрон має стійку зв'язок з усіма нейронами в попередньому шарі і де нейрони в функції одного шару повністю незалежні друг від друга і не мають спільних з'єднань. Останній повнозвязний шар називається вихідним шаром, і в налаштуваннях класифікації він демонструє число класів.

Звичайні нейронні сети погано масштабуються в разі зображень великих розмірів. Так, в системі комп'ютерного зору CIFAR–10 картинки мають розмір [32 × 32 × 3] (32 – ширина, 32 – висота, 3 – кольорові канали), тому один повністю підключений нейрон в першому схованому шарі звичайної нейронної мережі має масу 3 072 (32 * 32 * 3). Здається, що це значення можна змінити, але повнозв'язна структура не масштабується для великих зображень. Картинка з дозволом побільше, наприклад [200 × 200 × 3], приведе к тому, що повністю підключений нейрон буде важити 120 000.

Крім того, майже напевно мати кілька таких нейронів потрібно, що привело до додавання параметрів. Повна зв'язность – це розростання, і величезна кількість параметрів може швидко привести до переоснащення.

Згорткові нейронні мережі користуються тим, що введені дані складаються з зображень, і вони обмежують побудову мережі більш розумним шляхом. В відмінності від звичайної нейронної мережі, CNN складаються з нейронів, розташованих в 3–х вимірах: ширина, висоти і глибини, тобто виміри, які формують об'єм. Наприклад, зображення на вході CIFAR–10 є вхідними активаційними об'ємами, а об'єм утворений вимірами 32 × 32 × 3. Нейрони будуть підключатися тільки до невеликої області шару перед цим відрізком. Крім того, результуючий вихідний шар для даної системи комп'ютерної зору складе 1x1x10, оскільки до кінця побудови CNN ми перетворюємо повне зображення в єдиний вектор оцінок класу, розташованих по виміру глибини. Нижче приводиться візуалізація описаного процесу на рисунку 1.9.

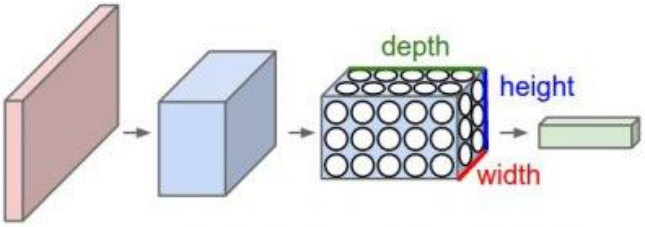

Рисунок 1.9 – Процес згортки мережою

Як уже було сказано, схематично CNN – це послідовність слоїв. Кожний шар перетворює один активаційний об'єм в інший за допомогою диференційованої функції. Для організації згорткової нейронної мережі використовується 3 основних шари:

a) згортки,
b) пулінга (інакше підвиборки або субдискретизації),
c) повносвязний слой.

Ці шари використовуються з метою побудови повної архітектури CNN.

Далі розглянемо організацію CNN в більш детальному вигляді, а простий приклад згорткової нейромережі в контексті класифікації розпізнавання зображень CIFAR–10 може служити архітектурі [INPUT – CONV – RELU – POOL – FC].

В INPUT (вхідні дані) [32 × 32 × 3] содержатся вихідна информація про изображении (в даному випадку 32 – ширина, 32 – висота, 3 – кольорові канали R, G, B).

Шар CONV (шар згортки) збільшує значення фільтра на вихідні значення пікселів зображення (після елементарного множення), після чого всі ці множення додаються. Кожна унікальна позиція введеного зображення робить число. Приклад: Якщо використовується 12 фільтрів, об'єм буде рівний 32 * 32 * 12 [32 × 32 × 12]. Якщо використовується фільтра 5 × 5 на активаційній карті на рисунку 1.10.

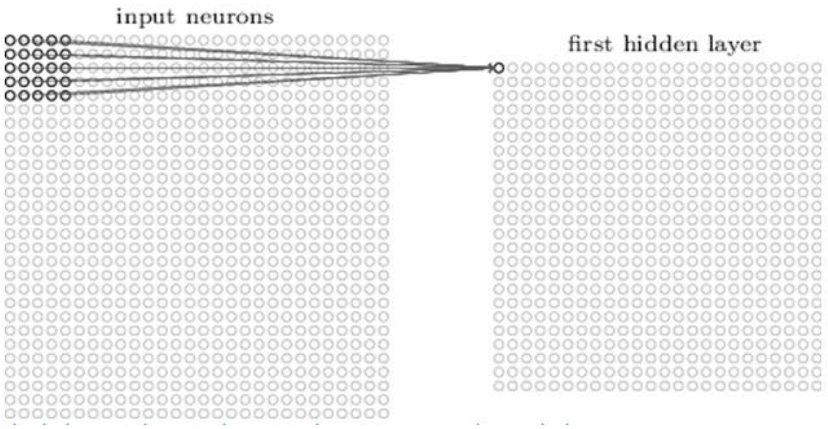

Рисунок 1.10 – Візуалізація згортки вхідного зображення фільтра 5 × 5 на активаційній карті

Шар RELU (блок лінійної ректифікації) застосовує елементну функцію активації у вигляді f (x) = max (0, x), встановлюючи нульовий порог. Іншими словами, RELU виконує наступні дії: якщо x> 0, то обсяг залишається попереднім ([32 × 32 × 12]), а якщо x <0, то відкидуються непотрібні деталі в каналі і замість замін на 0.

Шар POOL (шар пулинга) виконує операцію по зниженню дискретизації територіальних розмірів (ширина і висота), в результаті чого об'єм може скоротитися до [16 × 16 × 12]. Тобто на такій виконується нелінейне згущення карт ознак. Логіка роботи така: якщо на попередній операції звертки вже були виявлені деякі ознаки, то для подальшої обробки такий детальний образ вже не потрібний, і він ущільнюється до менш детальної картинки.

Шар FC (повносвязний шар) виводить N–мірний вектор (N – кількість класів) для визначення потрібного класу. Робота організується шляхом звернення до виходу попереднього шару (карта ознак) та визначення властивостей, які найбільш характерні для певного класу.

Саме таким чином CNN шар за шаром трансформує вихідне зображення, починаючи з вихідних значень пікселів і закінчуючи визначенням класу. Зверніть увагу, що шари не обов'язкового повинні містити параметри. Зокрема, шар згортки і повнозв'язну шар виконують перетворення, які є не тільки функцією від вхідного активационного обсягу, а й параметрів (ваги і зміщення нейронів). З іншого боку, блок лінійної ректифікації і шар POOL реалізують фіксовану функцію. Параметри в шарах CONV і FC навчатимуться за допомогою градієнтного спуску, тому визначення класу згорточної нейромережею залежить від міток в навчальному наборі для кожного зображення.

Отже у найпростішому випадку архітектура CNN – це набір шарів, які перетворять образ зображення в вихідний образ (наприклад, визначення класу). Кожен шар відповідає за певний етап процесу обробки зображення (шар згортки, лінійної ректифікації, пулінг і повнозв'язну шар – найпопулярніші на сьогоднішній день).

.

Кожен шар отримує на вході об'ємну 3D інформацію і трансформує зі збереженням 3D–об'єму за допомогою функції, що диференціюється. Шар може не мати параметрів (CONV і FC мають, RELU і POOL – немає). Шар може не мати додаткові гіперпараметри (наприклад, CONV, FC і POOL мають, RELU – немає).

Припустимо, що шар CONV працює без «мозковий» або нейронної підгрунтя. Параметри шару згортки складаються з набору учнів фільтрів. Кожен фільтр має малі просторові габарити (ширину і висоту), але проходить по всій глибині обсягу введення. Наприклад, стандартний фільтр першого шару згоркової нейронної мережі може бути розміру [5x5x3]. Під час проходу вперед, ми прослизають (точніше, скручуємо) кожен фільтр по ширині і висоті вхідних даних і обчислюємо скалярний твір між записами фільтра і входом в будь–яке положення. У міру проходження фільтра по ширині і висоті зображення, ми складаємо 2–мірну активаційну карту, яка надає відгуки цього фільтра на кожному просторової позиції. Мережа вивчає фільтри, активуються при виявленні певної візуальної особливості. Це може бути грань деякої спрямованості, плямистість конкретного кольору на першому шарі або кільцеподібні візерунки на більш високих рівнях мережі. Для цілого набору фільтрів в кожному де кожен з шарів згортки буде формувати окрему 2–мірну активаційну карту.

Якщо проводити аналогії з нейронами, тоді кожен запис в вихідному обсязі можна інтерпретувати як вихідний нейрон, який дивиться тільки на невелику ділянку вхідного обсягу і просторово ділить параметри з усіма нейронами зліва і справа (оскільки вони є результатом застосування такого ж фільтра). Розглянемо деталі зв'язків між нейронами, їх розташування в просторі і модель поділу параметрів.

Коли мова йде про роботу з вхідною інформацією з високою розмірністю, установка зв'язку між нейронами і всіма нейронами з колишнім обсягом є недоцільною. Замість цього підключається кожен нейрон тільки до локальної області вхідного обсягу. Просторова протяжність зв'язку з цим є гіперпараметром і називається рецептивним полем (полем сприйнятливості), де площа поля сприйнятливості дорівнює площі фільтра. Просторова протяжність уздовж осі глибини завжди еквівалента глибині вхідного обсягу. Слід ще раз підкреслити асиметрію в тому, як ми розглядаємо просторові виміри (ширину і висоту), а як – глибину: сполуки є локальними по ширині і висоті, але завжди проходять через всю глибину вхідного обсягу.

Існують 3 гіперпараметра, які контролюють розмір вихідної інформації: глибина, крок і доповнення нулями.

Де перший гіперпараметр вихідного обсягу – глибина. Він еквівалентний числу фільтрів, які потрібно використовувати; кожен з фільтрів навчається знаходженню різних даних на вході. Наприклад, перший згортковий шар в якості вхідної інформації бере необроблене зображення, після різні нейрони, розташовані уздовж глибини, можуть активуватися в разі наявності,

наприклад, згустків певного кольору. Будемо називати безліч нейронів, які «дивляться» на одну ділянку вхідного обсягу, стовпцем глибини (деякі вважають за краще термін «волокно»).

По–друге, слід вказати крок, з яким фільтр виконує прохід. Коли крок дорівнює 1, то за один раз переміщуємо фільтри на 1 піксель. Коли крок дорівнює 2 (іноді буває 3, але це рідко використовується на практиці), фільтри «перестрибують» через 2 точки за раз при кожному зверненні до них. Це дозволить виробляти менші вихідні обсяги просторових вимірів.

Іноді буває зручно оточувати кордон вхідного зображення нулями. Розмір цього доповнення нулями і є гіперпараметром. Ключовою особливістю доповнення нулями є те, що воно дозволяє контролювати просторовий розмір вихідних обсягів (найчастіше використовують таку властивість, щоб зберегти просторовий розмір вхідної інформації, тобто з метою збереження вхідних та вихідних значень висоти і ширини однаковими).

Просторовий розмір вихідного обсягу можна обчислити як функцію від вхідного обсягу (W), розміру рецептивного поля нейронів згорточного шару (F), кроку, з яким вони переміщуються (S), і кількості заповнення нулями на кордоні вхідний картинки (P). Тому формула (W–F + 2P) / S + 1 для підрахунку кількості «потрібних» нейронів є правильною. Наприклад, для вхідного обсягу 7 × 7 і фільтра 3 × 3 з кроком 1 і доповненням 0, на виході отримаємо обсяг 5 × 5. З кроком 2 вихідне значення було б дорівнює 3 × 3 [9].

1.5 Рекурентна нейрона мережа Recurrent Neural Networks, RNN

Представивиши найпростішу нейромережу – одношаровий перцептрон у якого видача всіх нейронів об'єднується тим чи іншим чином, і нейромережа дає відповідь, але можливості такої архітектури сильно обмежені. Якщо потрібно отримати більш просунутий функціонал, можна піти декількома шляхами, наприклад, збільшити кількість шарів і додати операцію згортки, яка б «розшаровується» входять дані на шматочки різних масштабів. В цьому випадку у вас вийдуть згорткові нейромережі для глибинного навчання, які досягли успіху в обробці зображень і розпізнаванні предметів. Однак що у примітивного перцептрону, що у згорткової нейромережі є загальне обмеження: і вхідні і вихідні дані мають фіксований, заздалегідь визначений розмір, наприклад, картинка 100 × 100 пікселів або послідовність з 256 біт. Нейросеть з математичної точки зору поводиться як звичайна функція, хоч і дуже складно влаштована: у неї є заздалегідь позначене число аргументів, а також позначений формат, в якому вона видає відповідь. Простий приклад – функція x2, вона приймає один аргумент і видає одне значення [5].

Якщо використовувати нейромережа для обробки тексту або музики – будь–який умовно нескінченної послідовності, в якій важливо не тільки зміст, а й порядок, в якому слід інформація. Ось для цих завдань і були придумані рекурентні нейромережі. Їх протилежності, які ми називали «звичайними», мають більш суворе назва – нейромережі прямого поширення (feed–forward neural networks), так як в них інформація передається тільки вперед по мережі,

від шару до шару. У рекурентних нейромереж нейрони обмінюються інформацією між собою: наприклад, до того ж до нового шматочку входять даних нейрон також отримує деяку інформацію про попередньому стані мережі. Таким чином в мережі реалізується «пам'ять», що принципово змінює характер її роботи і дозволяє аналізувати будь–які послідовності даних, в яких важливо, в якому порядку йдуть значення – від звукозаписів до котирувань акцій.

Схема однойшаровой рекуррентной нейронної мережі: на кожному циклі роботи внутрішній шар нейронів отримує набір вхідних даних X і інформацію про попередньому стані внутрішнього шару A, на підставі чого генерує відповідь h.

Наявність пам'яті у зворотних нейромереж дозволяє дещо розширити нашу аналогію з x2. Пам'ять рекурентних нейромереж (хоча і не повноцінна, але про це пізніше) робить їх Тьюринг–повними: при правильному завданні ваг нейросеть може успішно емулювати роботу комп'ютерних програм.

Ймовірно, першою РНМ була мережа Хопфілда (вперше згадана в 1974 році, остаточно оформилася в 1982–му), яка реалізовувала на практиці осередок асоціативної пам'яті. Від сучасних РНМ вона відрізняється тим, що працює з послідовностями фіксованого розміру. У найпростішому випадку мережа Хопфілда має один шар внутрішніх нейронів, пов'язаних між собою, а кожна зв'язок характеризується певною вагою, що задає її значимість. З такою мережею асоціюється якийсь еквівалент фізичної «енергії», який залежить від всіх ваг в системі. Мережа можна навчити за допомогою градієнтного спуску по енергії, коли мінімум відповідає стану, в якому мережу «запам'ятала» певний шаблон, наприклад 10101. Тепер, якщо їй на вхід подати спотворений, зашумленний або неповний шаблон, скажімо, 10000, вона «згадає» і відновить його аналогічно тому, як працює асоціативна пам'ять у людини. Ця аналогія досить віддалених, тому не варто сприймати її надто серйозно. Проте, мережі Хопфілда успішно справлялися зі своїм завданням і обходили за можливостями існуючі тоді перцептрони.

Проблема довгостроковій пам'яті в простих РНМ: чим більше циклів пройшло з моменту отримання тієї чи іншої інформації, тим більша ймовірність, що значимість цих даних не буде грати великої ролі на новому циклі роботи.

Наступним кроком в еволюції РНМ була «проста рекуррентная мережу» Джеффа Елмана, описана в 1990 році. Якщо є вхідні дані 1100 і 0110, чи можна їх вважати одним і тим же набором, зсунутим в часі? Звичайно, можна, але як навчити цьому нейромережа? Звичайний перцептрон легко запам'ятає цю закономірність для будь–яких прикладів, які йому запропонують, але кожен раз це буде завданням порівняння двох різних сигналів, а не завданням про еволюцію або зсуві одного і того ж сигналу. Рішення Елмана, засноване на попередніх напрацювань в цій галузі, грунтувалося на тому, що в просту нейромережу додавався ще один – «контекстний» – шар, в який просто копіювалося стан внутрішнього шару нейронів на кожному циклі роботи мережі. При цьому зв'язок між контекстним і внутрішнім шарами можна було

навчати. Така архітектура дозволяла порівняно легко відтворювати тимчасові ряди, а також обробляти послідовності довільної довжини, що різко відрізняло просту РНМ Елмана від попередніх концепцій. Більш того, ця мережа змогла розпізнати і навіть класифікувати іменники і дієслова в реченні, грунтуючись тільки на порядку слів, що було справжнім проривом для свого часу і викликало величезний інтерес як лінгвістів, так і фахівців з дослідження свідомості.

За простій РНМ Елмана пішли всі нові розробки, а в 1997 році Хохрейтер

«Long Short–term memory»

і Шмідхубер опублікували статтю («довгострокова короткострокова пам'ять», також існує безліч інших варіацій перекладу), що заклав основу для більшості сучасних РНМ. У своїй роботі автори описували модифікацію, вирішувати проблему довгостроковій пам'яті простих РНМ: їх нейрони добре «пам'ятають» недавно отриману інформацію, але не мають можливості надовго зберегти в пам'яті щось, що обробили багато циклів назад, якою б важливою та інформація не була. У LSTM–мережах внутрішні нейрони «обладнані» складною системою так званих воріт (gates), а також концепцією клітинного стану (cell state), яка і являє собою якийсь вид довгострокової пам'яті. Ворота ж визначають, яка інформація потрапить в клітинне стан, яка зітреться з нього, і яка вплине на результат, який видасть РНМ на даному етапі. Детально розглянуто LSTM не буде, однак відзначити, що саме ці варіації РНМ використовується зараз, наприклад, для машинного перекладу Google.

Всеж чому музику РНМ якось пишуть, а з повноцінними текстами Толстого і Достоєвського виникають проблеми? Справа в тому, що в інструментальній музиці, як би по–варварськи це не звучало, немає сенсу в тому ж значенні, в якому він є в більшості текстів. Тобто музика може подобатися або не подобатися, але якщо в ній немає слів – вона не містить інформаційного навантаження (звичайно, якщо це не секретний код). Саме з доданням своїм творам сенсу і спостерігаються проблеми у РНМ: вони можуть чудово вивчити граматику мови і запам'ятати, як повинен виглядати текст в певному стилі, але створити і донести якусь ідею або інформацію РНМ (поки) не можуть.

Схема тривимірної рекуррентной нейромережі для написання музичних фрагментів: на відміну від найпростішої архітектури, в даній системі фактично об'єднані дві РНМ, окремо описують послідовність в часі і поєднання нот в кожен момент.

Особливий випадок в цьому питанні – це автоматичне написання програмного коду. Дійсно, оскільки мова програмування по визначенню є мова, РНМ може його вивчити. На практиці виявляється, що програми, написані РНМ, цілком успішно компілюються і запускаються, проте вони не роблять нічого корисного, якщо їм заздалегідь не позначити завдання. А причина цього та ж, що і в разі літературних текстів: для РНМ мову програмування – не більше ніж стилізація, в яку вони, на жаль, не можуть вкласти ніякого сенсу.

Зрозуміло, РНМ, крім розважальних, повинні переслідувати і більш прагматичні цілі. З їх дизайну автоматично випливає, що головні області їх застосування повинні бути вимогливі до контексту і або тимчасової залежності в даних, що по суті одне і те ж. Тому РНМ використовуються, наприклад, для аналізу зображень. Ця область зазвичай сприймається в контексті згортальних нейромереж, однак і для РНМ тут знаходяться завдання: їх архітектура дозволяє швидше розпізнавати деталі, грунтуючись на контексті і оточенні. Аналогічним чином РНМ працюють в сферах аналізу і генерації текстів. З більш незвичайних завдань можна згадати спроби використовувати ранні РНМ для класифікації вуглецевих спектрів ядерного магнітного резонансу різних похідних бензолу, а з сучасних – аналіз появи негативних відгуків про товари.

Успіхи РНМ в машинному перекладі поточний момент в Google, де для машинного перекладу використовуються РНМ типу LSTM, що дозволило домогтися найбільшої точності в порівнянні з існуючими аналогами, проте, за словами самих авторів, машинному перекладу ще дуже далеко до рівня людини. Складнощі, з якими стикаються нейромережі в задачах перекладу, обумовлені відразу декількома факторами: по–перше, в будь–якому завданні існує неминучий розмін між якістю і швидкістю. На даний момент людина дуже сильно випереджає штучний інтелект за цим показником. Оскільки машинний переклад найчастіше використовується в онлайн–сервісах, розробники змушені жертвувати точністю на вигоду швидкодії. У недавній публікації Google на цю тему розробники детально описують багато рішень, які дозволили оптимізувати поточну версію Google Translate, однак проблема досі залишається. Наприклад, рідкісні слова, або сленг, або навмисне спотворення слова (наприклад, для більш яскравого заголовка) може збити з пантелику навіть перекладача–людини, якій доведеться витратити час, щоб підібрати найбільш адекватний аналог в іншій мові. Машину ж така ситуація поставить в глухий кут, і перекладач буде змушений «викинути» складне слово і залишити його без переклада. У підсумку проблема машинного перекладу не настільки обумовлена архітектурою (РНМ успішно справляються з рутинними завданнями в цій галузі), наскільки складністю і різноманіттям мови. Радує те, що ця проблема має більш технічний характер, ніж написання осмислених текстів, де, ймовірно, потрібно кардинально новий підхід.

Принцип роботи РНМ типу LSTM: нейрони внутрішніх шарів можуть зчитувати і змінювати стан осередку (cell state), яке поєднує в собі функції короткостроковій і довгостроковій пам'яті.

Одна з головних областей застосування РНМ на сьогоднішній день – робота з мовними моделями, зокрема – аналіз контексту і загальної зв'язку слів в тексті. Для РНМ структура мови – це довгострокова інформація, яку треба запам'ятати. До неї відносяться граматика, а також стилістичні особливості того корпусу текстів, на яких проводиться навчання. Фактично РНМ запам'ятовує, в якому порядку зазвичай йдуть слова, і може дописати

пропозицію, отримавши деяку приманку. Якщо ця приманка випадкова, може вийти зовсім безглуздий текст, стилістично нагадує шаблон, на якому вчилася РНМ. Якщо ж вихідний текст був осмисленим, РНМ допоможе його стилізувати, проте в останньому випадку однієї РНМ буде мало, так як результат повинен являти собою «суміш» випадкового, але стилізованого тексту від РНМ і осмисленої, але «неокрашенной» вихідної частини. Це завдання вже настільки нагадує популярні нині програми для обробки фотографій в стилі Моне і Ван Гога, що мимоволі напрошується аналогія.

Дійсно, завдання перенесення стилю з одного зображення на інший вирішується за допомогою нейромереж і операції згортки, яка розбиває зображення на кілька сфер зовнішньої та дозволяє нейромереж аналізувати їх незалежно один від одного, а згодом і перемішувати між собою. Аналогічні операції проводилися і з музикою (також за допомогою згортальних нейромереж): в цьому випадку мелодія є змістом, а аранжування – стилем. І ось з написанням музики РНМ якраз успішно справляється. Оскільки обидва завдання – і написання, і змішування мелодії з довільним стилем – вже успішно вирішені за допомогою нейромереж, поєднати ці рішення залишається справою техніки.

Нейронна машина Тьюринга (Neural Turing Machine), запропонована два роки тому колективом з Google DeepMind, відрізняється від інших РНМ тим, що останні насправді не зберігають інформацію в явному вигляді – вона кодується в вагах нейронів і зв'язків, навіть в просунутих варіаціях на прикладі LSTM. У нейронної машині Тьюринга розробники дотримувалися більш зрозумілою ідеї «стрічки пам'яті», як у класичній машині Тьюринга: в ній інформація в явному вигляді записується «на стрічку» і може бути зчитана в разі потреби. При цьому відстеження того, яка інформація потрібна, лягає на особливу нейросеть–контролер. В цілому можна відзначити, що ідея НМТ дійсно зачаровує своєю простотою і доступністю для розуміння. З іншого боку, в силу технічних обмежень сучасного апаратного забезпечення застосувати НМТ на практиці не представляється можливим, тому що навчання такої мережі стає надзвичайно довгим. У цьому сенсі РНМ є проміжною ланкою між більш простими нейромережами і НМТ, так як зберігають якийсь «зліпок» інформації, який при цьому не смертельно обмежує їх швидкодія.

Концепція уваги (attention) – це спосіб «підказати» мережі, на що слід витратити більше уваги при обробці даних. Іншими словами, увагу в рекуррентной нейронної мережі – це спосіб збільшити важливість одних даних в порівнянні з іншими. Оскільки людина не може видавати підказки кожен раз (це нівелювало б всю користь від РНМ), мережа повинна навчитися підказувати собі сама. Взагалі, концепція уваги є дуже сильним інструментом в роботі з РНМ, так як дозволяє швидше і якісніше підказати мережі, на які дані варто звертати увагу, а на які – ні. Також цей підхід може в перспективі вирішити проблему швидкодії в системах з великим об'ємом пам'яті. Щоб краще зрозуміти, як це працює, треба розглянути дві моделі уваги: «м'яку» (soft) і «жорстку» (hard). У першому випадку мережа все одно звернеться до

всіх даних, до яких має доступ, але значимість (тобто вага) цих даних буде різною. Це робить РНМ більш точною, але не більш швидкою. У другому випадку з усіх існуючих даних мережу звернеться лише до деяких (у інших будуть нульові ваги), що вирішує відразу дві проблеми. Мінусом «жорсткої» концепції уваги є той факт, що ця модель перестає бути безперервною, а значить – диференціюється, що різко ускладнює завдання її навчання. Проте, існують рішення, що дозволяють виправити цей недолік. Оскільки концепція уваги активно розвивається в останні пару років, нам залишається чекати найближчим часом новин з цього поля.

Під кінець можна навести приклад системи, що використовує концепцію уваги: це Dynamic Memory Networks – різновид, запропонована дослідним підрозділом Facebook. У ній розробники описують «модуль епізодичній пам'яті» (episodic memory module), який на підставі пам'яті про події, заданих у вигляді вхідних даних, а також питання про ці події, створює «епізоди», які в підсумку допомагають мережі знайти правильну відповідь на питання. Така архітектура була випробувана на bAbI, великої базі згенерованих завдань на простий логічний висновок (наприклад, дається ланцюжок з трьох фактів, потрібно видати правильну відповідь: «Мері будинку. Вона вийшла на подвір'я. Де Мері? На подвір'ї».), І показала результати, що перевершують класичні архітектури на кшталт LSTM.

Висновки за розділом

1. Типова нейронна мережа представляє собою від кількох десятків до сотень, тисяч або навіть мільйонів штучних нейронів, які називаються одиницями, розташованими за серією шарів, кожен з яких з'єднується з шаром з обох сторін. Деякі з них, відомі як одиниці введення, призначені для отримання різноманітних форм інформації з боку зовнішнього світу, які мережа спробує сприйняти, розпізнати чи інакше обробити. Інші підрозділи знаходяться на протилежній стороні мережі та сигналізують, як вона відповідає отриманій інформації; вони відомі як вихідні одиниці. Між пристроями вводу та вихідними блоками є один або декілька шарів прихованих блоків.

2. З'єднання між однією одиницею та іншою представлені значенням, яке називається вагою, яка може бути позитивною (якщо одна одиниця збуджує іншу) або негативною (якщо одна одиниця пригнічує або інгібує іншу). Чим вище вага, тим більше впливає одна одиниця на іншу. (Це відповідає тому, як фактичні клітини мозку провокують одна одну через зв’язки, що мають назву синапси.) Динамічна зміна значень ваги в процесі навчання нейромережі дозволяє «обучити» нейронну мережу. Поняття навчання досить багатофакторне, так як включає залежність від архітектури мережі, навчального набору і практичних підборів найбільш вдалих варіантів реалізації залежно від поставленої задачі. Фундаментальним для нейромережевих систем є зворотній зв'язок, що поширюється від вихідних елементів, які відправляють данні для корекції ваги попередніх слоїв нейронів.

2. ГОТОВІ ПЛАТФОРМИ ДЛЯ РОЗРОБКИ НЕЙРОМЕРЕЖ

2.1 Платформні та апаратні рішення від NVIDIA

Продовжуючи працювати над однією з найскладніших технологічних завдань сучасності, Nvidia створили новий програмно–апаратний комплекс, який забезпечує безпрецедентну швидкість, легкість і обчислювальну потужність для завдань глибокого навчання (deep learning).

Глибоке навчання – швидко розвивається сегмент області знання, пов'язаної зі штучним інтелектом, – є каталізатором розвитку самих різних галузей, починаючи від медицини і фармакології і закінчуючи автокеруємих автомобілях.

17 березня 2015, під час виступу перед 4000 учасників конференції по графічним технологіям (GTC) генеральний директор і співзасновник NVIDIA Дженсен Хуанг (Jen–Hsun Huang) представив три нових технології, які стануть потужним стимулом розвитку області глибокого навчання:

NVIDIA GeForce GTX TITAN X – найпотужніший графічний процесор з коли–небудь створених для швидкого навчання глибоких нейронних мереж;

DIGITS Deep Learning GPU Training System – програмне забезпечення, яке дозволяє вченим і дослідникам набагато швидше і легше створювати глибокі нейронні мережі.

DIGITS DevBox – найшвидше в світі спеціалізоване рішення для роботи з завданнями глибокого навчання, побудоване на базі чотирьох TITAN X GPU, що поставляється з попередньо встановленою інтуїтивно зрозумілою системою навчання DIGITS.

TITAN X – новий флагман лінійки ігрових графічних процесорів сімейства GeForce – також ідеально підходить і для завдань глибокого навчання. На конференції GDC (Game Developer Conference) графічні процесори TITAN X використовувалися для створення віртуальної реальності в демонстрації "Злодії в тіні" ("Thief in the Shadows").

Побудований на графічній архітектурі NVIDIA Maxwell, TITAN X має вдвічі більшу продуктивність та енергоефективність в порівнянні з попередником. TITAN X – це 12ГБ пам'яті і 3072 ядра, які забезпечують 7 терафлопс в обчисленнях одинарної точності.

Завдяки цій мощі і смузі пропускання пам'яті 336.5 ГБ / с, нове рішення легко справляється з обробкою мільйонів даних, що використовуються для навчання глибоких нейронних мереж. Так, TITAN X знадобилося менше трьох

днів для навчання мережі AlexNet за допомогою набору з 1.2 мільйона зображень ImageNet, тоді як 16–ядерний CPU справляється з цим завданням за 40 днів.

Щоб навчити комп'ютери класифікувати і розпізнавати об'єкти за допомогою глибоких нейронних мереж, потрібно дуже багато часу. Програмне забезпечення DIGITS Deep Learning GPU Training System вирішує цю проблему, надаючи користувачам все, що потрібно, для побудови глибоких нейронних мереж. Перша комплексна графічна система для створення, навчання і валідації глибоких нейронних мереж, призначених для класифікації зображень.

DIGITS бере на себе важкі завдання, допомагаючи користувачам налаштувати, конфігурувати і навчити глибокі нейронні мережі, щоб вчені могли зосередитися безпосередньо на дослідженнях і результатах.

Інтуїтивно зрозумілий інтерфейс і можливості управління DIGITS забезпечують простоту підготовки та згодовування тренувальних наборів даних, як на локальній системі, так і з мережі.

Це перша система в своєму роді з підтримкою моніторингу і візуалізації в реальному часі, що дозволяє користувачам тонко налаштувати свою роботу. Вона також підтримує прискорену на GPU версію Caffe, популярного пакету програм, який широко застосовується сьогодні вченими і дослідниками для створення нейронних мереж.

Створена командою інженерів NVIDIA, система DIGITS DevBox є ядром комплексної платформи для прискорення досліджень глибокого навчання. Кожен компонент DevBox, починаючи з чотирьох карт GPU TITAN X і закінчуючи пам'яттю і інтерфейсами, максимально оптимізований, щоб забезпечити найбільш ефективну роботу для найскладніших завдань глибокого навчання.

Дана система поставляється з попередньо встановленим програмним забезпеченням, яке необхідно вченим і дослідників для створення власних глибоких нейронних мереж. У список додатків входять пакет програм DIGITS, найпопулярніші платформи глибокого навчання Caffe, Theano і Torch, а також cuDNN 2.0 – GPU– прискоренна бібліотека для задач глубокого навчання від NVIDIA [11].

Найперші результати такого многопроцессорного навчання показують, що DIGITS DevBox забезпечує продуктивність майже в чотири рази вище в порівнянні з одним TITAN X в тестах глибокого навчання. За допомогою DIGITS DevBox натренувати мережу AlexNet можна всього за 13 годин, тоді як звичайному ПК на базі найшвидшого GPU треба було б більше двох діб, а системі на базі CPU – більше місяця [13].

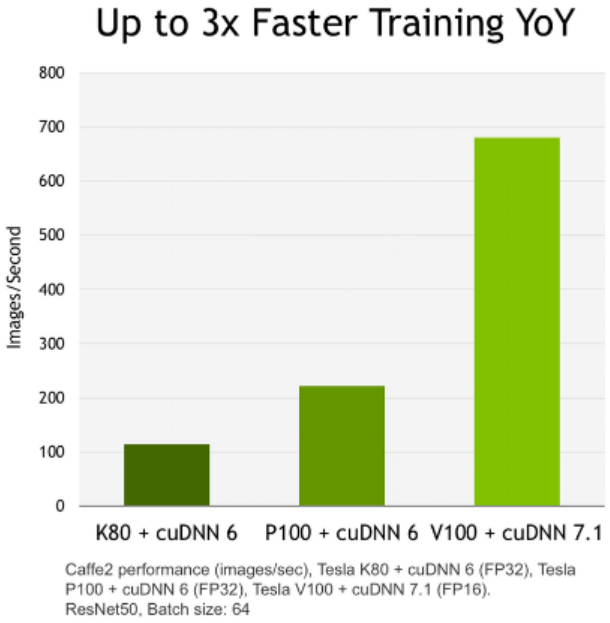

Рисунок 2.1 – Швидкість навчання для Caffe2 залежно від бібліотеки cuDNN

2.2 Система машинного навчання TensorFlow

TensorFlow є системою машинного навчання Google Brain другого покоління, випущеною як відкрите програмне забезпечення 9 листопада 2015 року. TensorFlow швидша, розумніша і гнучкіша в порівнянні з попередньою технологією Google (DistBelief, з 2011, та сама, що розпізнавала кішку без вчителя), завдяки чому стало значно простіше адаптувати її до використання в нових продуктах і дослідницьких проектах. TensorFlow – високомасштабована система машинного навчання, здатна працювати як на простому смартфоні, так і на тисячах вузлів в центрах обробки даних. Використовується TensorFlow для всього спектру завдань, від розпізнавання мови до автовідповідача в Inbox і пошуку в Google Photos. Така гнучкість дозволяє конструювати і тренувати нейромережі до 5 разів швидше в порівнянні з попередньою платформою, так що можливо дійсно використовувати технологію значно оперативніше. Обчислення TensorFlow виражаються як станові графи потоків даних. Назва TensorFlow походить від операцій, що такі нейронні мережі виконують над багатовимірними масивами даних. Ці багатовимірні масиви називають «тензорами».

В той час як еталонна реалізація працює на одиничних пристроях, TensorFlow може працювати на декількох ЦП та ГП (включно з додатковими розширеннями CUDA для обчислень загального призначення на графічних процесорах). TensorFlow доступна для 64–розрядних Linux, macOS, Windows, та для мобільних обчислювальних платформ, включно з Android та iOS

.

Використовуючи нову систему «прямо з цеху», стало зрозуміло, що вона здатна надати ще більший вплив за межами корпорації. Тому Google Brain відкрила TensorFlow для всіх розробників в open source, що це дозволило всьому співтовариствам навколо машинного навчання, всім – від дослідників до інженерів і просто любителів – обмінюватися ідеями ефективніше, за допомогою реально працюючого коду, а не шляхом одних лише дослідницьких статей. Для покрашення використовується зворотний зв'язок, щоб прискорити наші дослідження в області машинного навчання, в результаті зробивши цю технологію краще для всіх. Він також може бути корисний у всіх випадках, коли дослідники шукають зміст і сенс в дуже складноструктурованих даних – усюди від білкових згорток в біоінформатики до аналізу астрономічних таблиць. У той час як попередня технологія була заточена на нейронні мережі і нашу внутрішню особливості, TensorFlow досить узагальнений і більш ефективний.

В червні 2016 року Джефф Дін з Google заявив, що TensorFlow згадували 1 500 репозиторіїв на GitHub, лише 5 з яких були від Google [12].

2.3 Microsoft Azure–Machine Learning Studio

Написання ML – систем з нуля, використовуючи C # або будь–яку іншу мову програмування, – займає багато часу, що вимагає спеціалізованих знань і часто важко. Microsoft Azure ML Studio (випущена в липні 2014 г.) значно спрощує і прискорює створення ML–систем, а також робить їх більш ефективними [15].

Застосування ML Studio для створення ML–систем приблизно аналогічно використанню Visual Studio для створення виконуваних програм, хоча не слід занадто сильно покладатися на цю аналогію. Щоб задіяти Visual Studio, ви можете або купити її, або завантажити безкоштовну пробну версію. У разі ML Studio з вас стягують плату за користування сервісом, але потім ще з'являться варіанти, що дозволяють безкоштовно ознайомитися з цією системою.

ML Studio може зчитувати дані безпосередньо з Web або з Azure Storage, але я віддаю перевагу створювати власне сховище даних.

Модель ML можна розглядати як набір інформації (як правило, числових значень, званих ваговими), які використовуються для генерації виведення і прогнозів. Навчання моделі – це процес пошуку набору таких вагових значень, щоб, коли вони подаються разом з вхідними даними з навчального набору, обчислені вихідні значення близько відповідали відомим вихідним значенням в навчальних даних. Після визначення цих вагових значень кінцева отримана модель може бути представлена за допомогою перевірочних даних. Точність моделі на перевірочних даних (відсоток правильних прогнозів) дає вам приблизну оцінку того, наскільки добре модель буде працювати з новими даними, де справжній висновок не відомий.

Оскільки висновок для прогнозування має два можливих значення, потрібна двійкова модель. Але до ML існують десятки підходів, і, ймовірно,

найважча частина у використанні ML Studio полягає в дослідженні всіх за і проти різних класифікаторів ML. Аналогія для розробника з Visual Studio – в Microsoft.NET Framework є десятки структур даних, таких як узагальнена Dictionary, HashSet і узагальнена Queue, обов'язоко знати, що саме робить кожна структура даних. Точно так само потрібно вивчити класифікатори ML.

У модуля Logistic Regression є деякі параметри, які ви навряд чи відразу зрозумієте; до них відносяться, в тому числі, Optimization tolerance, L1 regularization weight і Memory size for L–BFGS. І знову розбиратися з цими параметрами ви будете самостійно. На щастя, ML Studio пропонує якісно підібрані значення за замовчуванням для більшості параметрів модулів. Я погодився з усіма значеннями за замовчуванням для параметрів, крім одного: для Random number seed я задав нульове значення.

Додаток Azure ML Studio спільно зі своїм серверним механізмом, сервісом Microsoft Azure Machine Learning (ML), є набагато більшим, ніж простий клієнтський інструмент. З точки зору розробника, ML Studio радикально спрощує створення систем прогнозування, але вона пропонує додаткові можливості, не настільки очевидні на перший погляд.

Одна з важливих областей, – здатність Azure ML створювати та публікувати веб–сервіси простим перетягуванням і кілька разів клацнувши. Azure ML автоматично обробляє розгортання, підготовку пропускної здатності, балансування навантаження, автоматичне масштабування і моніторинг працездатності. Один з клієнтів, які використовували попередню версію, оцінив, що за допомогою Azure ML його компанія змогла створити бізнес–рішення (систему виявлення спроб шахрайства), витративши на нього лише малу частку того, у що обходиться використання комерційного аналітичного ПО.

Azure ML підтримує R – популярна мова програмування в області наукових досліджень даних. Сотні існуючих R–модулів з відкритим вихідним кодом можна безпосередньо копіювати в систему Azure ML.

ML Studio забезпечує також колективну роботу. Експеримент можна легко поділити між декількома людьми. це набагато ефективніше, ніж звичайне обговорення по електронній пошті.

Azure ML дає переваги не тільки розробникам і дослідникам в області даних. «Розгортання просунутих засобів аналітики – справа важка, – сказав Джозеф Сайрош (Joseph Sirosh), віце–президент Microsoft по зв'язках з корпораціями в галузі управління розробки аналітичних моделей і їх розгортання без цих вузьких інформацією і машинного навчання. – На підприємствах втомилися платити високу ціну, наймати дорогі таланти і місяцями чекати результатів. Можливість швидкої місць змінює правила гри в цілій галузі. Azure ML дозволяє підприємствам розкрити цінність своїх даних і будувати системи, які зменшують витрати, підвищують доходи і краще служать своїм кінцевим клієнтам »

.

Одна з найцікавіших функцій ML Studio – можливість написання власних модулів на C #.

Можливо, що ML Studio збере навколо себе екосистему, в якій розробники пишуть витончені, спеціалізовані модулі і роблять їх доступними як на комерційній основі, так і по різних каналах, таким як проекти з відкритим вихідним кодом і блоги.

2.4 Бібліотека PyTorch

TensorFlow і Keras мають API на Python, все ще важко з'ясувати, що саме пішло не так у випадку помилки. Вони також погано працюють разом з бібліотеками numpy, scipy, scikit-learn, Cython і іншими. Бібліотека глибокого навчання PyTorch має заявлені переваги - добре працюють з Python і створена для апологетів Python. Крім того, приємне властивість PyTorch - побудова обчислювального динамічного графа, протилежно статичним обчислювальним графам, представленим в TensorFlow і Keras.

Перше дійсно PyTorch краще TensorFlow, суб'єктивно, так як з точки зору продуктивності немає великих відмінностей. У будь-якому випадку, PyTorch став серйозним суперником у змаганні між бібліотеками глибокого навчання. []

Головні ідеї, які стоять за PyTorch є тензори і обчислювальні графи, закінчуючи змінними класами і функціональністю автоматичного диференціювання.

Для ОС Windows, на веб-сайті PyTorch немає опції для простої установки бібліотеки для цієї операційної системи. Однак існують спеціальні інструкції по налаштуванню.

Перше, що необхідно зрозуміти про будь-яку бібліотеку глибокого навчання - ідею обчислювальних графів. Обчислювальний граф - набір обчислень, які називаються вузлами (nodes), і які з'єднані в прямому порядку обчислень. Іншими словами, обраний вузол залежить від вузлів на вході, який в свою чергу виробляє обчислення для інших вузлів. Нижче представлений простий приклад обчислювального графа для обчислення виразу a = (b + c) * (c + 2). Можна розбити обчислення на наступні кроки:

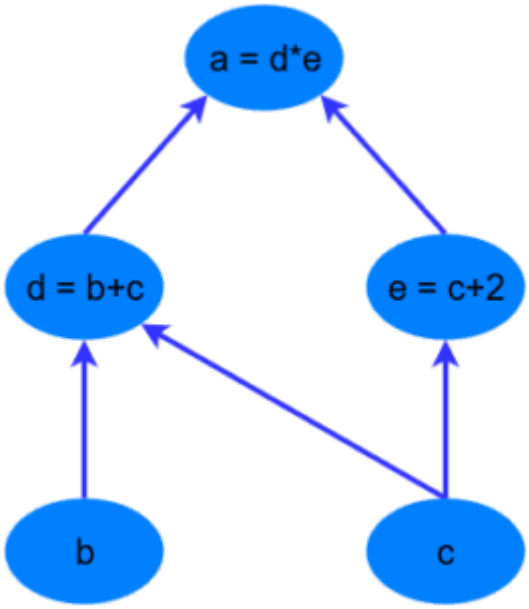

Рисунок 2.2 - Простий обчислювальний граф

Переваги використання обчислювального графа в тому, що кожен вузол є незалежним функціонуючим шматком коду, якщо отримає всі необхідні вхідні дані. Це дозволяє оптимізувати продуктивність при виконанні розрахунків, використовуючи багатоканальну обробку, паралельні обчислення. Всі основні фреймворки для глибокого навчання (TensorFlow, Theano, PyTorch і так далі) включають в себе конструкції обчислювальних графів, за допомогою яких виконуються операції всередині нейронних мереж і відбувається зворотне поширення градієнта помилки.

Тензори - подібні матриці структури даних, які є невід'ємними компонентами в бібліотеках глибокого навчання і використовуються для ефективних обчислень. Графічні процесори (GPU) ефективні при обчисленні операцій між тензорами, що стимулювало хвилю можливостей в глибокому навчанні. У PyTorch тензори можуть визначатися декількома способами:

import torch

x = torch.Tensor (2, 3)

Цей код створює тензор розміру (2,3), заповнений нулями. В даному прикладі перше число - кількість рядів, друге - кількість стовпців:

0 0 0

0 0 0

[Torch.FloatTensor of size 2x3]

Можна створити тензор, заповнений випадковими float-значеннями:

x = torch.rand (2, 3)

Множення тензорів, складання один з одним і інші операції алгебри

x = torch.ones (2,3)

y = torch.ones (2,3) * 2
x + y
Код повертає:

3 3 3
3 3 3
[Torch.FloatTensor of size 2x3]
Також доступна робота з функцією slice в numpy. Наприклад y [:, 1]:
y [:, 1] = y [:, 1] + 1
Яка повертає:

2 3 2
2 3 2
[Torch.FloatTensor of size 2x3]

Відображено основну концепцію створювання тензорів і роботи з ними в PyTorch.

У бібліотеках глибокого навчання є механізми обчислення градієнта помилки і зворотного поширення помилки через обчислювальний граф. Цей механізм, званий автоградієнтом в PyTorch, легко доступний і інтуїтивно зрозумілий. Змінний клас - головний компонент автоградієнтной системи в PyTorch. Змінний клас обгортає тензор і дозволяє автоматично обчислювати градієнт на тензор при виконанні функції .backward (). Об'єкт містить дані з тензора, градієнт тензора (один раз порахований по відношенню до деякого іншого значення, втрата) і містить також посилання на будь-яку функцію, створену змінної (якщо це функція створена користувачем, посилання буде порожній).

В оголошенні змінної використовується подвійний тензор розміру 2x2 і додатково вказується, що змінної необхідний градієнт. При використанні цієї змінної в нейронних мережах, вона стає здатна до навчання. Якщо останній параметр буде дорівнює False, то змінна не може використовуватися для навчання. У цьому простому прикладі ми нічого не будемо тренувати, але хочемо запросити градієнт для цієї змінної, як буде показано нижче.

Створення нової змінної на основі x.
z = 2 * (x * x) + 5 * x

Щоб обчислити градієнт цієї операції по x, dz / dx, можна аналітично отримати 4x + 5. Якщо всі елементи x - двійки, то градієнт dz / dx - тензор розмірності (2,2), заповнений числами 13. Однак спочатку необхідно запустити операцію зворотного поширення .backwards (), щоб обчислити градієнт щодо чого-небудь. У нашому випадку ініціалізується одиничний тензор (2,2), щодо якого вважаємо градієнт. В такому випадку обчислення - просто операція d / dx:

z.backward (torch.ones (2, 2))
print (x.grad)
Результатом є наступне:
Variable containing:

13 13
13 13
[Torch.FloatTensor of size 2x2]

Важливо що градієнт зберігається в змінної x в властивості .grad. Виконано найпростіші операцій з тензорами, змінними і функцією автоградієнта в PyTorch.

2.5 Апаратне прискорення навчання нейронних мереж

Глибоке навчання – це поле з інтенсивними обчислювальними вимогами, і вибір GPU принципово визначить ефективність навчання. Якщо ж немає графічного процесора, це процес навчання виглядає як очікування завершення експерименту до місяця або проведення експерименту протягом дня чи більше, лише щоб виключити вибрані параметри. З потужним графічним процесором можна швидко перейти до глибоких навчальних мереж і запускати експерименти за дні замість місяців, годин замість днів, хвилин замість годин тому, головним є покупка графічного процесора.

Наявність швидкого графічного процесора – це дуже важливий аспект, коли проводиться розробка глибокого вивчення, оскільки це дозволяє швидко отримати практичний досвід, щоб застосовувати глибоке навчання для вирішення нових проблем. Без швидкого зворотного зв'язку навчання займає надто багато часу, щоб вчитися на своїх помилках, і це може сповільнювати процес розробки.

Розпаралелити нейронні мережі на кількох графічних процесорах не тільки дуже важко, але і прискорення посереднє для щільних нейронних мереж. Малі нейронні мережі можуть бути досить ефективно паралелізовані за допомогою паралелізму даних, але більша кількість нейронних мереж майже не отримують прискорення.

При оптимізації паралельних алгоритмів з оптимізованим кодуванням для ряду завдань паралельність декількох графічних процесорів не працює належним чином. Потрібний високий рівень розуміння апаратного забезпечення і як воно взаємодіє з глибокими алгоритмами навчання, щоб визначити, чи можливо скористатися паралелізацією в першу чергу.

Для практичних замірів використовувався комп'ютер з трьома GXT Titan та карту InfiniBand. Підтримка паралелізму для графічних процесорів є більш поширеним, але все ще далека від загальнодоступного та ефективного. Єдина глибока навчальна бібліотека, яка в даний час реалізує ефективні алгоритми на різних GPU та комп'ютерах, – це CNTK, який використовує спеціальні алгоритми розпаралелювання Microsoft з 1–бітним квантуванням (ефективним) та імпульсом блокування (дуже ефективним). З CNTK і кластером з 96 графічними процесорами можна очікувати нову лінійну швидкість близько 90x–95x. При меті паралелізувати на одній машині, то варіанти в основному CNTK, Torch, Pytorch. Ця бібліотека забезпечує хорошу

швидкість (3.6x–3.8x) і попередньо визначені алгоритми паралелізму на одній машині з частотою до 4 графічних графічних процесорів. Існують і інші бібліотеки, які підтримують паралелізм, але вони або повільні (наприклад, TensorFlow з 2x–3x) або важко використовувати для декількох графічних процесорів.

Якшо паралелізм є головною метою, то варто використовувати Pytorch або CNTK.

Використання декількох графічних процесорів без паралелізму

Ще однією перевагою використання кількох графічних процесорів, навіть якщо не розпаралелювати алгоритми, полягає в тому, що можливо запускати кілька алгоритмів або експериментів окремо на кожному графічному процесорі. Пофакту не отримується прискорення, але отримується більше інформації про продуктивність, використовуючи різні алгоритми або параметри одночасно. Це дуже корисно, якщо головна мета полягає в тому, щоб якнайшвидше отримати глибокий досвід навчання, а також дуже корисно для дослідників, які хочуть одночасно спробувати кілька версій нового алгоритму.

Чим коротші інтервали для виконання задачі та отримання зворотного зв'язку для завдання, тим краще людина здатна інтегрувати відповідні частини пам'яті для цього завдання в узгоджену картину. Якщо навчаються дві згорткові мережі на окремих графічних процесорах на малих наборах даних, можливо ще краще розібратися у тому, що важливо для успішної роботи; легко можливо виявляти шаблони у помилці перевірки на перехресті та правильно їх інтерпретувати. Більші можливості виявити шаблони, які дадуть підказки про те, який параметр або шар потрібно додати, видалити чи скорегувати.

Таким чином, загалом можна сказати, що одного графічного процесора повинно бути достатньо для практично будь–якого завдання, але багато GPU стає все більш важливим для прискорення навчання глибоких моделей. Кілька дешевих графічних процесорів також відмінні, для швидкого вивчення глибокого навчання.

Стандартні бібліотеки NVIDIA дозволяють легко встановити перші глибокі бібліотеки для вивчення в CUDA, тоді як для OpenCL не було таких потужних стандартних бібліотек. На даний момент, для карт пам'яті виробнка AMD немає хороших глибоких навчальних бібліотек серед інших популярних – це NVIDIA. Навіть якщо деякі бібліотеки OpenCL будуть доступні в майбутньому, перспективніше дотримуватися NVIDIA: справа в тому, що обчислення GPU чи GPGPU є значно швидшими для CUDA і досить невеликим для OpenCL [17].

Крім того, NVIDIA пішла по шляху розвитку у відношенні глибокого навчання, хоча глибоке навчання було саме в початку широкого поширення. Ця ставка окупилася. В той час, як інші компанії зараз ставлять гроші та зусилля позаду глибокого навчання, вони все ще дуже відстають через їх пізній початок. В даний час використання будь–якої програмно–апаратної

комбінації для глибокого навчання, відмінного від NVIDIA–CUDA, призведе до серйозних розчарувань.

У випадку з Intel Xeon Phi пропонується щоб використовувати стандартний код C і легко перетворити цей код у прискорені коди Xeon Phi. Ця функція може здатися досить цікавою, тому що покладається на величезні ресурси C–коду. Однак насправді підтримуються лише дуже маленькі частини C–коду, так що ця функція насправді не є корисною, і більшість частин C–коду, які будуть запущені, будуть повільними.

Також особливості які підривають продуктивність, як розміри тензора, на яких працює мережа, змінюються послідовно. Наприклад, якщо у вас є повністю підключені шари різних розмірів або відсівні шари, Xeon Phi працює повільніше, ніж центральний процесор. Отже, Xeon Phis, не найкраще рішення для вивчення та розробок.

Найважливішою функцією для продуктивності глибокого навчання є пропускна спроможність пам'яті.

GPU оптимізовано для смуги пропускання пам'яті, при цьому приносячи в жертву час доступу до пам'яті (латентність). Процесори розробляють точно на протилежному рівні: процесори можуть виконувати швидкі обчислення, якщо задіяні невеликі обсяги пам'яті, наприклад множення кількох чисел (3 * 6 * 9), але для операцій на великих обсягах пам'яті, таких як множення матриць (A * B * C) вони повільніші. Графічні процесори відрізняються від проблем, пов'язаних із великою кількістю пам'яті через пропускну здатність пам'яті. Тому при виборі швидкого графічного процесора, передусім звертати увагу на пропускну здатність цього графічного процесора.

Порівняння пропускної здатності процесорів та графічних процесорів за часом: пропускна спроможність є однією з основних причин швидшого обчислення графічних процесорів, ніж процесорів.

Пропускна здатність може бути безпосередньо порівняна в рамках архітектури, наприклад, продуктивність карт Pascal, таких як GTX 1080 або GTX 1070, можна безпосередньо порівняти, переглядаючи їх пропускну здатність пам'яті. Наприклад, GTX 1080 (320 Гбіт / с) результативніша близько 25% (320/256) швидше, ніж GTX 1070 (256 Гбіт / с). Проте в архітектурі, як наприклад, Паскаль або Максвелл, як GTX 1080 або GTX Titan X, не можна порівнювати безпосередньо через те, що різні архітектури з різними процесами виготовлення (в нанометрах) використовують задану пропускну здатність пам'яті по–різному. В цілому це ускладнює порівння, але загальна смуга пропускання дає хороший огляд того, наскільки швидко GPU приблизно [16].

Однак важливим фактором є те, що не всі архітектури сумісні з cuDNN. Оскільки майже всі бібліотеки deep learning використовують cuDNN для згорткових операцій, це обмежує вибір графічних процесорів на GPU Kepler або краще, тобто серії GTX 600 або вище. Крім того, GPU Kepler, як правило, досить повільно. Отже, це означає, що перевага повинна віддаватися графічним процесорам із серії 900 або 1000 для швидкої роботи.

Для того, щоб дати приблизну оцінку того, як карти працюють по відношенню один до одної при глибокому навчання, побудована схема еквівалентності графічного процесора на рисунку 2.2.. Наприклад, одна GTX 980 настільки ж швидка, як 0.35 Titan X Pascal, або іншими словами, Titan X Pascal майже втричі швидший, ніж GTX 980.

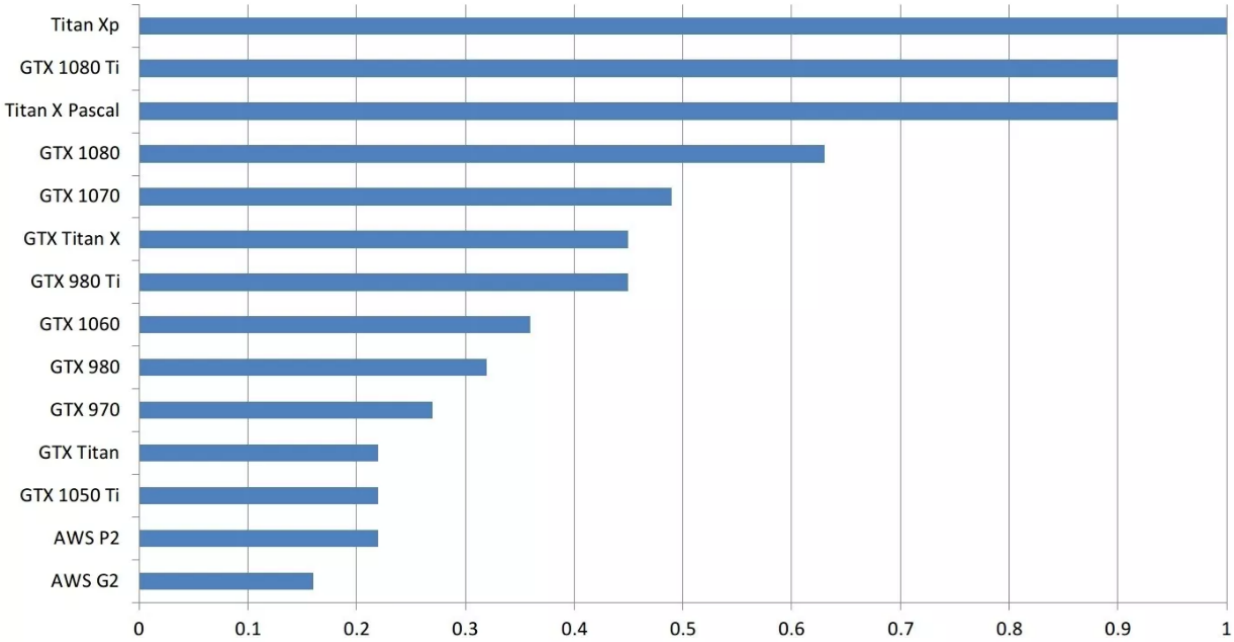

Рисунок 2.3 – Відносне порівняння продуктивності між GPU при великих обчисленнях

Порівняння виводяться з порівнянь специфікацій карт разом з обчислювальними критеріями (деякі випадки видобутку криптовалют – завдання, які в обчисленні порівнянні з глибоким вивченням). Отже, це приблизні оцінки. Реальні числа можуть дещо відрізнятися, але зазвичай помилка повинна бути мінімальною і порядок карт має бути правильним. Також зауважте, що невеликі мережі, які не використовують GPU, змусять більші графічні процесори виглядати не ефективними. Наприклад, невелика LSTM (128 прихованих пристроїв; розмір партії> 64) на GTX 1080 Ti не буде набагато швидше, ніж запустити її на GTX 1070. Щоб отримати різницю в продуктивності на графіку, потрібно запустити більшу мережу, скажімо LSTM з 1024 прихованими одиницями (і розміром пакету> 64). Це також важливо мати на увазі під час вибору графічного процесора, який підходить для поставленої задачі.

Міра ранжування графічних процесорів не враховує об'єм пам'яті GPU. Часто потрібно більше пам'яті, ніж у GTX 1050 Ti, і таким чином, при економічній ефективності, деякі з високопоставлених карток не є практичними рішеннями. Аналогічним чином, складніше використовувати 4 невеликих GPU, а не 1 великий GPU, і, отже, малі GPU мають невигідне становище. Крім того, не практично придбати 16 GTX 1050 Ti, щоб отримати продуктивність 4 GTX 1080 Ti, також доведеться придбати 3 додаткових комп'ютера, що є дорогим. Якщо взяти останню точку врахування, то діаграма виглядає так на рисунку 2.4.

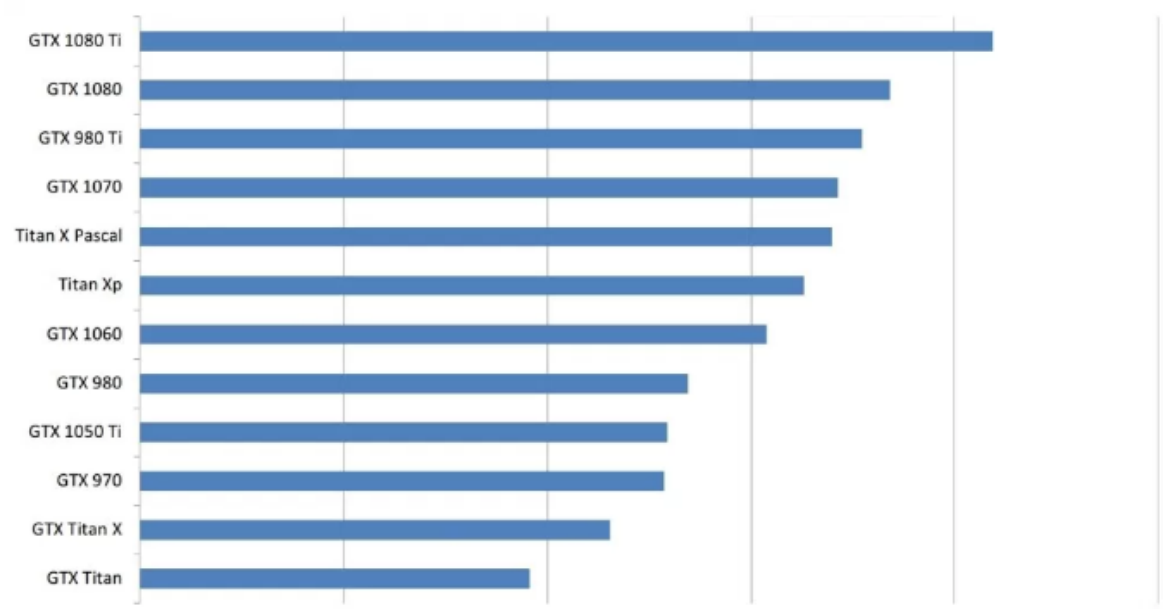

Рисунок 2.4 – Економічна ефективність GPU з враховуванням ціни іншого обладнання.

Порівнюється повноцінна машина, тобто 4 GPU, а також високотехнологічне обладнання (процесор, материнська плата тощо) вартістю 1500 доларів.

Покупка багатьох слабших графічних процесорів значно невигідніша за продуктивністю, ніж потужних GPU, хоча це дешевше, якщо купувати більш економічно ефективні комбінації комп'ютерів та GPU (а не просто економічно ефективні графічні процесори). Неважливо, наскільки економічно ефективно 4 GTX 1080 Ti у коробці, якщо є фінансові обмеження, і немає можливості дозволити таке рішення в першу чергу. Також повинно розглядатися перспективи масштабування надалі або заміни графічного процесора, можливості продажі та ефективність завантаження GPU.

2.6 API нейронних мереж Keras

Keras – це високорівневий API нейронних мереж, написаний на Python і здатний працювати з TensorFlow, CNTK або Theano. Він був розроблений з урахуванням можливості швидкого експерименту. Можливість перейти від ідеї до результату з мінімальною затримкою – це ключ до успішного дослідження. Монтаж API виконується командою на рисунку 2.5.

```
C:\Windows\system32>pip install keras
```

Рисунок 2.5 – Установка API Keras

Використання глибокої бібліотеки навчання Keras, яка характеризується:

— можливістю легко і швидко створювати прототипи (за допомогою зручності, модульності та розширюваності);

— підтримка як згорткові мережі, так і періодичні мережі, а також комбінації обох;

— працює без проблем на процесорі та графічному процесорі;

— Keras сумісний з: Python 2.7–3.6.

Зручність у використанні. Keras – API, призначений для людей, а не машин. Це робить користувальницький досвід фронтом та центром. Keras дотримується найкращих практик зменшення когнітивного навантаження: він пропонує послідовні та прості API, мінімізує кількість дій користувача, необхідних для звичайних випадків використання, і забезпечує чіткий та дієвий відгук щодо помилок користувача.

Модульність. Модель розуміється як послідовність чи графік автономних, повністю налаштовуваних модулів, які можна підключити разом із якомога меншими обмеженнями. Зокрема, нервові шари, функції витрат, оптимізатори, схеми ініціалізації, функції активації, схеми регуляризації – це автономні модулі, які можна поєднати для створення нових моделей.

Легко розширювана. Нові модулі легко додавати (як нові класи та функції), а існуючі модулі наводять достатньо прикладів. Можливість легко створювати нові модулі дозволяє тотальну виразність, що робить Keras придатним для передових досліджень.

Робота з Python. Немає окремих файлів конфігурації моделей у декларативному форматі. Моделі описані в коді Python, який компактний, простий у налагодженні та дозволяє легко розширювати [15].

Висновки за розділом

1. Розробка нейронних мереж з нуля неоправдано складно, якщо взяти до уваги велику кількість платформ то API розроблених для розробки та навчання нейронних мереж.

2. З боку програмного забезпечення були розробленні потужні готові рішення компаніями Microsoft та Nvidia такі як Azure ML, бібліотеки Cuda. Головний принцип розробки платформ та бібліотек було поширення використання машинного навчання, спрощення вхідного порогу створення нових рішень. Розглядаючи з точки зору розробника, ML Studio автоматично обробляє розгортання, підготовку пропускної здатності, балансування навантаження, автоматичне масштабування і моніторинг працездатності радикально спрощує створення систем прогнозування. Використання серверного механізму, що надає можливість створювати та публікувати веб–сервіси завдяки простому інтерфейсу.

3. Важливим фактором для навчання та тестування моделей нейронних мереж – підримка на апаратному рівні GPU. Необхідність використання cuDNN 2.0 – GPU– прискоренна бібліотека для задач глубокого навчання від NVIDIA, для масштабних проектів дозволяє задіяти потужність графічного прискорювача.

4. Апаратна частина надзвичайно важлива і правильно вибрати яку нелегко. Згідно даних практичних замірень виграш не дають поєднання багатьох дешевших відеокарт проти меншої кількості дорожчих, за умови складності реалізації програмного розпаралелення. Таке твердження справедливе для великих проектів і найефективніше апаратне забезпечення для розробки ШНМ стануть відеокарти 900 та 1000 серії GTX компанії Nvidia.

3. ОГЛЯД ПЛАТФОРМИ TensorFlow

3.1 Налаштування платформи TensorFlow

Існує багато API–інтерфейсів TensorFlow, за порадою розробників розглянуться такі концепцій використання TensorFlow:

— монтування і запуск середовище виконання програми;

— імпортувати дані з API Datasets;

— побудувати моделі та шари з API Keras TensorFlow.

Буде розглянуто API серед інших програм TensorFlow:

— імпорт та розділення набору даних;

— вибір типу моделі;

— тренування моделі;

— оцінка ефективність моделі;

— використання навченої моделі для прогнозування.

Для моделі використовуватиметься ««прискорене виконання»», яке доступне в TensorFlow 1.8., тому для роботи потрібно встановити останню версію TensorFlow командою на рисунку 3.1.

```
C:\Windows\system32>pip install -q --upgrade tensorflow
```

Рисунок 3.1 – Команда для установки TensorFlow

Потім необхідно імпортувати модулі Python, включаючи TensorFlow, і дозволити «пришвидшене виконання» цієї програми. «Пришвидшене виконання» робить TensorFlow розрахункові операцій миттєвими, повертаючи конкретні значення, замість того, щоб створювати графік обчислень, який виконується пізніше.

Якщо поставити завдання для автоматичного класифікування кожну квітку ірису, наприклад. Машине навчання забезпечує безліч алгоритмів статистичного класифікації квітів. Наприклад, складна програма машинного навчання може класифікувати квіти на основі фотографій. Для спрощення програма буде класифікувати ірисові квіти на основі вимірювань довжини і ширини їх черепів та пелюсток.

Генетична родова Ірису тягне за собою близько 300 видів, але програма класифікує лише наступні три:

1) Iris setosa
2) Iris virginica
3) Iris versicolor

Використовуватиметься готовий набір даних з 120 ірисових квітів з вимірами сепла і пелюстки. Це класичний набір даних, популярний для початкових проблем класифікації комп'ютерного навчання [20].

«Швидке виконання» TensorFlow – це імперативне середовище програмування, яке негайно оцінює операції без побудови графіків: операції повертають конкретні значення замість побудови графіку обчислень для запуску пізніше. Це полегшує початок роботи з TensorFlow та налагоджувальними моделями, а також зменшує завантаження.

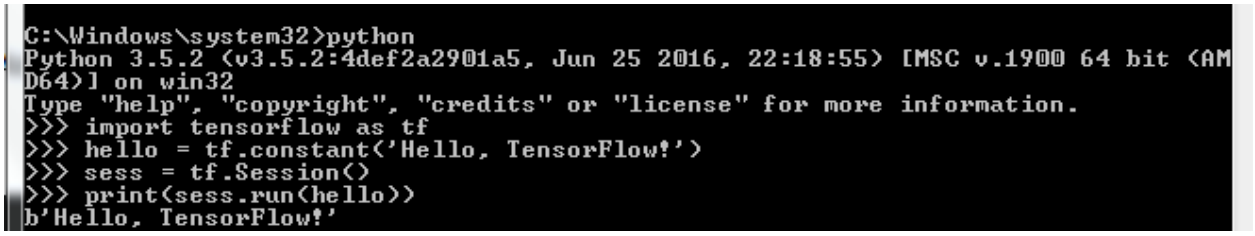

Рисунок 3.2 – Перевірка коректнсті роботи TensorFlow

```
>>> from __future__ import absolute_import, division, print_function
>>> import os
>>> import matplotlib.pyplot as plt
>>> import tensorflow as tf

Warning (from warnings module):
  File "C:\Users\Роман\AppData\Local\Programs\Python\Python35\lib\site-packages\h
5py\__init__.py", line 36
    from ._conv import register_converters as _register_converters
FutureWarning: Conversion of the second argument of issubdtype from `float` to `n
p.floating` is deprecated. In future, it will be treated as `np.float64 == np.dty
pe(float).type`.
>>> import tensorflow.contrib.eager as tfe
>>> tf.enable_eager_execution()
>>> print("TensorFlow version: {}".format(tf.VERSION))
TensorFlow version: 1.8.0
>>> print("Eager execution: {}".format(tf.executing_eagerly()))
Eager execution: True
>>> train_dataset_url = "http://download.tensorflow.org/data/iris_training.csv"
>>> train_dataset_fp = tf.keras.utils.get_file(fname=os.path.basename(train_datas
et_url),origin=train_dataset_url)
>>> print("Local copy of the dataset file: {}".format(train_dataset_fp))
Local copy of the dataset file: C:\Users\Роман\.keras\datasets\iris_training.csv
```

Рисунок 3.3 – Завантаження даних у форматі CSV

Оскільки набір даних являє собою текстовий файл, відформатований у форматі CSV, проаналізуємо значення функції та мітки у форматі, який може використовувати реалізуєму модель Python. Кожен рядок або рядок у файлі передаються до функції parse_csv, яка захоплює перші чотири поля функцій і об'єднує їх у єдиний тензор. Потім останнє поле аналізується як мітка. Функція повертає як критерії так і ярлики тензорів:

```
>>> def parse_csv(line):
        example_defaults = [[0.], [0.], [0.], [0.], [0]]  #
        parsed_line = tf.decode_csv(line, example_defaults)
        features = tf.reshape(parsed_line[:-1], shape=(4,))
        label = tf.reshape(parsed_line[-1], shape=())
        return features, label
```

Рисунок 3.4 – Опис функції parse_csv

API TensorFlow Dataset API обробляє багато поширених випадків передачі даних у модель. Це API високого рівня для читання даних та перетворення його у форму, яка використовується для навчання.

Програма використовує tf.data.TextLineDataset для завантаження текстового файлу у форматі CSV і аналізується за допомогою функції parse_csv на рисунку 3.4. Tf.data.Dataset являє собою вхідний потік як сукупність елементів і серію перетворень, що діють на ці елементи. Методи трансформації об'єднуються або об’являються послідовно – потрібно переконатися, що є посилання на об'єкт, який був повернутий.

Тренування найкраще, якщо приклади розташовані в довільному порядку. Використано tf.data.Dataset.shuffle для рандомізації записів, встановлюючи buffer_size до значення, більшого за кількість прикладів (у цьому випадку 120). Щоб тренування модель пришвидшити, розмір пакетного набору набору становить 32 приклади для одночасного тренування на рисунку 3.5.

```
>>> train_dataset = tf.data.TextLineDataset(train_dataset_fp)
>>> train_dataset = train_dataset.skip(1)
>>> train_dataset = train_dataset.map(parse_csv)
>>> train_dataset = train_dataset.shuffle(buffer_size=1000)
>>> train_dataset = train_dataset.batch(32)
>>> features, label = iter(train_dataset).next()
>>> print("example features:", features[0])
example features: tf.Tensor([5.8 2.8 5.1 2.4], shape=(4,), dtype=float32)
>>> print("example label:", label[0])
example label: tf.Tensor(2, shape=(), dtype=int32)
```

Рисунок 3.5 – Рандомізація записів та визначення розміру набору даних

Модель – це зв'язок між функціями та міткою. Для проблеми класифікації ірисів модель визначає зв'язок між вимірами чашелистиків і пелюстками та прогнозованим видом ірису. Деякі прості моделі можна описати кількома рядками алгебри, але складні моделі машинного навчання мають велику кількість параметрів, які важко узагальнити.

Якщо використовувати традиційні методи програмування (наприклад, багато умовних висловлювань) для створення моделі, проаналізувати набір даних достатньо довго, щоб визначити зв'язок між вимірюваннями пелюстки та сепла до певного виду. І це важко для традиційного програмування може бути неможливим – на більш складні набори даних. Хороший підхід до машинного навчання визначає модель для вас. Якщо подати достатньо

репрезентативних прикладів у правильному типу моделі машинного навчання, програма створить зв’язки.

3.2 Підбір моделі

Потрібно вибрати тип навчання для тренувань. Існує багато типів моделей, і вибір хорошого – це практичний досвід. Нейромережі можуть знайти складні зв'язки між функціями та міткою. Це графічно структурований графік, який складається з одного або декількох прихованих шарів. Кожен прихований шар складається з одного або декількох нейронів. Є кілька категорій нейронних мереж, і програма використовує щільну або повнозвязну нейронну мережу: нейрони в одному шарі отримують вхідні з'єднання з кожного нейрона в попередньому шарі. Наприклад, на рисунку показана щільна нейронна мережа, що складається з вхідного шару, двох прихованих шарів і вихідного шару:

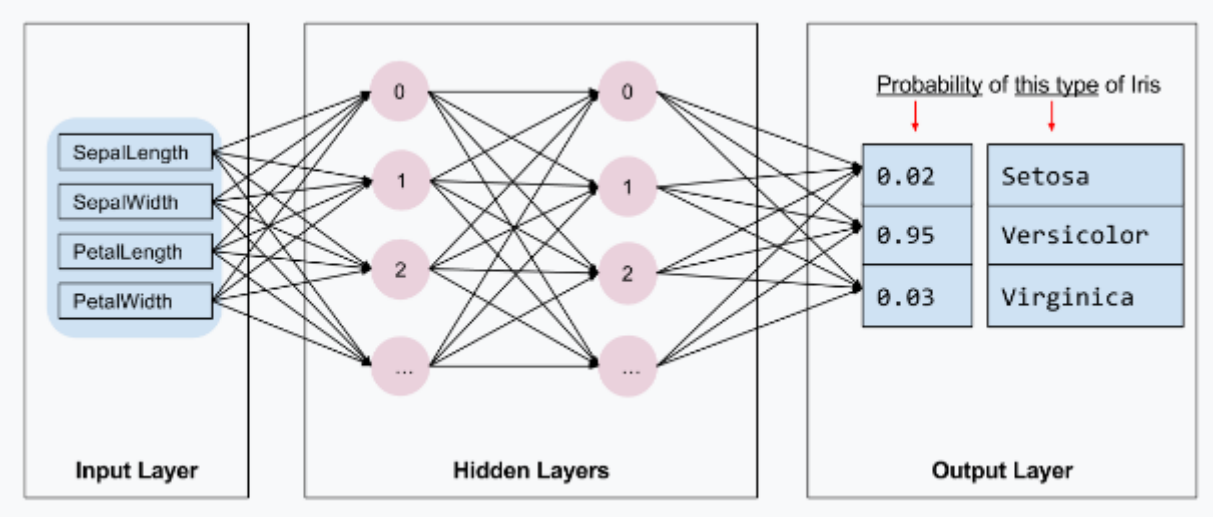

Рисунок 3.6 – Щільна нейрона мережа

Коли модель з малюнка навчається і подає немаркований приклад, вона дає три прогнози: вірогідність того, що ця квітка – це дана назва ірису. Це передбачення називається висновком. Для цього прикладу сума вихідних прогнозів становить 1,0. На малюнку цей прогноз розбивається на: 0,03 для Iris setosa, 0,95 для Iris versicolor та 0,02 для Iris virginica. Це означає, що модель передбачає, з 95% ймовірністю, що незамічена приклади квітка – це версікол Iris versicolor.

API TensorFlow tf.keras – це найкращий спосіб створити моделі та шари. Це дозволяє легко створювати моделі та експериментувати, а Keras обробляє складність об'єднання всього. Tf.keras.Sequential модель – лінійний стек шару. Його конструктор приймає список шарів–екземплярів, в цьому випадку два щільні шари з кожними 10 вузлами і вихідний шар з 3 вузлами, що представляють передбачувані мітки. Параметр input_shape першого шару

відповідає кількості функцій з набору даних і є обов'язковим.

```
>>> model = tf.keras.Sequential([
        tf.keras.layers.Dense(10, activation="relu", input_shape=(4,)),
        tf.keras.layers.Dense(10, activation="relu"),
         tf.keras.layers.Dense(3)
        ])
```

Рисунок 3.7 – Tf.keras.Sequential модель лінійного стеку шару

Функція активації визначає вихід одного нейрона на наступний шар. Це вільно залежить від того, як пов'язані нейронні слої. Є багато доступних активацій, але ReLU є загальним для прихованих шарів.

Ідеальне число прихованих шарів і нейронів залежить від проблеми та набору даних. Як і багато аспектів машинного навчання, підбір найкращої форми нейронної мережі вимагає суміші знань та експериментів. Як правило, збільшення кількості прихованих шарів і нейронів зазвичай створює більш потужну модель, для якої потрібні додаткові дані для ефективного навчання.

Навчання є етапом машинного навчання, коли модель поступово оптимізується, або модель вивчає набір даних. Мета – достатньо вивчити структуру навчального набору для прогнозування невидимих даних. Якщо занадто багато «навчити» на набірі навчальних програм, то прогнози працюють лише на даних, які він бачив, і їх не можна буде узагальнити. Ця проблема називається перенавчанням – це як запам'ятовування відповідей, а не розуміння того, як вирішити проблему.

Проблема класифікації ірисів – це приклад наглядового машинного навчання: модель навчається з прикладів, що містять мітки. У режимі безумовного машинного навчання приклади не містять міток. Замість цього модель зазвичай знаходить шаблони серед функцій.

На етапах підготовки та оцінки необхідно розрахувати втрату моделі. Це вимірює, наскільки передбачені прогнози моделі від потрібної мітки, іншими словами, наскільки некоректно працює модель. В ідеальному випадку звести до мінімуму або оптимізувати це значення.

Модель обчислює свою втрату за допомогою функції tf.losses.sparse_softmax_cross_entropy, яка приймає передбачення моделі та потрібну мітку. Значення повернутої втрати поступово збільшується, оскільки прогноз погіршується.

```
>>> def loss(model, x, y):
        y_ = model(x)
        return tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_)

>>> def grad(model, inputs, targets):
  with tf.GradientTape() as tape:
    loss_value = loss(model, inputs, targets)
  return tape.gradient(loss_value, model.variables)
```

Рисунок 3.7 – Tf.keras.Sequential модель лінійного стеку шару

Функція grad використовує функцію втрат і tf.GradientTape для запису операцій, які обчислюють градієнти, використовувані для оптимізації моделі.

Оптимізатор застосовує обчислені градієнти до змінних моделі, щоб мінімізувати функцію втрат. Можна створити криву поверхню, і потрібно знайти найнижчу точку, проходячи по кривій. Градієнти вказують у напрямку стрімкого підйому, тому рух направлений навпаки вниз. По ітераційному розрахунку втрат і градієнта для кожної партії, коригується модель під час навчання. Поступово модель знайде найкраще поєднання ваг і зміщення, щоб мінімізувати втрати. І чим нижча втрата, тим кращі прогнози моделі.

TensorFlow має багато алгоритмів оптимізації, доступних для навчання. Ця модель використовує tf.train.GradientDescentOptimizer на рисунку 3.8, який реалізує алгоритм стохастичного градієнтного спуску (SGD). Learning_rate визначає розмір кроку для кожної ітерації вниз по пагорбу. Це гіперпараметр, який зазвичай коригуєтесь, щоб досягти кращих результатів.

```
>>> optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
```

Рисунок 3.8 – Реалізація алготму стохастичного градієтного спуску

3.3 Навчальний цикл

З усіма елементами на місці, модель готова до навчання. Навчальний цикл подає приклади набору даних у модель, щоб допомогти йому зробити кращі прогнози. Наступний кодовий блок встановлює такі етапи навчання:

1. Ітератувати кожну епоху. Епоха – це один прохід через набір даних.
2. Протягом однієї епохи повторити кожен приклад у навчальному наборі даних, захоплюючи його функції (x) та мітку (y).
3. Використовуючи функції прикладу, скласти прогноз і порівняти його з міткою. Виміряти неточність прогнозу та використати його для розрахунку втрат моделі та градієнтів.
4. Використати оптимізатор, щоб оновити змінні моделі.
5. Відстежувати деякі статистичні дані для візуалізації.
6. Повторити для кожної епохи.

Змінна num_epochs – це кількість разів, щоб перемістити колекцію набору даних. Контр–інтуїтивно, тренування моделі довше не гарантує кращої моделі. num_epochs – це гіперпараметр, який можна настроїти. Вибір правильного номера зазвичай вимагає як досвіду, так і експериментів.

Оцінка моделі схожа на навчання моделі. Найбільша різниця – приклади наведені з окремого тестового набору, а не навчального набору. Для точнішого оцінювання ефективність моделі, приклади, що використовуються для оцінки моделі, повинні відрізнятись від прикладів, які використовуються для навчання моделі.

```
>>> train_loss_results = []
>>> train_accuracy_results = []
>>> num_epochs = 201
>>> for epoch in range(num_epochs):
        epoch_loss_avg = tfe.metrics.Mean()
        epoch_accuracy = tfe.metrics.Accuracy()
        for x, y in train_dataset:
                grads = grad(model, x, y)
                optimizer.apply_gradients(zip(grads, model.variables),global_st
ep=tf.train.get_or_create_global_step())

>>> epoch_loss_avg(loss(model, x, y))
<tf.Tensor: id=73432, shape=(), dtype=float32, numpy=0.18478529>
>>> epoch_accuracy(tf.argmax(model(x), axis=1, output_type=tf.int32), y)
(<tf.Tensor: id=73449, shape=(24,), dtype=int32, numpy=
array([2, 2, 2, 2, 2, 0, 0, 1, 1, 2, 2, 0, 1, 1, 2, 2, 2, 0, 1, 0, 0, 1,
       0, 2])>, <tf.Tensor: id=73286, shape=(24,), dtype=int32, numpy=
array([2, 2, 1, 2, 1, 0, 0, 1, 1, 2, 2, 0, 1, 1, 2, 2, 2, 0, 1, 0, 0, 1,
       0, 2])>)
>>> train_loss_results.append(epoch_loss_avg.result())
>>> train_accuracy_results.append(epoch_accuracy.result())
>>> if epoch % 50 == 0:
        print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,epoch
_loss_avg.result(),epoch_accuracy.result()))

Epoch 200: Loss: 0.051, Accuracy: 98.333%
```

Рисунок 3.9 – Тренерування мережі протягом 200 епох

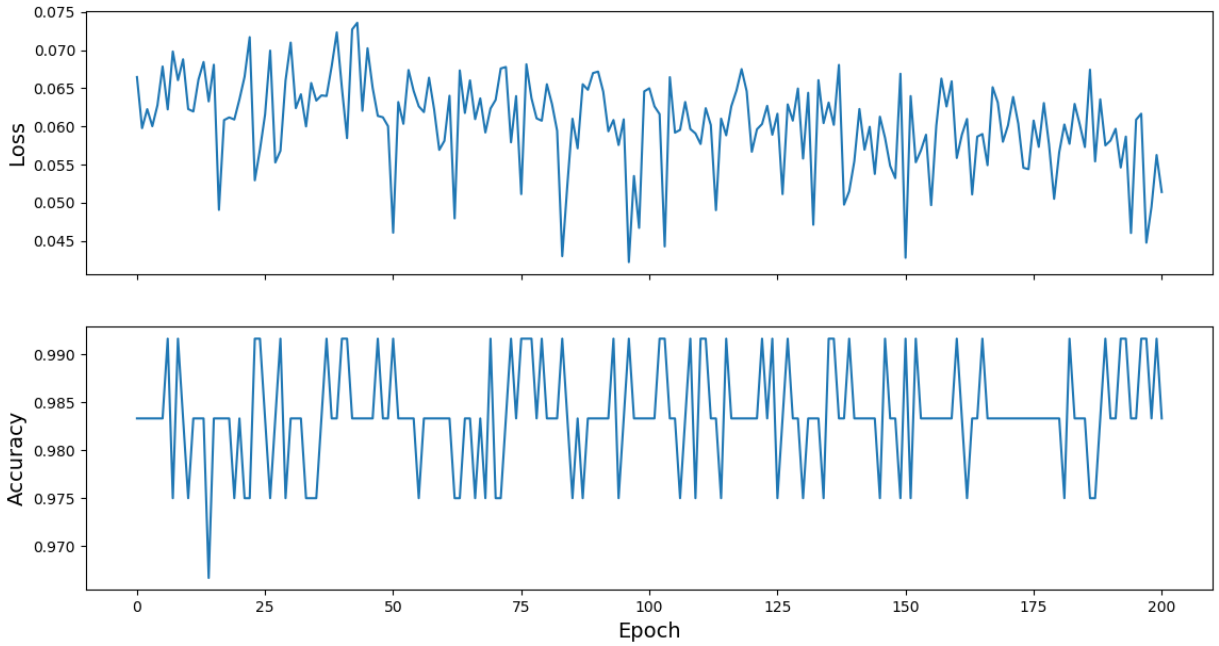

Рисунок 3.10 – Графічні дані результату

Налаштування тесту Dataset аналогічно налаштуванню для набору даних навчання на рисунку 3.11. Завантажити текстовий файл CSV та проаналізувати ці значення, а потім додати його трохи перемішеним.

```
>>> test_url = "http://download.tensorflow.org/data/iris_test.csv"
>>> test_fp = tf.keras.utils.get_file(fname=os.path.basename(test_url),origin=test_url)
Downloading data from http://download.tensorflow.org/data/iris_test.csv
8192/573 [==============================================================================
==============================================================================
==============================================================================
==============================================================================
==========================================] - 0s 0us/step
>>> test_dataset = tf.data.TextLineDataset(test_fp)
>>> test_dataset = test_dataset.skip(1)
>>> test_dataset = test_dataset.map(parse_csv)
>>> test_dataset = test_dataset.shuffle(1000)
>>> test_dataset = test_dataset.batch(32)
```

Рисунок 3.11 – Налаштування тесту Dataset

На відміну від етапу тренування, модель оцінює лише одну епоху даних тесту. У наступній коді повторюється над кожним прикладом у тестовому наборі та порівнюємо передбачення моделі з фактичною міткою. Це використовується для вимірювання точності моделі по всьому тестовому набору.

```
>>> test_accuracy = tfe.metrics.Accuracy()
>>> for (x, y) in test_dataset:
        prediction = tf.argmax(model(x), axis=1, output_type=tf.int32)
        test_accuracy(prediction, y)
        print("Test set accuracy: {:.3%}".format(test_accuracy.result()))


(<tf.Tensor: id=483769, shape=(30,), dtype=int32, numpy=
array([0, 2, 1, 1, 1, 0, 1, 0, 1, 2, 0, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1, 0,
       0, 2, 0, 1, 2, 1, 1, 0])>, <tf.Tensor: id=483754, shape=(30,), dtype=int32, numpy=
array([0, 2, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1, 0,
       0, 2, 0, 1, 2, 1, 1, 0])>)
Test set accuracy: 96.667%
>>> test_accuracy = tfe.metrics.Accuracy()
```

Рисунок 3.12 – Графічні дані результату

Після тренувань моделі і для класифікації видів ірису. Далі використано навчену модель, щоб зробити деякі прогнози на незамічених прикладах; тобто на прикладах, що містять функції, але не мітку.

На практиці непомічені приклади можуть надходити з різних джерел, включаючи додатки, файли CSV та канали даних. Для прикладу вручну надається три незамічені приклади для прогнозування їх міток. Де номери міток зв'язані з названим представленням як:

1) 0: Iris setosa
2) 1: Iris versicolor
3) 2: Iris virginica–

```
>>> class_ids = ["Iris setosa", "Iris versicolor", "Iris virginica"]
>>> predict_dataset = tf.convert_to_tensor([
    [5.1, 3.3, 1.7, 0.5,],
    [5.9, 3.0, 4.2, 1.5,],
    [6.9, 3.1, 5.4, 2.1]
])
>>> predictions = model(predict_dataset)
>>> for i, logits in enumerate(predictions):
  class_idx = tf.argmax(logits).numpy()
  name = class_ids[class_idx]
  print("Example {} prediction: {}".format(i, name))


Example 0 prediction: Iris setosa
Example 1 prediction: Iris versicolor
Example 2 prediction: Iris virginica
```

Рисунок 3.13 –Дані тестування

Висновки за розділом

1. TensorFlow – високомасштабована система машинного навчання, здатна працювати на різних платформах. Використання TensorFlow має широкий діапазон як розпізнання мови або вирішення завдань класифікації. Завдяки гнучкості системи, конструювання і навчання нейромереж значно пришвидшується.

2. Для кращого розуміння роботи TensorFlow було роглянуто приклад класифікації квітів ірису трьох окремих видів, тому що більша кількість видів значно ускладнює задачу. Як і багато аспектів машинного навчання, підбір найкращої форми нейронної мережі вимагає суміші знань та експериментів. Як правило, збільшення кількості прихованих шарів і нейронів зазвичай створює більш потужну модель, для якої потрібні додаткові дані для ефективного навчання. Основна концепція розпізнавання на отриманні прогнозу ймовірності, що вхідні дані відповідають конкретному виду ірису.

3. Завершаючим став етап тестування даних, що містять функцію але не мітку, тобто прикладах які не використовувалися при навчанні. Були побудовані графіки похибки які показували незначні величини через перенавчання мережі.

4 ОПТИМІЗАЦІЯ НАВЧАННЯ ШТУЧНИХ ДОСЛІДЖУВАНИХ НЕЙРОСТРУКТУР

4.1 Оцінка затрати ресурсів від кількості епох

Дослідження впливу кількості епох на якість навчання, було протестовано різні варіації параметра num_epochs. Під час навчання великої мережі буде момент, коли модель припинить узагальнювати і почне вивчати статистичний шум у навчальному наборі даних.

Це перевищення навчального набору даних призведе до збільшення помилки узагальнення, що зробить модель менш корисною для прогнозування нових даних.

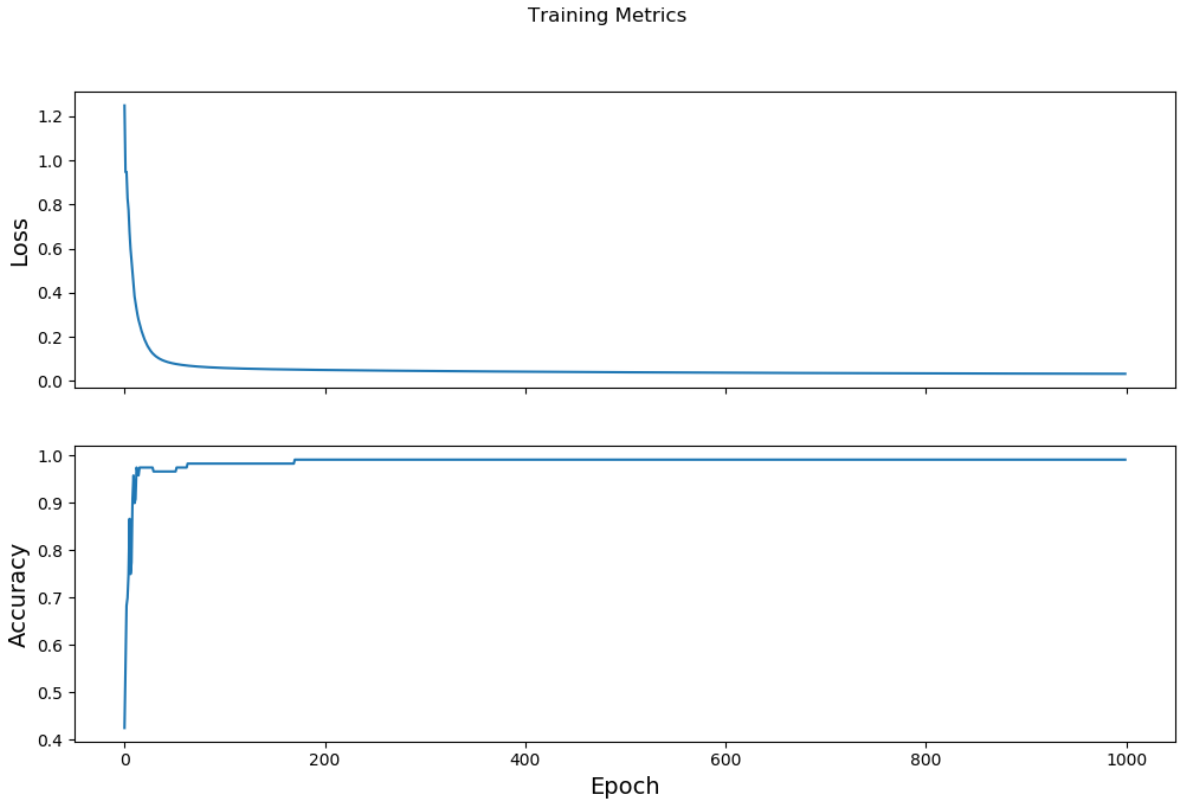

Рисунок 4.1– Дані точності та втрат для 1000 циклів навчання

```
Loss test: 1.2132184505462646
Step: 0, Initial Loss: 1.2132184505462646
Step: 1,         Loss: 1.1524553298950195
Epoch 000: Loss: 1.094, Accuracy: 51.667%
Epoch 050: Loss: 0.065, Accuracy: 98.333%
Epoch 100: Loss: 0.053, Accuracy: 98.333%
Epoch 150: Loss: 0.049, Accuracy: 98.333%
Epoch 200: Loss: 0.046, Accuracy: 99.167%
Epoch 250: Loss: 0.044, Accuracy: 99.167%
Epoch 300: Loss: 0.042, Accuracy: 99.167%
Epoch 350: Loss: 0.040, Accuracy: 99.167%
Epoch 400: Loss: 0.038, Accuracy: 99.167%
Epoch 450: Loss: 0.037, Accuracy: 99.167%
Epoch 500: Loss: 0.035, Accuracy: 99.167%
Epoch 550: Loss: 0.034, Accuracy: 99.167%
Epoch 600: Loss: 0.033, Accuracy: 99.167%
Epoch 650: Loss: 0.033, Accuracy: 99.167%
Epoch 700: Loss: 0.031, Accuracy: 99.167%
Epoch 750: Loss: 0.030, Accuracy: 99.167%
Epoch 800: Loss: 0.030, Accuracy: 99.167%
Epoch 850: Loss: 0.029, Accuracy: 99.167%
Epoch 900: Loss: 0.029, Accuracy: 99.167%
Epoch 950: Loss: 0.028, Accuracy: 99.167%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (100.0%)
Example 2 prediction: Iris virginica (100.0%)
```

Рисунок 4.2 – Тестові приклади для 1000 циклів навчання

Мета полягає в тому, щоб навчити нейрону мережу достатньо довго, щоб вона була здатна вивчити відображення від входів до результатів, але не тренувати модель настільки довго, щоб вона перевищувала дані навчання.

Один із підходів до вирішення цієї проблеми полягає в тому, щоб розглядати кількість навчальних епох як гіперпараметр і тренувати модель кілька разів з різними значеннями, а потім вибирати кількість епох, які призводять до найкращої ефективності в поїзді або набору даних про тестування.

Мінусом цього підходу є те, що він вимагає підготовки декількох моделей та їх відмови. Це може бути обчислювально неефективно і забирає багато часу, особливо для великих моделей, що навчаються на великих наборах даних протягом днів або тижнів.

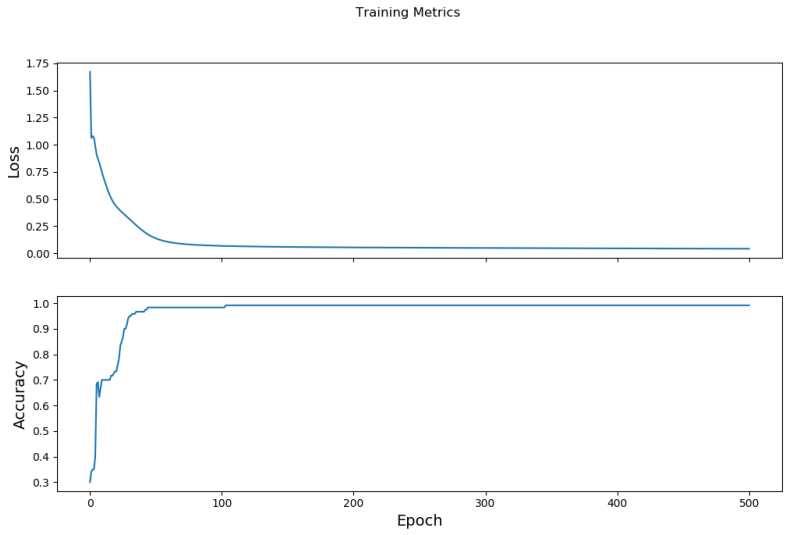

Рисунок 4.3 – Дані точності та втрат для 500 циклів навчання

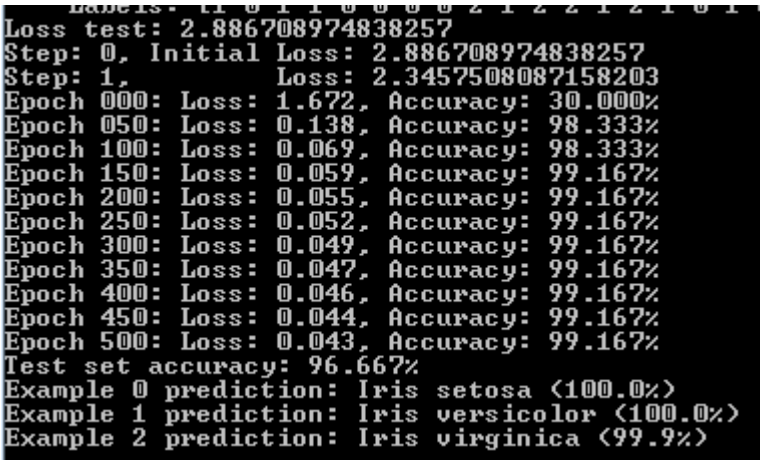

Рисунок 4.4 – Тестові приклади

Альтернативний підхід полягає в тому, щоб навчити модель одноразово для великої кількості навчальних епох.

Під час тренінгу модель оцінюється на наборі даних валідації проведення, після кожної епохи. Якщо продуктивність моделі на наборі даних перевірки починає погіршуватися (наприклад, втрати починають збільшуватися або точність починає знижуватися), тоді навчальний процес припиняється.

Похибка, виміряна щодо незалежних даних, зазвичай називається набором валідації, спочатку часто показує зменшення з подальшим збільшенням, коли мережа починає надмірно підходити. Тому навчання може бути припинено в точці найменшої помилки стосовно набору даних про валідацію. Потім використовується модель, коли припиняється навчання, і вона, як відомо, має хороші результати узагальнення.

Ця процедура називається «ранньою зупинкою» і є, мабуть, однією з найдавніших і найпоширеніших форм регуляризації нейронної мережі.

Ця стратегія відома як рання зупинка. Це, мабуть, найпоширеніша форма регуляризації в глибокому навчанні, популярність обумовлена як його ефективністю, так і простотою.

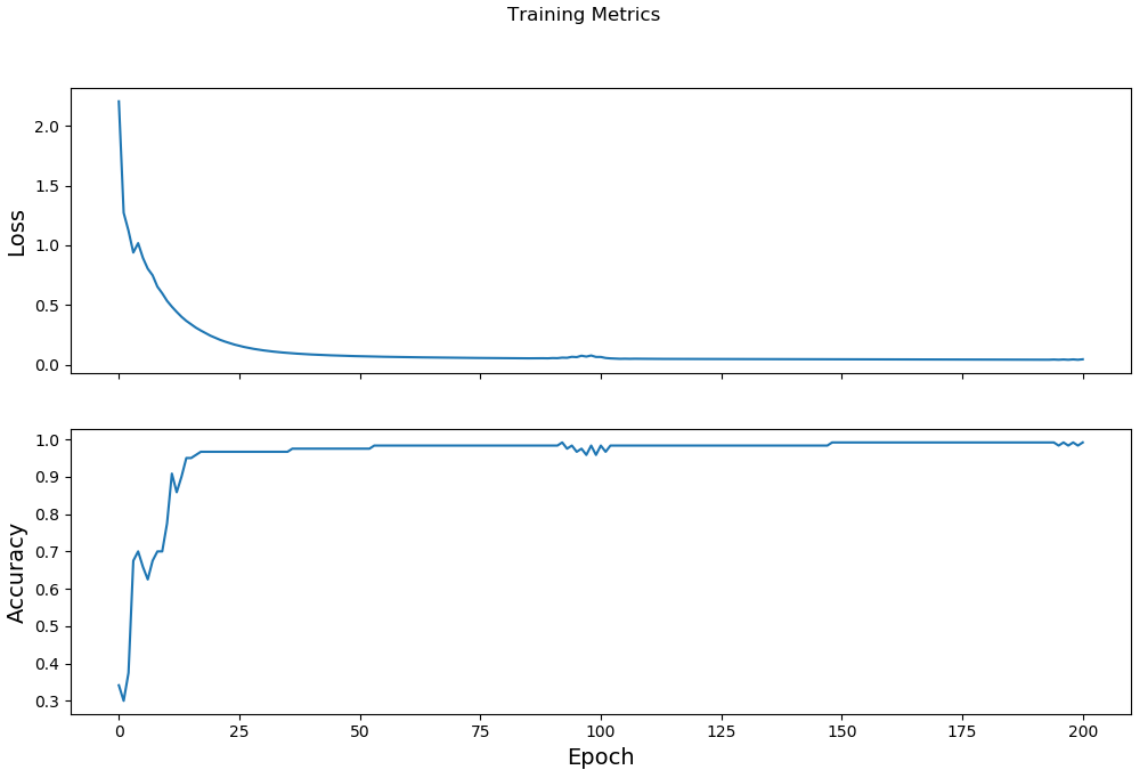

Рисунок 4.5 – Дані точності та втрат для 201 циклів навчання

```
Loss test: 3.9369044303894043
Step: 0, Initial Loss: 3.9369044303894043
Step: 1,         Loss: 3.204029083251953
Epoch 000: Loss: 2.205, Accuracy: 34.167%
Epoch 050: Loss: 0.071, Accuracy: 97.500%
Epoch 100: Loss: 0.065, Accuracy: 98.333%
Epoch 150: Loss: 0.045, Accuracy: 99.167%
Epoch 200: Loss: 0.045, Accuracy: 99.167%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (100.0%)
Example 2 prediction: Iris virginica (99.0%)
```

Рисунок 4.6 – Тестові приклади для 201 циклів навчання

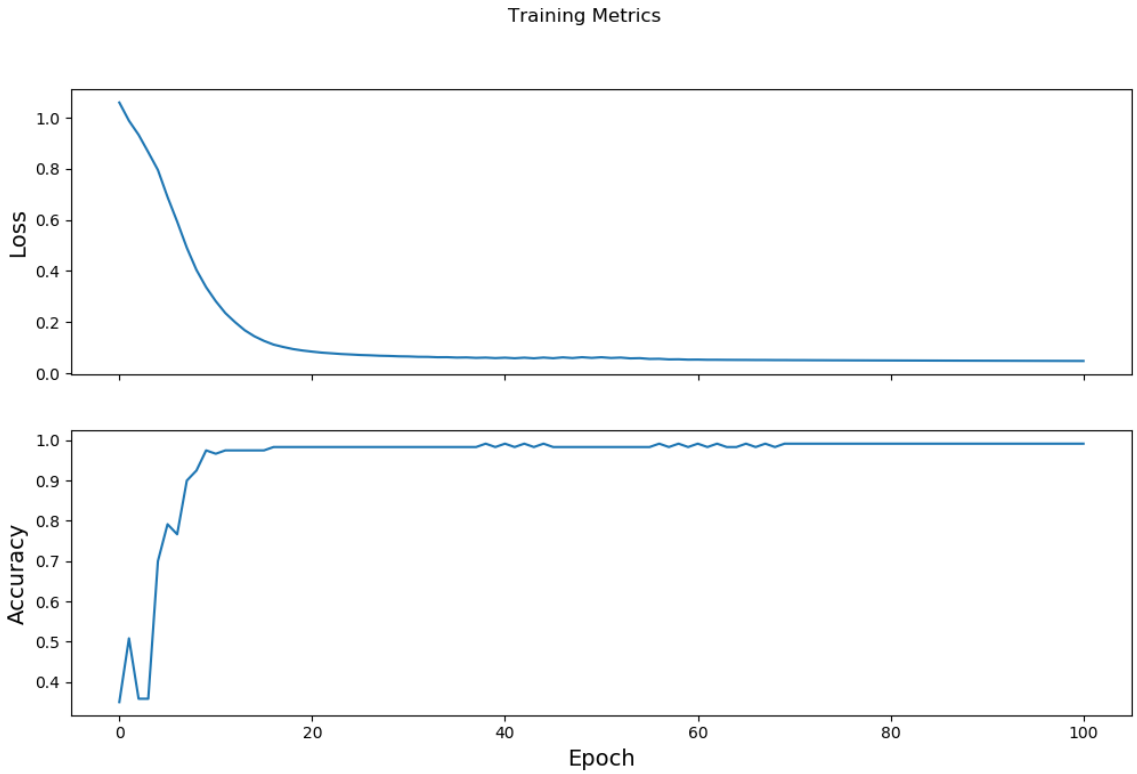

Рисунок 4.7 – Дані точності та втрат для 101 циклів навчання

```
Loss test: 1.1039094924926758
Step: 0, Initial Loss: 1.1039094924926758
Step: 1,         Loss: 1.0326567888259888
Epoch 000: Loss: 1.059, Accuracy: 35.000%
Epoch 050: Loss: 0.063, Accuracy: 98.333%
Epoch 100: Loss: 0.048, Accuracy: 99.167%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (99.9%)
Example 2 prediction: Iris virginica (99.0%)
```

Рисунок 4.7 – Тестові приклади для 101 циклів навчання

Якщо такі методи регуляризації, як зменшення ваги, які оновлюють функцію втрат для заохочення менш складних моделей, вважаються "явною" регуляризацією, то раннє припинення може розглядатися як тип "неявної" регуляризації, як, наприклад, використання меншої мережі, яка має меншу потужність. Регуляризація також може бути неявною, як це відбувається у випадку ранньої зупинки. [28]

4.2. Алгоритми оптимізації

4.2.1 Алгоритм ADAM

Вибір алгоритму оптимізації для моделі глибокого навчання може означати різний результат в хвилинах, годинах та днях.

Алгоритм оптимізації Адама - це розширення до стохастичного градієнтного спуску, яке останнім часом сприйняло більш широке застосування для програм глибокого навчання в комп'ютерному зорі та природній обробці мови.

Адам - це алгоритм оптимізації, який може використовувати замість класичної процедури стохастичного градієнта спуску для оновлення ітераційних мережевих ваг на основі даних навчальних екземплярів.

Адама представили Дідерік Кінгма з OpenAI та Джиммі Ба з Університету Торонто у своєму документі (плакаті) МКЛР за 2015 рік під назвою «Адам: метод стохастичної оптимізації». Алгоритм називається Адамом. Це не є абревіатурою і не пишеться як "ADAM". Вводячи алгоритм, автори перераховують привабливі переваги використання Адама для невіпучих задач оптимізації:[ ]

— Пряма реалізація.

— Обчислювально ефективний.

— Малі вимоги до пам'яті.

— Інваріантне до діагонального масштабування градієнтів.

— Добре підходить для великих проблем з даними та / або параметрами.

— Підходить для нестаціонарних цілей.

— Підходить для проблем з дуже галасливими / або розрідженими градієнтами.

— Гіперпараметри мають інтуїтивну інтерпретацію і зазвичай потребують невеликої настройки.

Адам відрізняється від класичного стохастичного градієнтного спуску.

Стохастичний градієнтний спуск підтримує єдину швидкість навчання (називається альфа) для всіх оновлень ваги, а рівень навчання не змінюється під час навчання.

Швидкість навчання підтримується для кожної ваги (параметра) мережі та окремо адаптується під час розробки навчання.

Метод обчислює індивідуальні пристосувальні темпи навчання для різних параметрів від оцінок першого та другого моментів градієнтів.

Автори описують Адама як поєднання переваг двох інших розширень стохастичного градієнтного спуску.

Алгоритм адаптивного градієнта (AdaGrad), який підтримує швидкість навчання за параметрами, що покращує продуктивність при проблемах із розрідженими градієнтами (наприклад, проблеми з природним мовою та комп'ютерним зором).

Кореневе середнє квадратне поширення (RMSProp), яке також підтримує показники навчання за параметрами, які адаптуються на основі середніх останніх величин градієнтів для ваги (наприклад, як швидко воно змінюється). Адам поєднує переваги як AdaGrad, так і RMSProp.

Замість адаптації темпів навчання параметрів на основі середнього першого моменту (середнього значення), як у RMSProp, Адам також використовує середнє значення для другого моменту градієнтів (без центричне відхилення).

Зокрема, алгоритм обчислює експоненціальну рухому середню градієнта та градієнта квадрата, а параметри beta1 та beta2 контролюють швидкості спаду цих рухомих середніх.

Початкове значення рухомої середніх значень та значень бета1 та бета2, близьких до 1,0 (рекомендовано), призводять до зміщення оцінок моменту у напрямку до нуля. Цей ухил долається спочатку обчисленням упереджених оцінок, а потім обчисленням, скоригованих з ухилом.

Адам - популярний алгоритм у галузі глибокого навчання, оскільки він швидко досягає хороших результатів.

Емпіричні результати демонструють, що Адам добре працює на практиці та вигідно порівнює інші методи стохастичної оптимізації.

У первинному документі Адаму було продемонстровано емпіричним шляхом, щоб показати, що конвергенція відповідає очікуванням теоретичного аналізу. Адама було застосовано до алгоритму логістичної регресії на розпізнаванні знаків MNIST та наборах даних аналізу настроїв IMDB, алгоритмі багатошарового перцептрону на наборі даних MNIST та конволюційних нейронних мережах на наборі даних розпізнавання зображень CIFAR-10. Вони роблять висновок:

Використовуючи великі моделі та набори даних, ми демонструємо, що Адам може ефективно вирішувати практичні проблеми глибокого навчання.

Себастьян Рудер розробив вичерпний огляд сучасних алгоритмів оптимізації спуску градієнта під назвою «Огляд алгоритмів оптимізації градієнта спуску», опублікований спочатку як допис у блозі, а потім технічний звіт у 2016 році.

Наскільки, RMSprop, Adadelta та Adam - це дуже схожі алгоритми, які добре справляються за подібних обставин. Його корекція зміщення допомагає Адаму трохи перевершити RMSprop до кінця оптимізації, оскільки градієнти стають все рідшими. Наскільки Адам може бути найкращим загальним вибором.

На практиці Адама в даний час рекомендується використовувати як алгоритм за замовчуванням для використання, і він часто працює трохи краще, ніж RMSProp. Однак часто варто спробувати SGD Nesterov Momentum як альтернативу. Два рекомендованих оновлення для використання - це SGD Nesterov Momentum або Adam.

Параметри конфігурації Адама

альфа. Також називається швидкістю навчання або розміром кроку. Пропорція ваги оновлюється (наприклад, 0,001). Більші значення (наприклад, 0,3) призводять до швидшого початкового навчання до оновлення курсу. Менші значення (наприклад, 1.0E-5) сповільнюють навчання безпосередньо під час тренувань

бета1. Експоненціальна швидкість занепаду для оцінок першого моменту (наприклад, 0,9).

бета2. Експоненціальна швидкість занепаду для оцінок другого моменту (наприклад, 0,999). Це значення має бути встановлено близьким до 1,0 при проблемах із розрідженим градієнтом (наприклад, проблеми з NLP та комп'ютерним зором).

епсилон. Це дуже невелика кількість, щоб запобігти будь-якому поділу на нуль у реалізації (наприклад, 10E-8).

Крім того, зниження швидкості навчання також може використовуватися разом з Адамом. У роботі використовується швидкість занепаду альфа = альфа / sqrt (t), оновлена кожна епоха (t), для демонстрації логістичної регресії.

У статті Адама пропонується:

Рекомендовані налаштування за замовчуванням для тестованих проблем машинного навчання - альфа = 0,001, бета1 = 0,9, бета2 = 0,999 і epsilon = 10^{-8}

Документація TensorFlow пропонує деяку настройку epsilon:

Значення за замовчуванням 1e-8 для epsilon в цілому може бути непоганим за замовчуванням. Наприклад, при навчанні мережі Inception в ImageNet поточним хорошим вибором є 1,0 або 0,1.

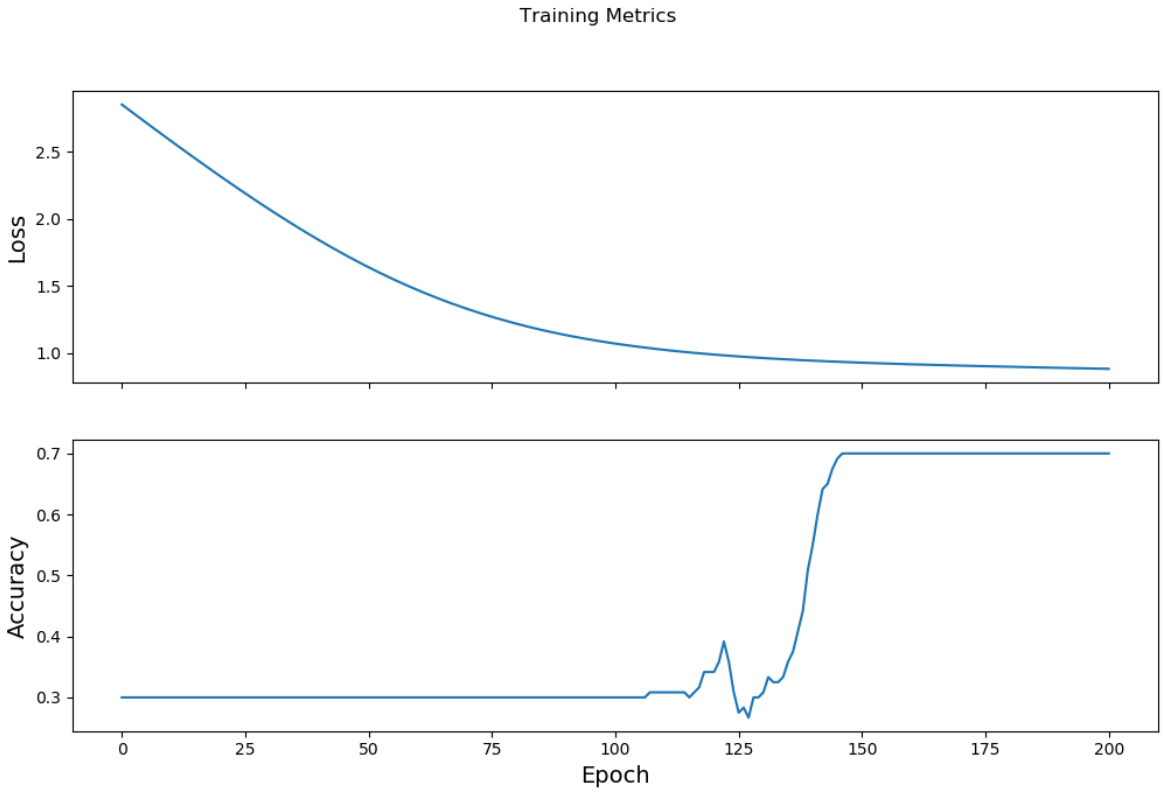

Рисунок 4.8 – Дані точності та втрат для коефіцієнта навчання 0.0001

```
Loss test: 2.8300857543945312
Step: 0, Initial Loss: 2.8300857543945312
Step: 1,         Loss: 2.823307991027832
Epoch 000: Loss: 2.850, Accuracy: 30.000%
Epoch 050: Loss: 1.640, Accuracy: 30.000%
Epoch 100: Loss: 1.072, Accuracy: 30.000%
Epoch 150: Loss: 0.930, Accuracy: 70.000%
Epoch 200: Loss: 0.884, Accuracy: 70.000%
Test set accuracy: 53.333%
Example 0 prediction: Iris setosa (45.4%)
Example 1 prediction: Iris virginica (38.2%)
Example 2 prediction: Iris virginica (48.6%)
```

Рисунок 4.8 – Тестові набори

Проаналізувавши параметри точності та помилки які показані на рисунку 4.8 і підстановки тестових наборів, можна стверджувати що мережа не відповідає мінімальним вимогам точності. Значення розпізнавання на прикладах таких не досягають потрібних 85%, основною причиною таких результатів є малий коефіцієнт навчання.

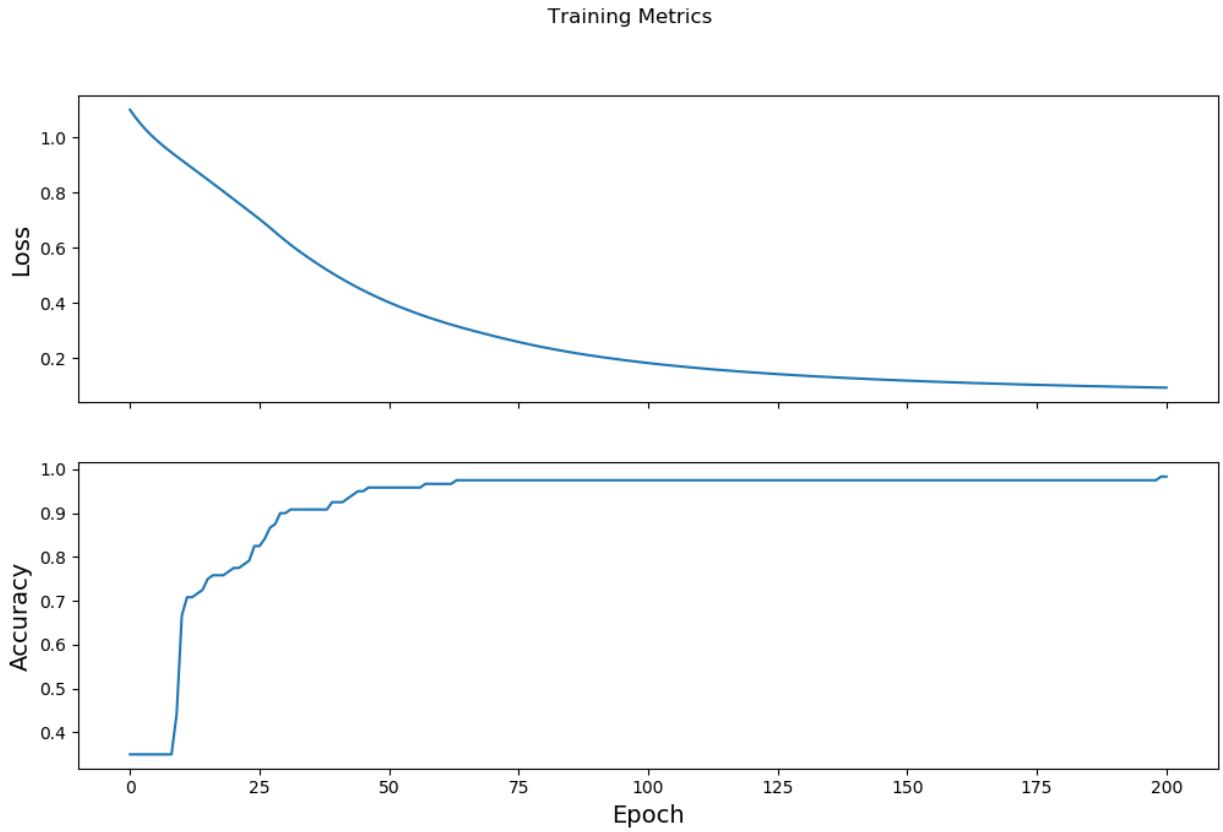

Рисунок 4.9 – Дані точності та втрат для коефіцієнта навчання 0.001

```
Loss test: 1.1940513849258423
Step: 0, Initial Loss: 1.1940513849258423
Step: 1,         Loss: 1.1808098554611206
Epoch 000: Loss: 1.099, Accuracy: 35.000%
Epoch 050: Loss: 0.402, Accuracy: 95.833%
Epoch 100: Loss: 0.182, Accuracy: 97.500%
Epoch 150: Loss: 0.118, Accuracy: 97.500%
Epoch 200: Loss: 0.093, Accuracy: 98.333%
Test set accuracy: 100.000%
Example 0 prediction: Iris setosa (98.4%)
Example 1 prediction: Iris versicolor (92.8%)
Example 2 prediction: Iris virginica (86.4%)
```

Рисунок 4.10 – Тестові набори

Змінна коефіцієнта навчання до значення 0.1 дозволяє на тестових прикладах досягнути точності розпізнавання за 85%.

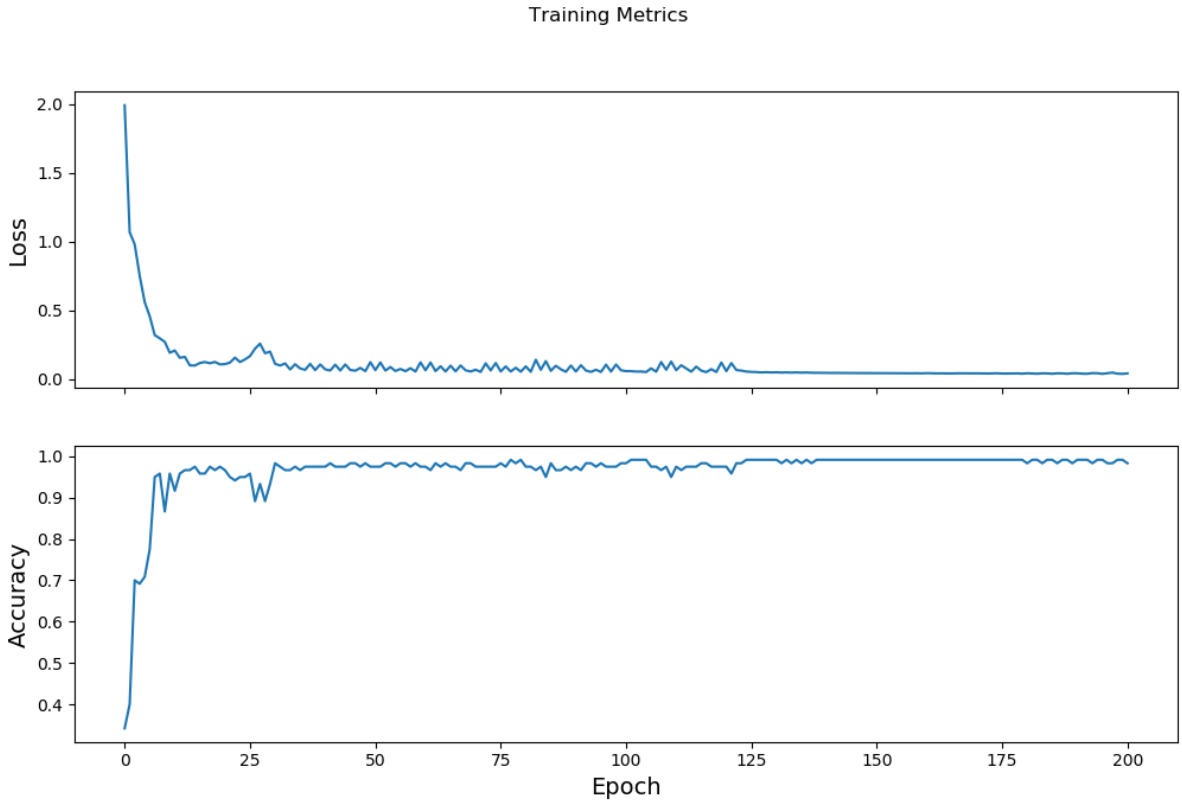

Рисунок 4.11 – Дані точності та втрат для коефіцієнта навчання 0.1

```
Loss test: 1.5901000499725342
Step: 0, Initial Loss: 1.5901000499725342
Step: 1,         Loss: 3.0444490909576416
Epoch 000: Loss: 1.992, Accuracy: 34.167%
Epoch 050: Loss: 0.067, Accuracy: 97.500%
Epoch 100: Loss: 0.059, Accuracy: 98.333%
Epoch 150: Loss: 0.045, Accuracy: 99.167%
Epoch 200: Loss: 0.043, Accuracy: 98.333%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (100.0%)
Example 2 prediction: Iris virginica (99.5%)
```

Рисунок 4.12 – Тестові набори

Змінна коефіцієнта навчання до значення 0.3 відповідає необхідному рівню точності розпізнавання, але нейрона мережа починає втрачати властивості узагальненняю Мережа починає запам’ятовувати набори на яких навчалась.

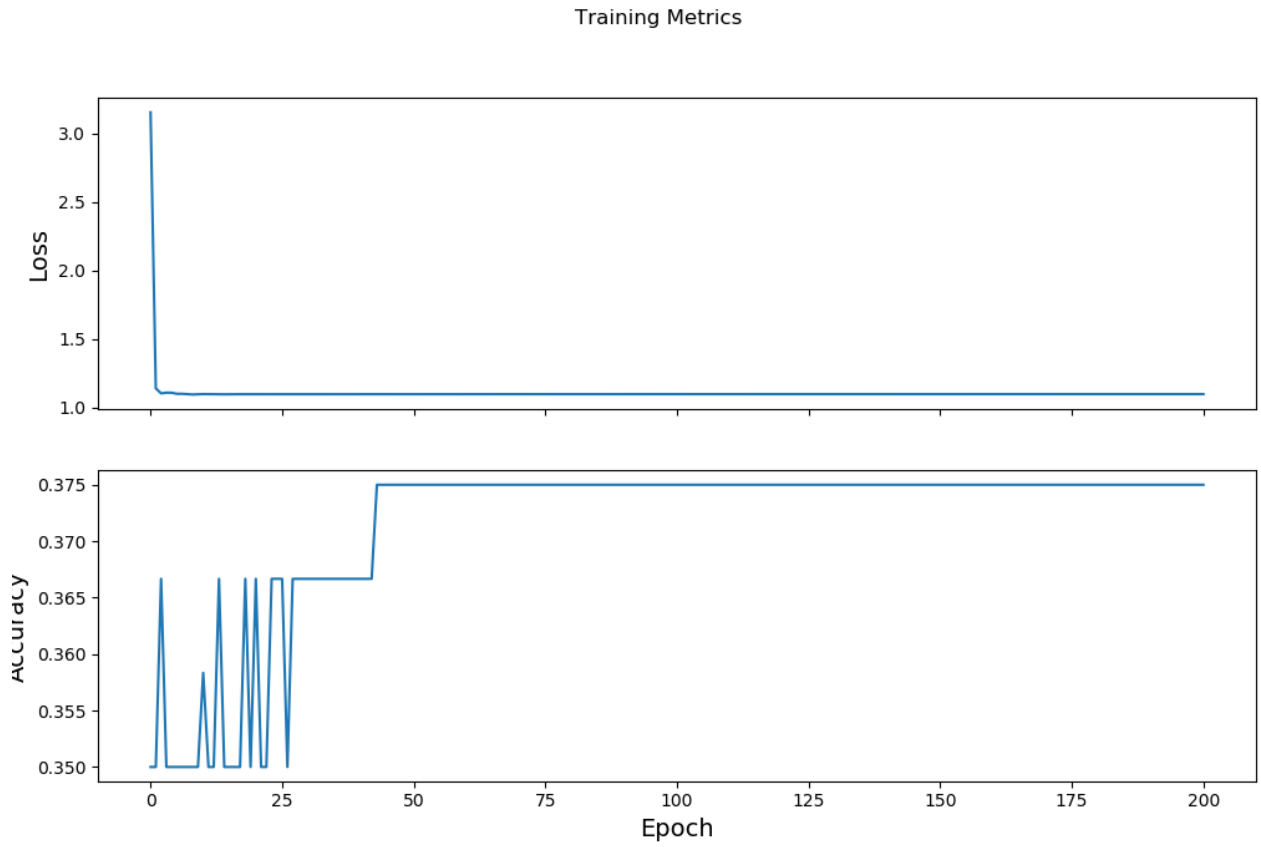

Рисунок 4.13 – Дані точності та втрат для коефіцієнта навчання 0.3

```
Epoch 000: Loss: 3.153, Accuracy: 35.000%
Epoch 050: Loss: 1.098, Accuracy: 37.500%
Epoch 100: Loss: 1.098, Accuracy: 37.500%
Epoch 150: Loss: 1.098, Accuracy: 37.500%
Epoch 200: Loss: 1.098, Accuracy: 37.500%
Test set accuracy: 26.667%
Example 0 prediction: Iris setosa (35.5%)
Example 1 prediction: Iris setosa (35.5%)
Example 2 prediction: Iris setosa (35.5%)
```

Рисунок 4.14 – Тестові набори

Дані точності та похибки для коефіцієнта навчання 0.3 як показано на графіках рисунка 4.13 не відповідають поставленим вимогам. На тестових прикладах рівень точності розпізнавання був нижчий за 40% відсотків, і після 50 епохи мережа перестала навчатися потрапивши в локальний екстремум.

Змінна коефіцієнта навчання до значення 0.9

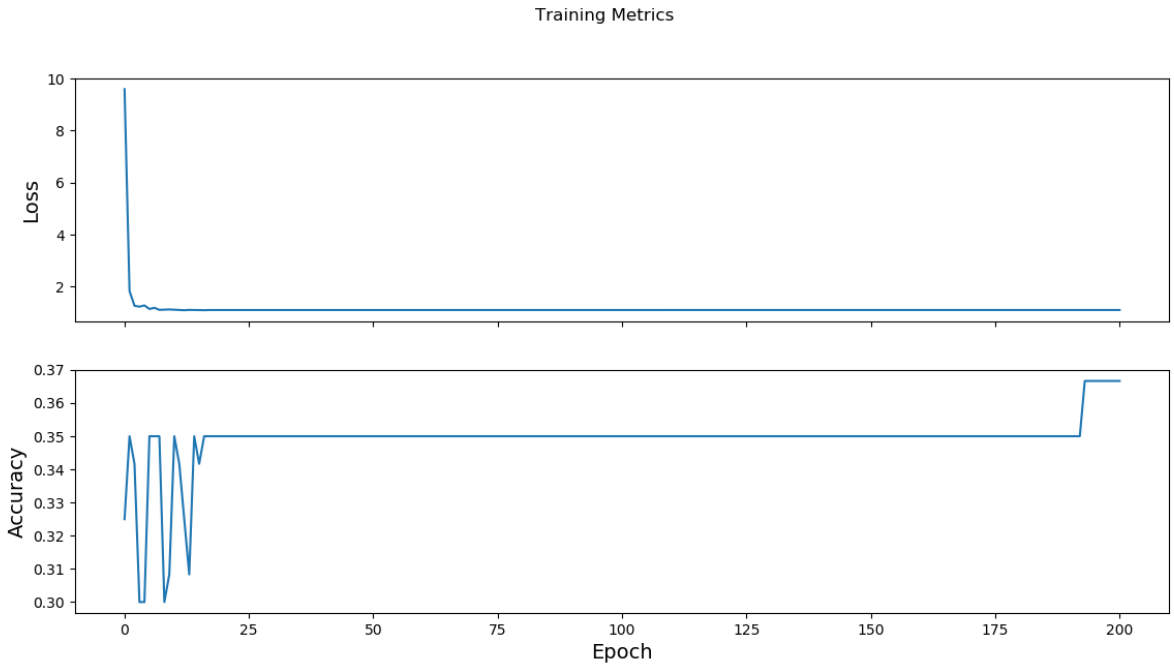

Рисунок 4.15 – Дані точності та втрат для коефіцієнта навчання 0.9

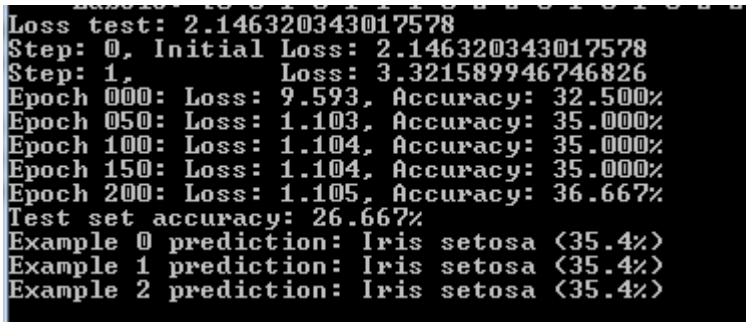

Рисунок 4.16 – Тестові набори

4.2.2 Алгоритм SGD

Gradient Descent - це популярна методика оптимізації в машинному та глибокому навчанні, і вона може використовуватися з більшістю, якщо не з усіх алгоритмів навчання. Градієнт - це в основному нахил функції; ступінь зміни параметра з величиною зміни іншого параметра. Математично він може

бути описаний як часткові похідні набору параметрів щодо його входів. Чим більше градієнт, тим крутіший схил. Спуск градієнта - це опукла функція.

Спуск градієнта може бути описаний як ітераційний метод, який використовується для пошуку значень параметрів функції, що максимально мінімізує функцію витрат. Параметри спочатку визначаються певним значенням, і з цього, Gradient Descent виконується ітеративно, щоб знайти оптимальні значення параметрів, використовуючи обчислення, щоб знайти мінімально можливе значення заданої функції витрат.
Зазвичай існує три типи градієнтного спуску:

— Пакетний градієнт спуску

— Стохастичний градієнт спуск

— Міні-серійний градієнтний спуск

Слово "стохастичний" означає систему або процес, пов'язаний з випадковою ймовірністю. Отже, у стохастичному градієнтному поході кілька вибірків вибираються випадковим чином, а не весь набір даних для кожної ітерації. У Gradient Descent існує термін під назвою "batch", який позначає загальну кількість вибірок із набору даних, який використовується для обчислення градієнта для кожної ітерації. У типовій оптимізації Gradient Descent, як Batch Gradient Descent, пакет вважається цілим набором даних. Хоча використання всього набору даних є дійсно корисним для отримання мінімумів менш галасливим або менш випадковим чином, але проблема виникає, коли наші набори даних отримують дійсно величезну кількість.

Припустимо, у наборі даних є мільйон зразків, тому якщо використовувати типову техніку оптимізації градієнтного спуску, доведеться використовувати всі мільйон зразків для виконання однієї ітерації під час виконання градієнтного спуску, і це потрібно зробити для кожну ітерацію до досягнення мінімумів. Отже, виконувати обчислювання дуже затратно по апаратним ресурсам.

Цю проблему вирішує Stohastic Gradient Descent. У SGD для виконання кожної ітерації використовується лише один зразок, тобто розмір партії одиниці. Зразок довільно перемішують і відбирають для проведення ітерації.

Отже, в SGD знаходиться градієнт функції вартості єдиного прикладу при кожній ітерації замість суми градієнта функції витрат у всіх прикладах.

У SGD, оскільки для кожної ітерації випадковим чином вибирається лише один зразок із набору даних, шлях алгоритму для досягнення мінімумів

зазвичай шумніший, ніж ваш типовий алгоритм Gradient Descent. Але це не має великого значення, тому що шлях алгоритму не має значення, поки не досягне мінімумів і зі значно меншим часом навчання.

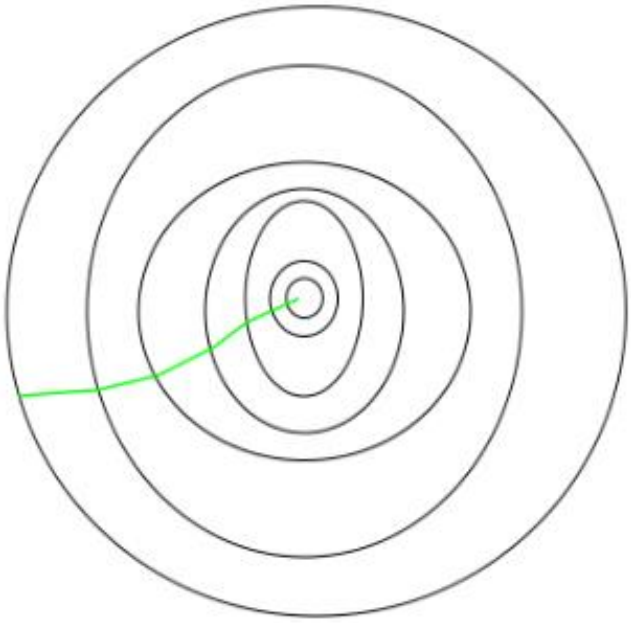

Рисунок 4.17 - Шлях, зроблений Пакетним градієнтом, -gd_path

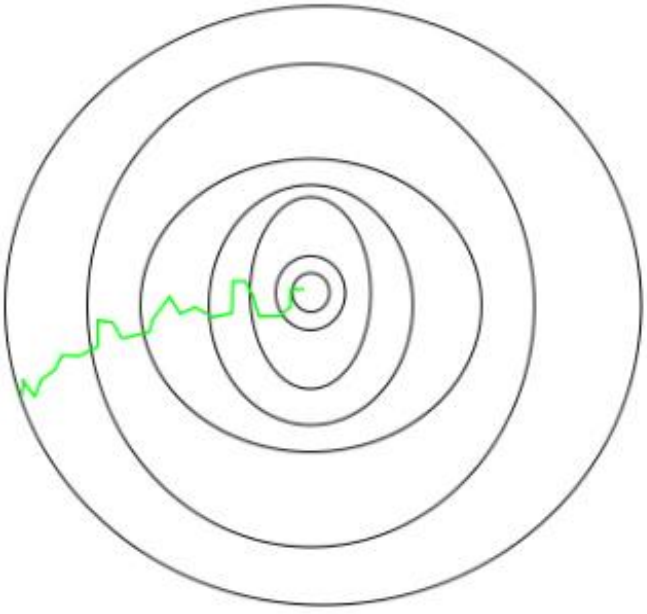

Рисунок 4.18 - Шлях по стохастичному градієнтному походженню - sgd_path

Варто зазначити одне, що SGD, як правило, задумливіше типового градієнтного спуску. Для досягнення мінімумів, як правило, знадобилася більша кількість ітерацій, через випадковість його спуску. Незважаючи на те, що для досягнення мінімумів потрібна більша кількість ітерацій, ніж типові градієнтні спуски, вона все одно обчислювальна набагато дешевше, ніж типовий спуск градієнта. Отже, у більшості сценаріїв для оптимізації алгоритму навчання SGD віддається перевагу Batch Gradient Descent.[25]

Змінна коефіцієнта навчання для алгоритму SGD до значення 0.01

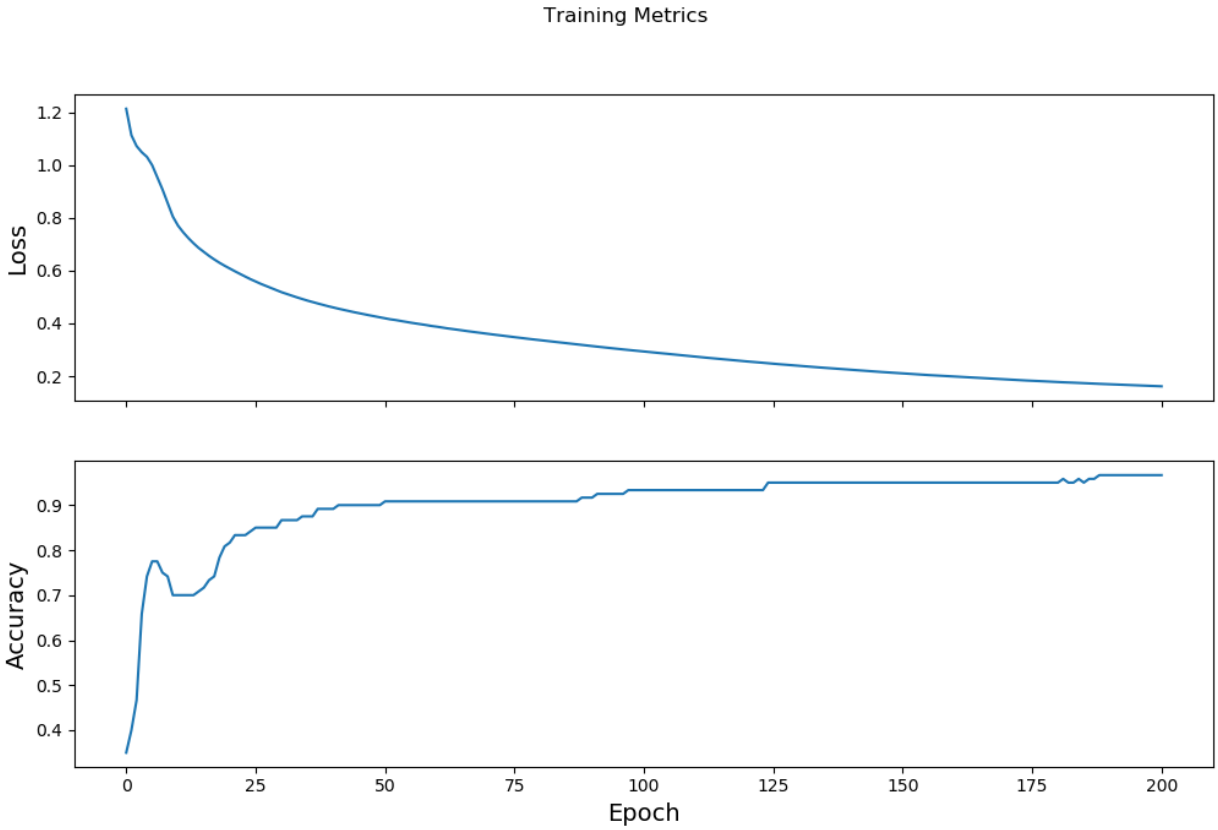

Рисунок 4.19 - Параметр навчання 0.01 для SGD

```
Loss test: 1.4588375091552734
Step: 0, Initial Loss: 1.4588375091552734
Step: 1,         Loss: 1.3312382698059082
Epoch 000: Loss: 1.214, Accuracy: 35.000%
Epoch 050: Loss: 0.420, Accuracy: 90.833%
Epoch 100: Loss: 0.293, Accuracy: 93.333%
Epoch 150: Loss: 0.210, Accuracy: 95.000%
Epoch 200: Loss: 0.161, Accuracy: 96.667%
Test set accuracy: 100.000%
Example 0 prediction: Iris setosa (97.8%)
Example 1 prediction: Iris versicolor (86.1%)
Example 2 prediction: Iris virginica (64.7%)
```

Рисунок 4.20 – Дані тесту для параметра навчання 0.01 для SGD

Змінна коефіцієнта навчання для алгоритму SGD до значення 0.0001

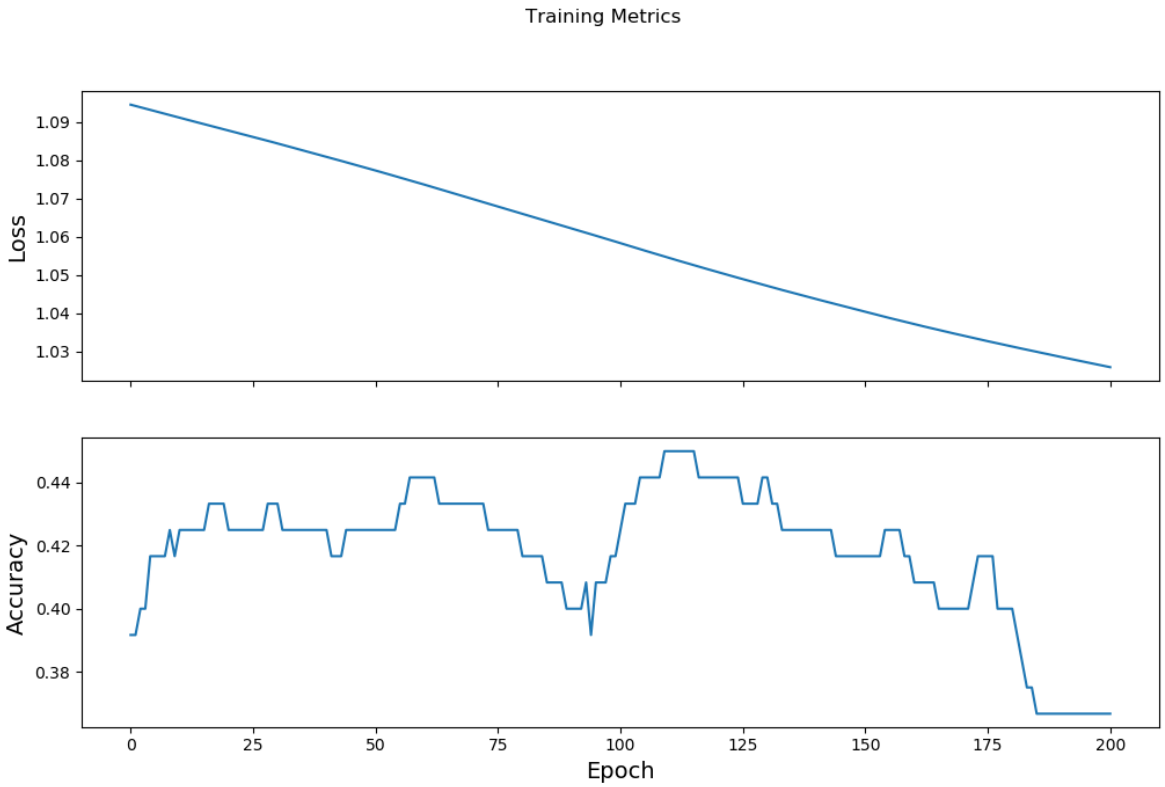

Рисунок 4.21- Параметр навчання 0.0001 для SGD

```
Loss test: 1.0874398946762085
Step: 0, Initial Loss: 1.0874398946762085
Step: 1,         Loss: 1.087362289428711
Epoch 000: Loss: 1.095, Accuracy: 39.167%
Epoch 050: Loss: 1.077, Accuracy: 42.500%
Epoch 100: Loss: 1.058, Accuracy: 42.500%
Epoch 150: Loss: 1.040, Accuracy: 41.667%
Epoch 200: Loss: 1.026, Accuracy: 36.667%
Test set accuracy: 46.667%
Example 0 prediction: Iris versicolor (34.2%)
Example 1 prediction: Iris versicolor (38.3%)
Example 2 prediction: Iris versicolor (42.1%)
```

Рисунок 4.22– Дані тесту для параметра навчання 0.0001 для SGD

На рисунку 4.21 графіки точності та похибки побудовані при параметрі 0.0001 для SGD низьке значення параметру не забезпечує коректного пошуку градієнта. Сама функція досягає локального есктремума и не може знайти глобальне значення.

Проведено зміна коефіцієнта навчання для алгоритму SGD до значення 0.1

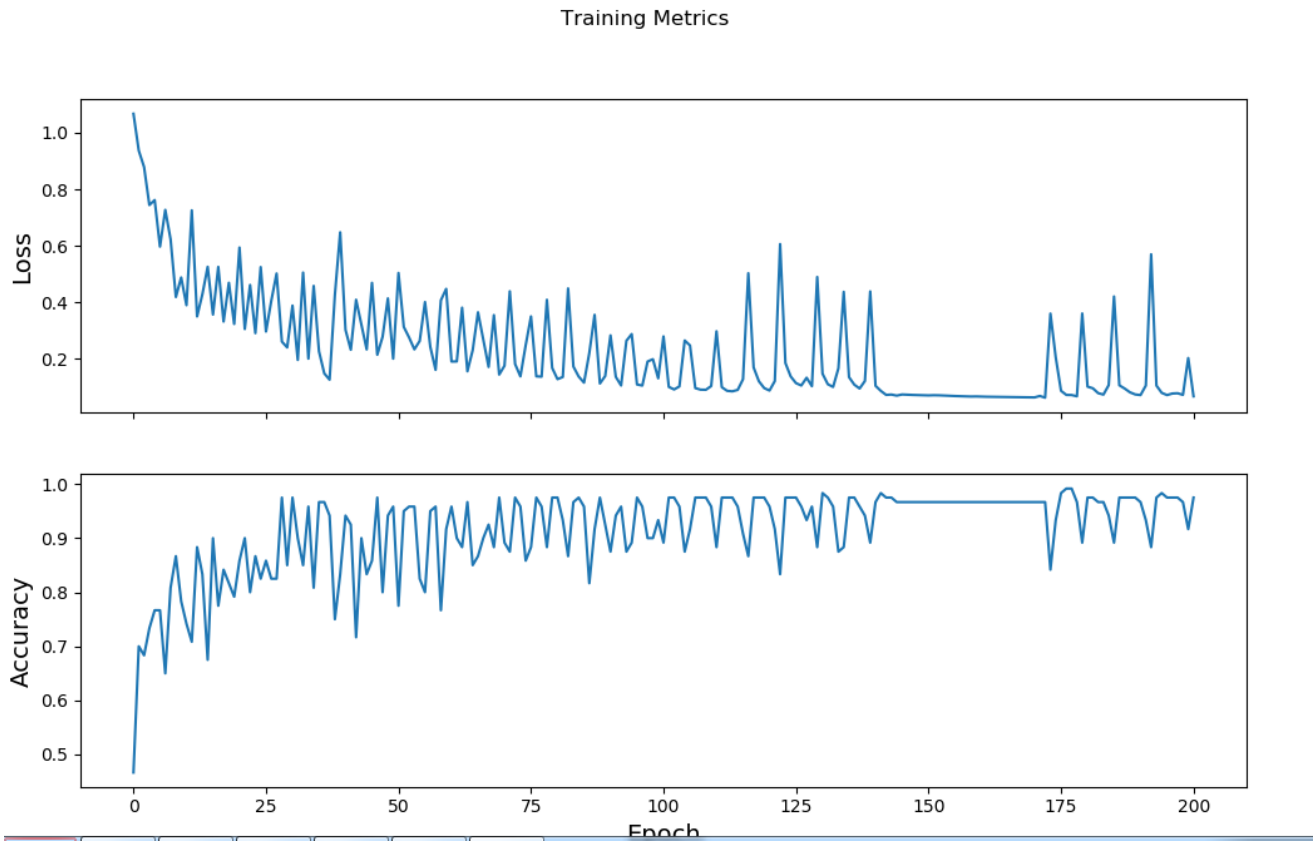

Рисунок 4.23 - Параметр навчання 0.1 для SGD

```
Loss test: 1.346954107284546
Step: 0, Initial Loss: 1.346954107284546
Step: 1,         Loss: 0.9732356071472168
Epoch 000: Loss: 1.068, Accuracy: 46.667%
Epoch 050: Loss: 0.504, Accuracy: 77.500%
Epoch 100: Loss: 0.279, Accuracy: 89.167%
Epoch 150: Loss: 0.070, Accuracy: 96.667%
Epoch 200: Loss: 0.067, Accuracy: 97.500%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (99.6%)
Example 2 prediction: Iris virginica (76.7%)
```

Рисунок 4.24 – Дані тесту для параметра навчання 0.1 для SGD

На рисунку 4.23 графіки точності та похибки побудовані при параметрі 0.1 для SGD значення параметру дає досить коректний пошуку градієнта.

Змінна коефіцієнта навчання для алгоритму SGD до значення 0.9

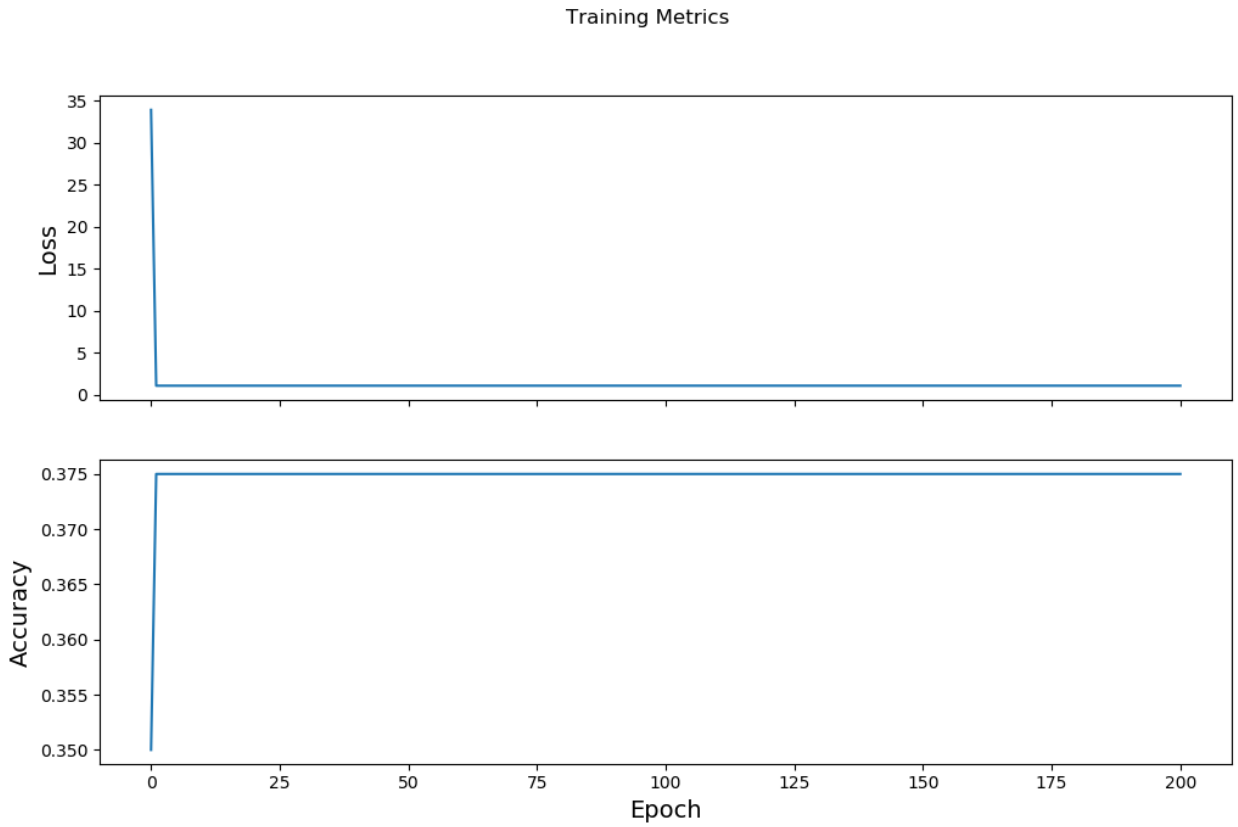

Рисунок 4.24 - Параметр навчання 0.9 для SGD

```
Loss test: 5.438350200653076
Step: 0, Initial Loss: 5.438350200653076
Step: 1,         Loss: 39.411643981933594
Epoch 000: Loss: 33.877, Accuracy: 35.000%
Epoch 050: Loss: 1.103, Accuracy: 37.500%
Epoch 100: Loss: 1.103, Accuracy: 37.500%
Epoch 150: Loss: 1.103, Accuracy: 37.500%
Epoch 200: Loss: 1.103, Accuracy: 37.500%
Test set accuracy: 26.667%
Example 0 prediction: Iris setosa (38.0%)
Example 1 prediction: Iris setosa (38.0%)
Example 2 prediction: Iris setosa (38.0%)
```

Рисунок 4.25 – Дані тесту для параметра навчання 0.9 для SGD

Згідно даних рисунка 4.24 на графіках значення точності не збільшується під час навчання, через високе значення коефіцієнта.

Отримані результати замірів навчання нейронної мережі занесенно до єдиної таблиці як на рисунку 4.26, для виявлення неформалізованих залежностей між вхідними і вихідними даними.

	num_epochs	coefficient_learn	ADAM	SGD	example0	example1	example2	accuracy_last_epoche
1	1000	,0010000	1	0	99,90	100,00	100,00	99,167
2	500	,0010000	1	0	100,00	100,00	99,90	99,167
3	201	,0010000	1	0	99,90	100,00	99,00	99,167
4	101	,0010000	1	0	99,90	99,90	99,00	99,167
5	201	,0001000	1	0	45,40	38,20	48,60	70,000
6	201	,0010000	1	0	98,40	92,80	86,40	98,333
7	201	,1000000	1	0	99,90	100,00	99,50	98,333
8	201	,3000000	1	0	35,50	35,50	35,50	37,500
9	201	,9000000	1	0	35,40	35,40	35,40	36,667
10	201	,0001000	0	1	34,20	38,30	42,10	36,667
11	201	,0100000	0	1	97,80	86,10	64,70	96,667
12	201	,1000000	0	1	99,90	99,60	76,70	97,500
13	201	,9000000	0	1	38,00	38,00	38,00	37,500

Рисунок 4.26– Статистичні дані різних параметрам

Оцінка впливу кожного параметру, який змінювався перед навчанням системи можна відобразити на таблиці кореляцій, що створена засобами пакетного SPSS.

Кореляційна залежність - взаємозалежність двох або декількох випадкових величин. Суть її пропонується у тому, що при вимірюванні значень однієї змінної відбувається закономірне вимірювання (зменшення або збільшення) інших зміних.

При розслідуванні кореляцій запитують визначити, що існують у статистичних достовірних місцях між двома або кількома переказами в одній або декількох вибірках.

Важно показувати, що коригувальна залежність відбиває лише взаємозв'язок між змінами і не говорить про причинно-наступні зв'язки. Кореляційний зв'язок може говорити про взаємозв'язок даних параметрів, присутніх у даній конкретній виборці, в іншій виборці ми можемо не спостерігати отримані корекції.

Показник кореляції. Коефіцієнт кореляції (r) робить велику отражающую ступінь взаємозв'язку двох перемінних між собою. Він може варіювати в межах від -1 (негативна кореляція) до +1 (позитивна кореляція). Якщо коефіцієнт кореляції рівний 0 то, це говорить про відсутність кореляційних зав’язків між змінними. Якщо коефіцієнт кореляції ближче к 1 (або -1), то говорить про сильну кореляції, а якщо ближче до 0, то про слабку.

Приблизна корекція збільшеного (або зменшення) значної однієї змінної веде до закономірного збільшення (або зменшення) іншої змінної, тобто взаємозв'язок типу збільшення-збільшення (зменшення-зменшення).

При негативній корекції збільшення або (зменшення) значень однієї перемінної веде до закономірного вживання (або збільшення) іншої переметної, тобто взаємозв'язки типу збільшення-зменшення (зменшення-збільшення).[24]

Корреляции

		num_epochs	example0	ADAM	coefficient_learn	example1	example2	accuracy_last_epoche	SGD
num_epochs	Корреляция Пирсона	1	,273	,227	-,189	,298	,368	,266	-,227
	Знач. (двухсторонняя)		,366	,456	,536	,323	,216	,379	,456
	N	13	13	13	13	13	13	13	13
example0	Корреляция Пирсона	,273	1	,182	-,618*	,991**	,932**	,976**	-,182
	Знач. (двухсторонняя)	,366		,552	,024	,000	,000	,000	,552
	N	13	13	13	13	13	13	13	13
ADAM	Корреляция Пирсона	,227	,182	1	-,156	,195	,393	,246	-1,000**
	Знач. (двухсторонняя)	,456	,552		,611	,523	,185	,418	,000
	N	13	13	13	13	13	13	13	13
coefficient_learn	Корреляция Пирсона	-,189	-,618*	-,156	1	-,602*	-,625*	-,694**	,156
	Знач. (двухсторонняя)	,536	,024	,611		,030	,022	,009	,611
	N	13	13	13	13	13	13	13	13
example1	Корреляция Пирсона	,298	,991**	,195	-,602*	1	,955**	,953**	-,195
	Знач. (двухсторонняя)	,323	,000	,523	,030		,000	,000	,523
	N	13	13	13	13	13	13	13	13
example2	Корреляция Пирсона	,368	,932**	,393	-,625*	,955**	1	,919**	-,393
	Знач. (двухсторонняя)	,216	,000	,185	,022	,000		,000	,185
	N	13	13	13	13	13	13	13	13
accuracy_last_epoche	Корреляция Пирсона	,266	,976**	,246	-,694**	,953**	,919**	1	-,246
	Знач. (двухсторонняя)	,379	,000	,418	,009	,000	,000		,418
	N	13	13	13	13	13	13	13	13
SGD	Корреляция Пирсона	-,227	-,182	-1,000**	,156	-,195	-,393	-,246	1
	Знач. (двухсторонняя)	,456	,552	,000	,611	,523	,185	,418	
	N	13	13	13	13	13	13	13	13

*. Корреляция значима на уровне 0,05 (двухсторонняя).

**. Корреляция значима на уровне 0,01 (двухсторонняя).

У математичній статистичній лінійній регресії представлений єдиний метод апроксимації залежностей між вхідними та вихідними перекладами на основі лілейної моделі. Є частиною більш широкої статистичної методики, називається регресійним аналізом.

У регресійному аналізі вхідні (незаповідні) перемінні називають також попередніми перемінними або регрессорамі, а залежні перемінні - критериальними.

Якщо розглядається залежність між однією вхідною і однією вихідною переменними, то існує можливість встановити лінійну регресію.

Таблиця 4.1 - Значення ANOVA для вибірки даних

ANOVA[a]

Модель		Сумма квадратов	ст.св.	Средний квадрат	F	Значимость
1	Регрессия	9955,309	6	1659,218	66,979	,000[b]
	Остаток	148,633	6	24,772		
	Всего	10103,942	12			
2	Регрессия	9955,306	5	1991,061	93,769	,000[c]
	Остаток	148,636	7	21,234		
	Всего	10103,942	12			
3	Регрессия	9954,797	4	2488,699	133,491	,000[d]
	Остаток	149,145	8	18,643		
	Всего	10103,942	12			
4	Регрессия	9907,490	3	3302,497	151,297	,000[e]
	Остаток	196,451	9	21,828		
	Всего	10103,942	12			

a. Зависимая переменная: accuracy_last_epoche
b. Предикторы: (константа), ADAM, coefficient_learn, num_epochs, example1, example2, example0
c. Предикторы: (константа), coefficient_learn, num_epochs, example1, example2, example0
d. Предикторы: (константа), coefficient_learn, example1, example2, example0
e. Предикторы: (константа), example1, example2, example0

Таблиця 4.2 – Коєфіцієнти моделей для вибірки даних

Коэффициенты[a]

Модель		Нестандартизованные коэффициенты		Стандартизованные коэффициенты	т	Значимость
		B	Стандартная ошибка	Бета		
1	(Константа)	11,972	5,950		2,012	,091
	num_epochs	-,001	,007	-,008	-,143	,891

	coefficient_learn	-8,063	6,042	-,092	-1,335	,230
	example0	1,850	,422	2,002	4,384	,005
	example1	-1,370	,583	-1,453	-2,352	,057
	example2	,402	,304	,386	1,322	,234
	ADAM	,047	4,643	,001	,010	,992
2	(Константа)	11,967	5,489		2,180	,066
	num_epochs	-,001	,006	-,008	-,155	,881
	coefficient_learn	-8,048	5,412	-,092	-1,487	,181
	example0	1,852	,368	2,003	5,037	,002
	example1	-1,374	,454	-1,456	-3,028	,019
	example2	,404	,189	,388	2,144	,069
3	(Константа)	11,927	5,138		2,321	,049
	coefficient_learn	-8,074	5,068	-,092	-1,593	,150
	example0	1,855	,344	2,007	5,395	,001
	example1	-1,373	,425	-1,456	-3,231	,012
	example2	,397	,171	,381	2,318	,049
4	(Константа)	5,884	3,749		1,569	,151
	example0	2,025	,354	2,191	5,726	,000
	example1	-1,568	,441	-1,662	-3,558	,006
	example2	,483	,176	,464	2,750	,022

a. Зависимая переменная: accuracy_last_epoche

Коефіцієнти b вказують, скільки одиниць продуктивності збільшується для одного одиничного збільшення кожного предиктора.[5]

Рівняння регресії: У = 11,972 – 0,001* Х 1 - 8,063 * Х 2 + 1,850 *Х 3 - 1,370 *Х 4 0,402*Х 5 +0,047*Х 6

де значення Х відповідають фактичним:

Х 1 – кількості єпох

Х 2 – коєфіцієнту навчання

Х 3 – Приклад 0

Х 4 – Приклад 1

Х 5 – Приклад 2

Х 6 – Наявність ADAM

R позначає кореляцію між прогнозованою та спостережуваною роботою. У нашому випадку R = 0.993. Оскільки це висока кореляція, модель прогнозує роботу досить точно.

Таблиця 4.3 – Визначення коєфіцієнта детермінації

Сводка для модели

Модель	R	R-квадрат	Скорректированный R-квадрат	Стандартная ошибка оценки
1	,993[a]	,985	,971	4,977165
2	,993[b]	,985	,975	4,607998
3	,993[c]	,985	,978	4,317769
4	,990[d]	,981	,974	4,672038

a. Предикторы: (константа), ADAM, coefficient_learn, num_epochs, example1, example2, example0
b. Предикторы: (константа), coefficient_learn, num_epochs, example1, example2, example0
c. Предикторы: (константа), coefficient_learn, example1, example2, example0
d. Предикторы: (константа), example1, example2, example0

Дисперсійний аналіз являє собою статистичний метод аналізу результатів, які залежать від якісних ознак.

4.3 Змінна функції активації шарів

4.3.1 Аналіз функції для прихованого шару

Модуль tf.nn. відповідає за примітивні обчислення нейронних мереж.

Зазвичай шар в нейронної мережі має деякий вектор введення, і примножує його на вагову матрицю, в результаті чого знову в векторі.

Кожне значення в результаті (як правило, float) вважається результатом. Тим не менше, більшість верств в нейронних мережах в даний час пов'язані з не лінійностями, отже, надбудова, яка, можна сказати, додає складності до цих вихідних значень. Надовго це були сигмоїди і тангенсоїди.

Функцію, яка призводить до 0, якщо вхід негативний, і сам введення, якщо цей вхід дорівнює 0 або позитивний. Ця спеціальна функція надбудови (або краще "функція активації") називається relu.

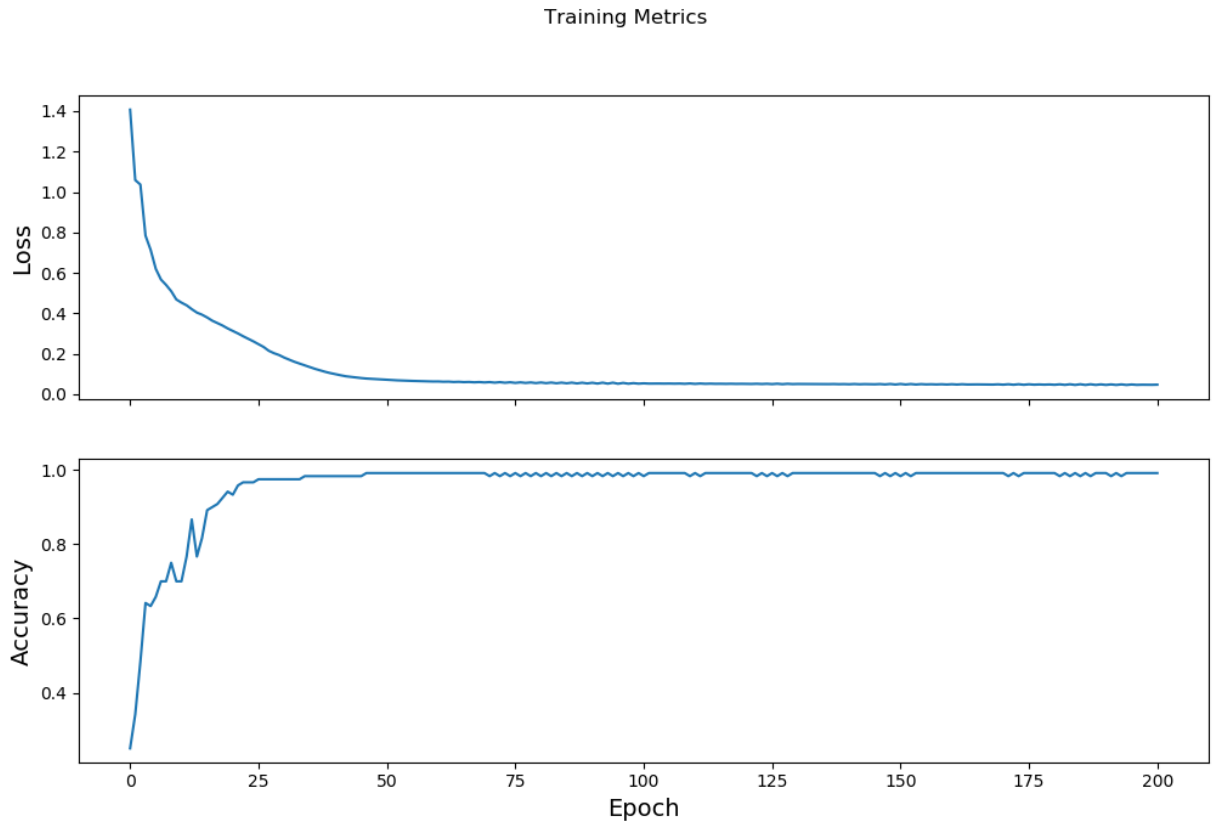

Рисунок 4.28– Тип функції Selu для прихованого шару

```
Step: 0, Initial Loss: 2.668111562728882
Step: 1,         Loss: 1.6912332773208618
Epoch 000: Loss: 1.407, Accuracy: 25.000%
Epoch 050: Loss: 0.071, Accuracy: 99.167%
Epoch 100: Loss: 0.054, Accuracy: 98.333%
Epoch 150: Loss: 0.051, Accuracy: 98.333%
Epoch 200: Loss: 0.047, Accuracy: 99.167%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (100.0%)
Example 1 prediction: Iris versicolor (99.9%)
Example 2 prediction: Iris virginica (99.1%)
```

Рисунок 4.29– Тестові набори для функції Selu

Потенційна проблема із зворотним розповсюдженням полягає в тому, що градієнти можуть бути занадто маленькими. Ця проблема називається зникаючими градієнтами. Якщо мережа страждає від зниклих градієнтів, ваги не будуть коригуватися, а навчання припиняється. На високому рівні глибокі мережі примножують багато градієнтів під час розмноження. Якщо градієнти близькі до нуля, весь продукт падає. Це, у свою чергу, штовхає інші градієнти ближче до нуля тощо. [29]

SELU має якість саморегуляції, і більше не потрібно боятися зниклих градієнтів. Існує три причини, використовувати SELU замість ReLU:

— Подібно до ReLU, SELU включають глибокі нейронні мережі, оскільки немає проблем з зникаючими градієнтами.

— На відміну від ReLU, SELU не можуть загинути.

— SELU самостійно навчаються швидше та краще, ніж інші функції активації, навіть якщо вони поєднуються з пакетною нормалізацією.

У деяких випадках реальної різниці між ReLU та SELU немає. Однак, якщо така є, SELU значно перевищують ReLU.

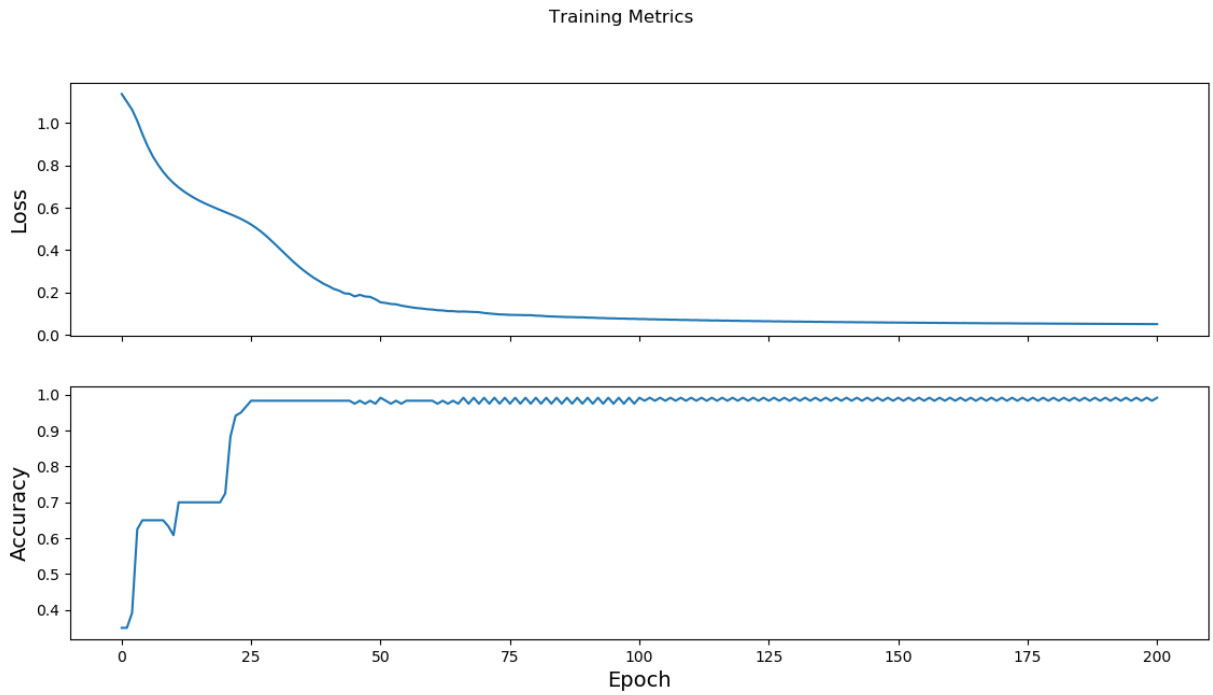

Рисунок 4.30 – Тип функції Softmax для прихованого шару

```
Loss test: 1.133005976676941
Step: 0, Initial Loss: 1.133005976676941
Step: 1,         Loss: 1.11509370803833
Epoch 000: Loss: 1.137, Accuracy: 35.000%
Epoch 050: Loss: 0.154, Accuracy: 99.167%
Epoch 100: Loss: 0.075, Accuracy: 99.167%
Epoch 150: Loss: 0.058, Accuracy: 99.167%
Epoch 200: Loss: 0.052, Accuracy: 99.167%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.2%)
Example 1 prediction: Iris versicolor (99.4%)
Example 2 prediction: Iris virginica (98.0%)
```

Рисунок 4.31 – Тестові набори для функції Softmax

Softmax - це форма логістичної регресії, яка нормалізує вхідне значення у вектор значень, який слід за розподілом ймовірностей, загальний підсумок

якого до 1. Вихідні значення знаходяться між діапазоном [0,1], тому здатні уникнути бінарної класифікації та вмістити якомога більше класів чи розмірів у нашій моделі нейронної мережі. Ось чому softmax іноді називають багаточленною логістичною регресією.

Окрім іншого, ще одна назва Softmax Regression - це максимальний ентропійний (MaxEnt) класифікатор.

Функція зазвичай використовується для обчислення втрат, які можна очікувати при навчанні набору даних. Відомі випадки регресії softmax є в дискримінаційних моделях, таких як перехресна ентропія та оцінка проти помірного шуму. Це лише дві серед різних методики, які намагаються оптимізувати поточний навчальний набір, щоб збільшити ймовірність передбачення правильного слова чи речення.

Визначення може звучати як тривіальне, але в NLP ця функція регресії була корисною в якості порівняльної бази. Слід зазначити, що softmax ідеально не використовувати як функція активації, як Sigmoid або ReLU , а кращим рішенням є реалізація прихованими між шарами, які можуть бути декількома або лише одним. [27]

4.3.2 Аналіз функції для вхідного шару

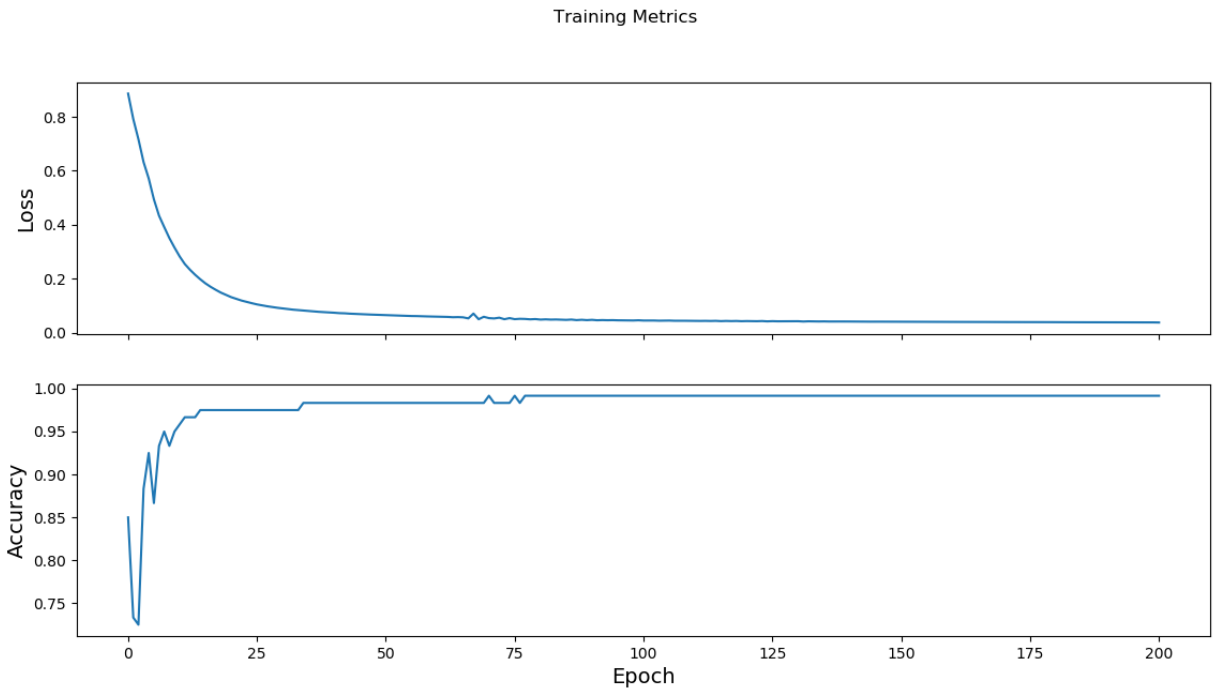

Рисунок 4.32 – Тип функції Selu для вхідного шару

```
Loss test: 1.1424906253814697
Step: 0, Initial Loss: 1.1424906253814697
Step: 1,         Loss: 0.9071813821792603
Epoch 000: Loss: 0.886, Accuracy: 85.000%
Epoch 050: Loss: 0.066, Accuracy: 98.333%
Epoch 100: Loss: 0.046, Accuracy: 99.167%
Epoch 150: Loss: 0.042, Accuracy: 99.167%
Epoch 200: Loss: 0.039, Accuracy: 99.167%
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (100.0%)
Example 2 prediction: Iris virginica (100.0%)
```

Рисунок 4.33 – Тестові набори для функції Selu

Змінна функції активації для вхідного шару

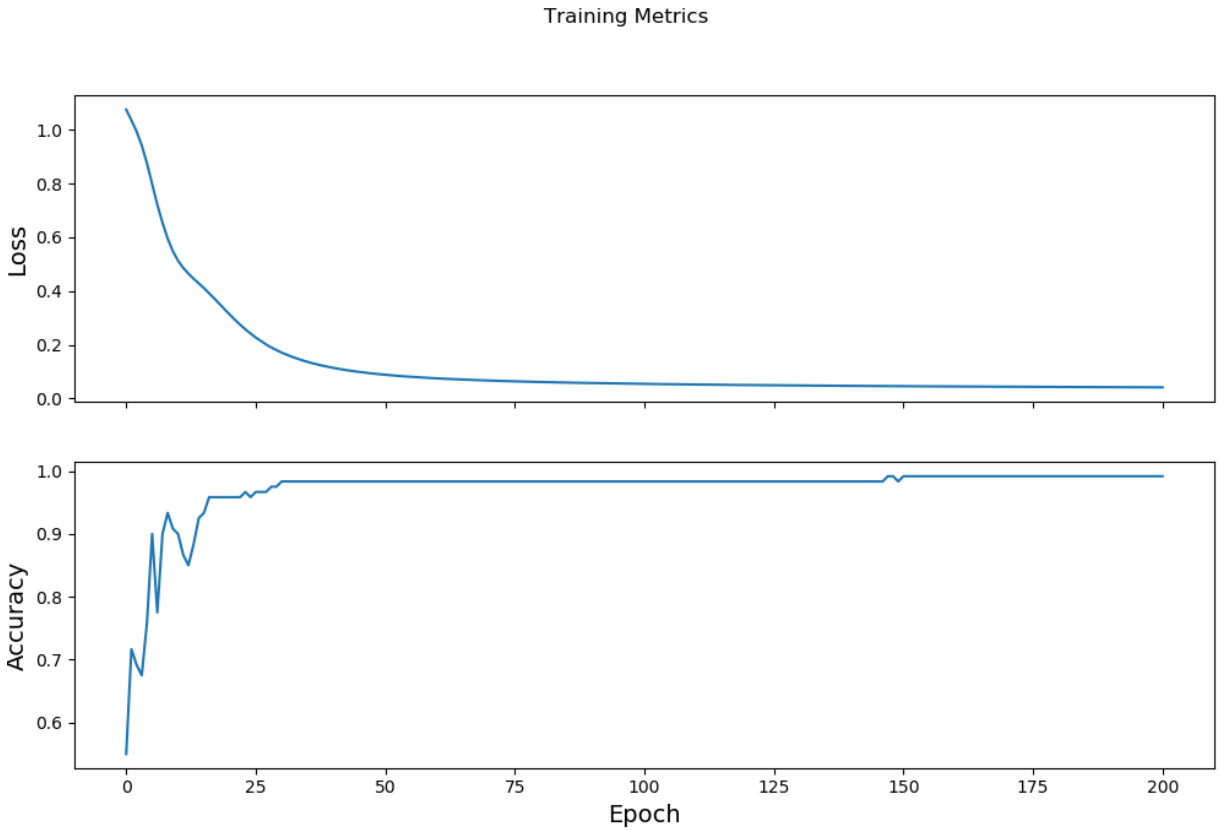

Рисунок 4.34 – Тип функції Softmax для вхідного шару

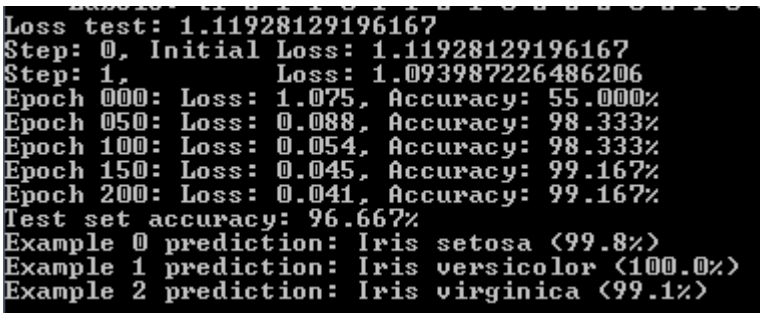

Рисунок 4.35 – Тестові набори для функції Softmax

	input_SELU	hide_SELU	input_RULE	hide_RULE	input_SOFT	hide_SOFT	example0	example1	example2	accuracy
1	0	1	1	0	,00	,00	100,00	99,90	99,90	99,167
2	1	0	0	0	,00	1,00	99,20	99,40	98,00	99,167
3	1	0	0	1	,00	,00	99,90	100,00	100,00	99,167
4	0	0	0	1	1,00	,00	99,80	100,00	99,10	99,167

Рисунок 4.36 – Дані тесту

4.3.3 Оцінка швидкості навчання нейронної мережі в залежності від параметра Batch

Розмір партії - це гіперпараметр, який визначає кількість проб, до яких потрібно опрацювати перед оновленням внутрішніх параметрів моделі.

Уявлення про партію як про повторне циклічне повторення одного чи декількох зразків та прогнозування. В кінці партії прогнози порівнюються з очікуваними вихідними змінними та обчислюється помилка. З цієї помилки алгоритм оновлення використовується для вдосконалення моделі, наприклад. рухатися вниз по градієнту помилки.

Навчальний набір даних може бути розділений на одну або кілька партій.

Коли всі навчальні зразки використовуються для створення однієї партії, алгоритм навчання називається пакетним градієнтом спуску. Коли партія має розмір одного зразка, алгоритм навчання називається стохастичним градієнтом. Коли розмір партії більше одного зразка і менше розміру навчального набору даних, алгоритм навчання називається міні-пакетним градієнтом спуску. [25]

У випадку мініатюрного градієнта спуску популярні розміри партії включають 32, 64 та 128 проб.

Програмна реалізація функції яка дозволяє зробити заміри періоду навчання нейронної мережі.

```
#timer
start_time = time.time()
```

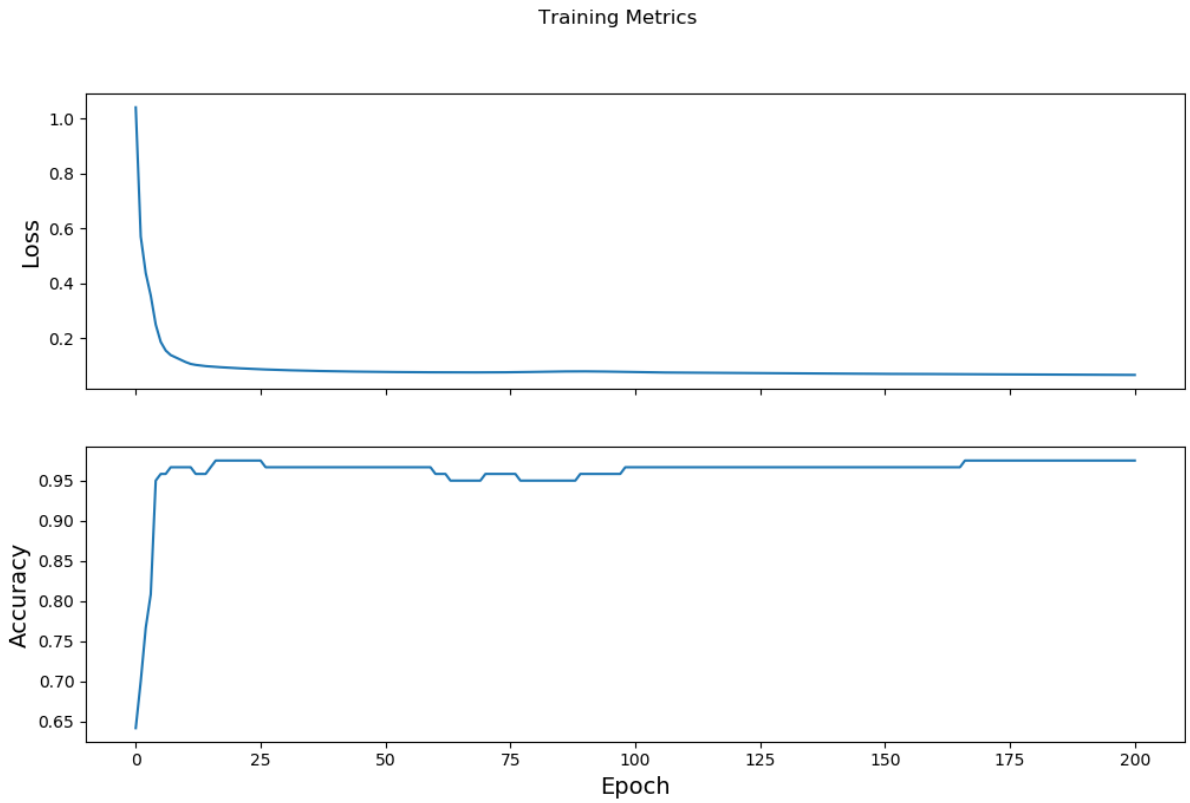

Рисунок 4.37– Розмір параметра Batch = 4

```
Loss test: 1.187119960784912
Step: 0, Initial Loss: 1.187119960784912
Step: 1,         Loss: 0.9702122211456299
Epoch 000: Loss: 1.023, Accuracy: 59.167%
Epoch 050: Loss: 0.080, Accuracy: 97.500%
Epoch 100: Loss: 0.071, Accuracy: 97.500%
Epoch 150: Loss: 0.069, Accuracy: 97.500%
Epoch 200: Loss: 0.067, Accuracy: 97.500%
--- 83.5777804851532 seconds ---
Test set accuracy: 93.333%
Example 0 prediction: Iris setosa (100.0%)
Example 1 prediction: Iris versicolor (99.6%)
Example 2 prediction: Iris virginica (99.7%)
```

Рисунок 4.38 – Результати тесту при batch = 4

Якщо задати значення рівне 8, то одна партія розглядатиметься із восьми точок.

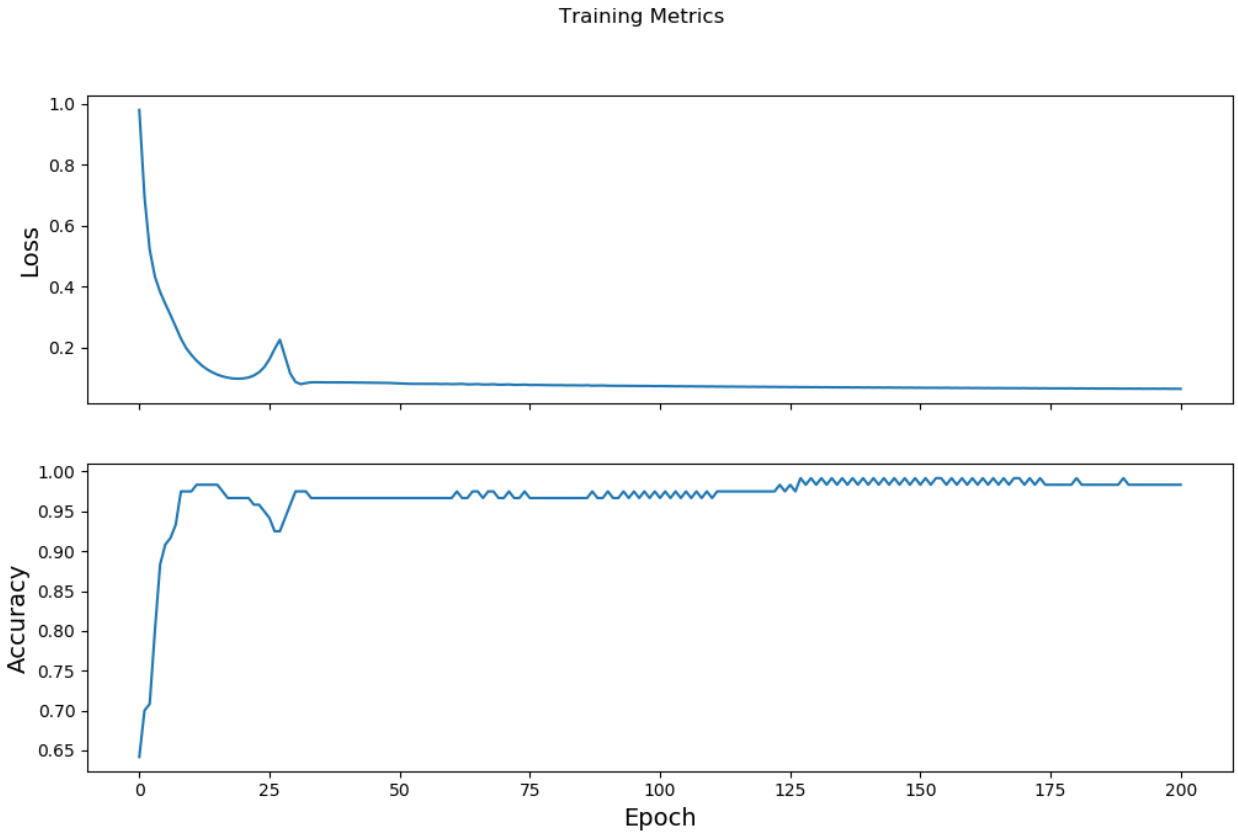

Рисунок 4.39 – Розмір параметра Batch = 8

```
Labels: [1 0 2 1 0 2 0 1]
Loss test: 1.3687093257904053
Step: 0, Initial Loss: 1.3687093257904053
Step: 1,         Loss: 1.2274218797683716
Epoch 000: Loss: 1.080, Accuracy: 41.667%
Epoch 050: Loss: 0.079, Accuracy: 97.500%
Epoch 100: Loss: 0.063, Accuracy: 98.333%
Epoch 150: Loss: 0.056, Accuracy: 99.167%
Epoch 200: Loss: 0.052, Accuracy: 99.167%
--- 52.86402344703674 seconds ---
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (100.0%)
Example 1 prediction: Iris versicolor (99.9%)
Example 2 prediction: Iris virginica (99.9%)
```

Рисунок 4.40– Результати тесту при batch = 8

Результати навчання при визначенні партії рівної 16.

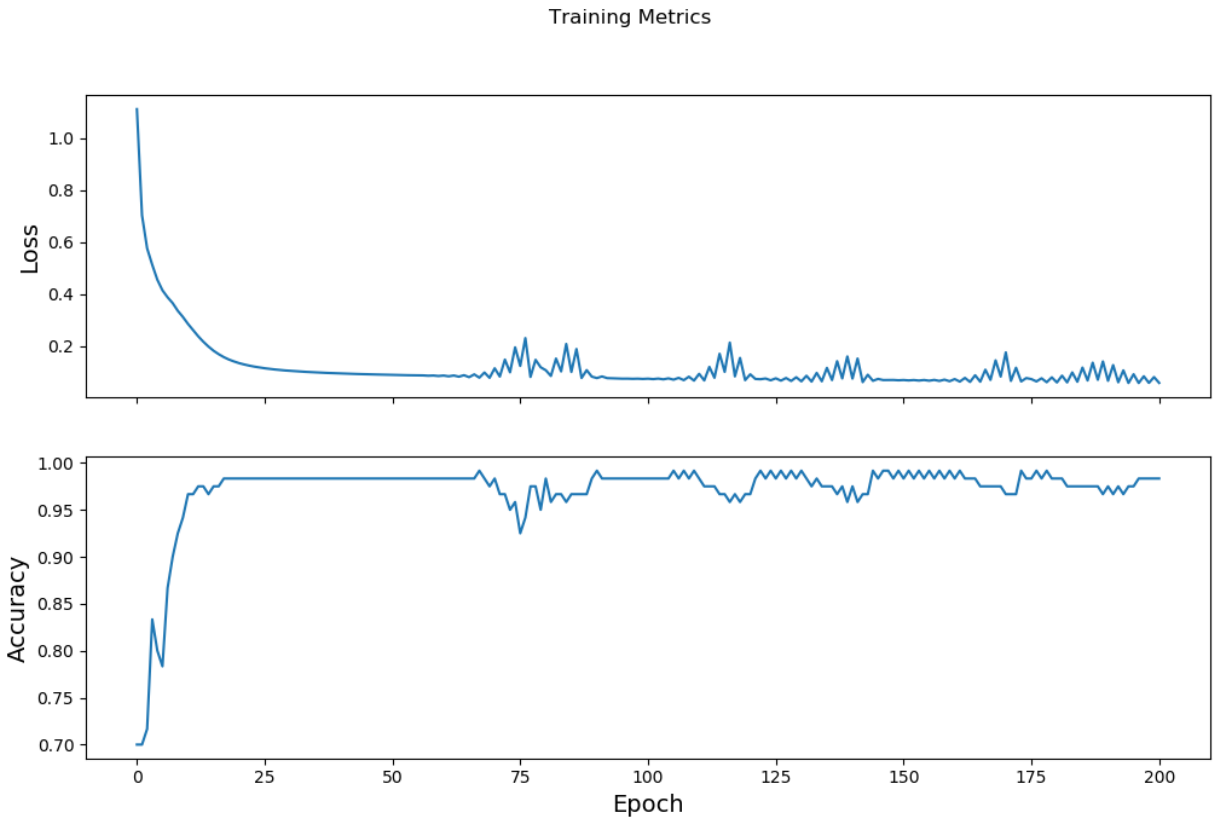

Рисунок 4.41 – Розмір параметра Batch = 16

```
Loss test: 2.2115578651428223
Step: 0, Initial Loss: 2.2115578651428223
Step: 1,         Loss: 1.8303236961364746
Epoch 000: Loss: 1.086, Accuracy: 55.833%
Epoch 050: Loss: 0.067, Accuracy: 98.333%
Epoch 100: Loss: 0.059, Accuracy: 98.333%
Epoch 150: Loss: 0.052, Accuracy: 98.333%
Epoch 200: Loss: 0.052, Accuracy: 98.333%
--- 26.616522550582886 seconds ---
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (100.0%)
Example 1 prediction: Iris versicolor (100.0%)
Example 2 prediction: Iris virginica (95.0%)
```

Рисунок 4.42 – Результати тесту при batch = 16

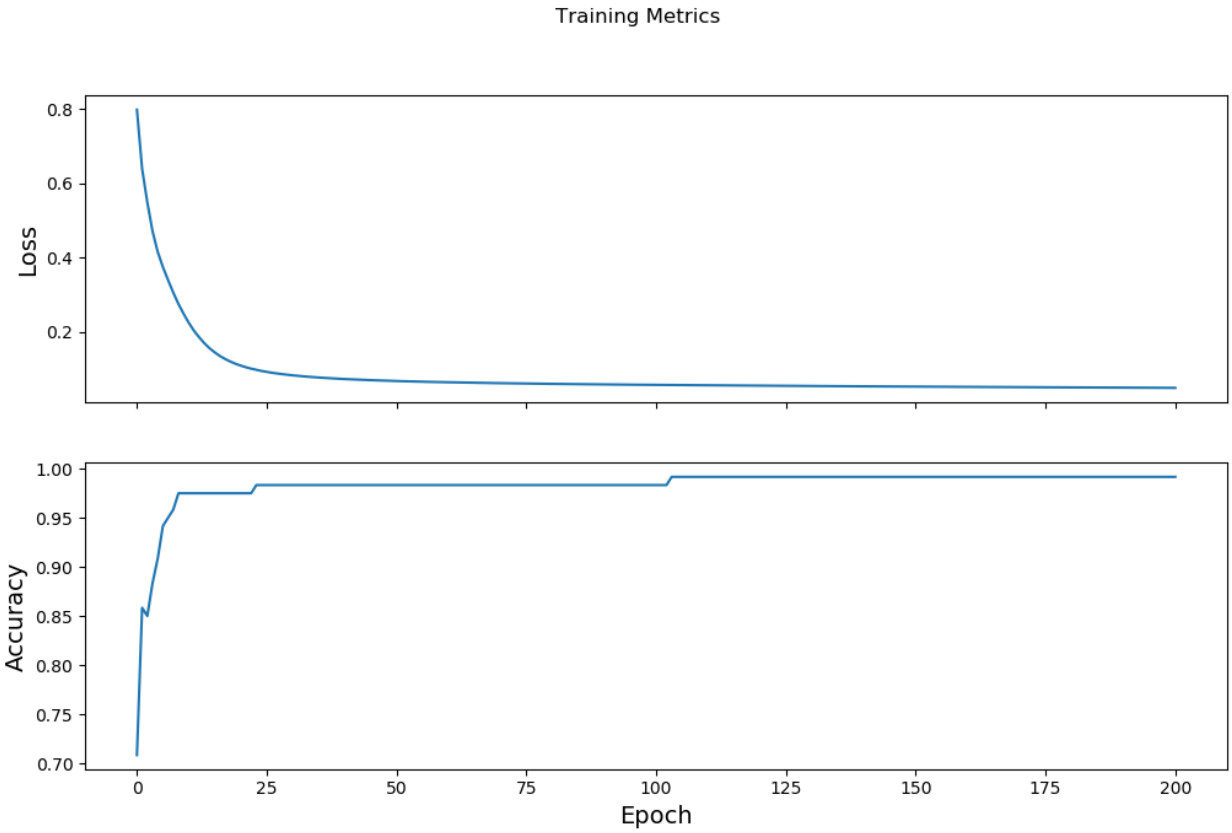

Рисунок 4.43 – Розмір параметра Batch = 32

```
Loss test: 2.687939167022705
Step: 0, Initial Loss: 2.687939167022705
Step: 1,         Loss: 2.386060953140259
Epoch 000: Loss: 1.837, Accuracy: 30.000%
Epoch 050: Loss: 0.093, Accuracy: 98.333%
Epoch 100: Loss: 0.062, Accuracy: 99.167%
Epoch 150: Loss: 0.055, Accuracy: 99.167%
Epoch 200: Loss: 0.052, Accuracy: 99.167%
--- 17.20798397064209 seconds ---
Test set accuracy: 96.667%
Example 0 prediction: Iris setosa (99.9%)
Example 1 prediction: Iris versicolor (99.9%)
Example 2 prediction: Iris virginica (99.4%)
```

Рисунок 4.44 – Результати тесту при batch = 32

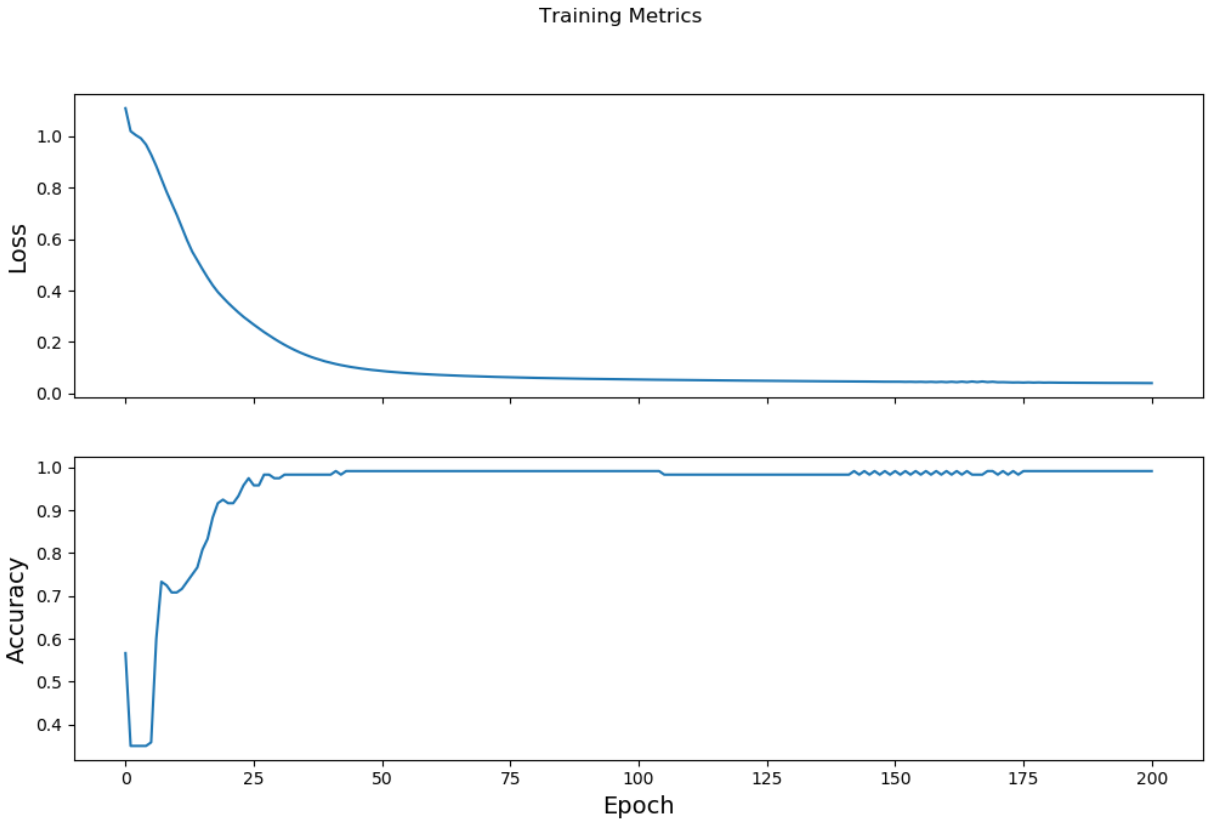

Рисунок 4.45 – Розмір параметра Batch = 64

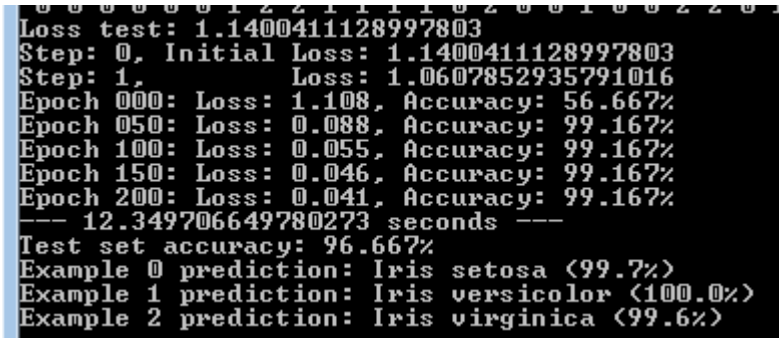

Рисунок 4.46 – Результати тесту при batch = 64

Показник batch = 64 є оптимальним за часом навчання, в даному випадку. У великих мережах можна досягнути скорочення значного скорочення часу навчання.

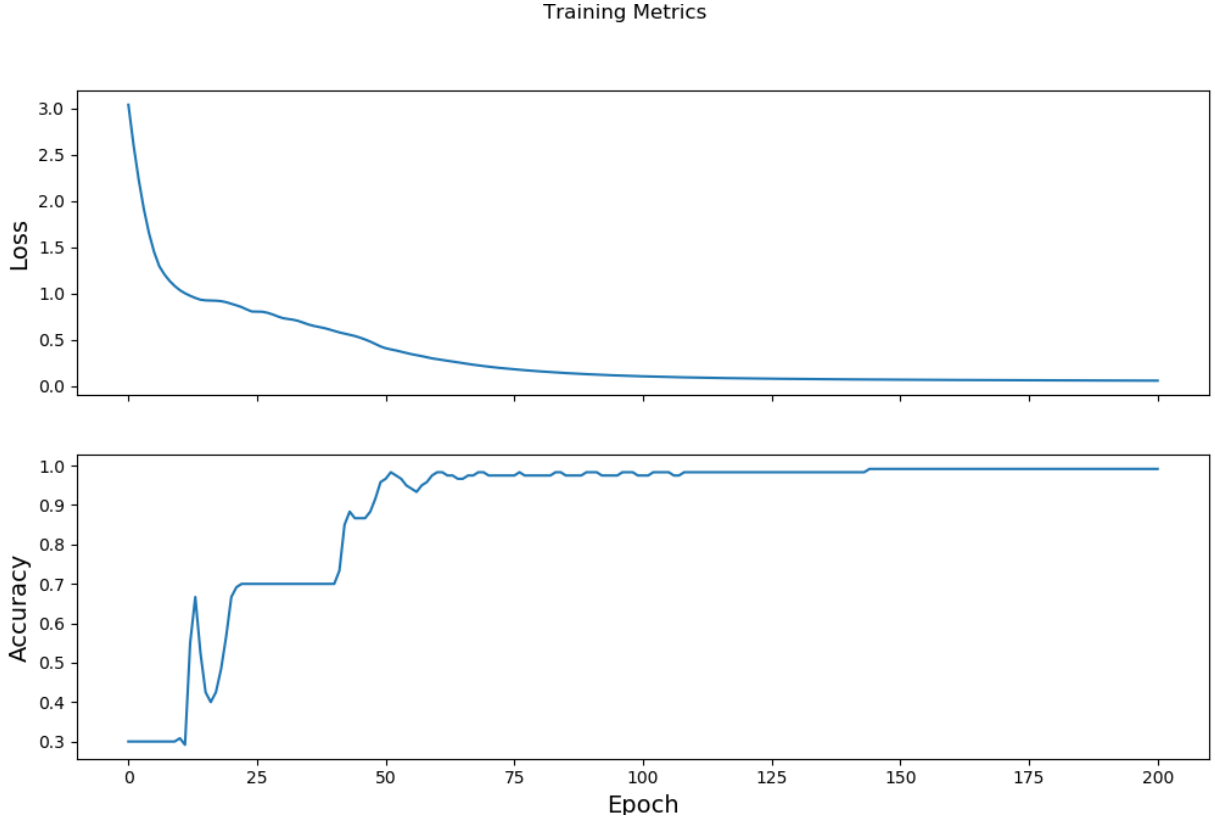

Рисунок 4.47 – Розмір параметра Batch = 128

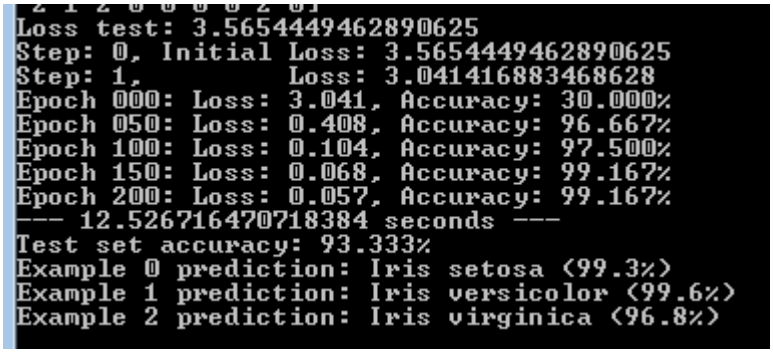

Рисунок 4.48 – Результати тесту при batch = 128

Значення batch = 128 не зменшело час навчання мережею Згідно графіку точності на рисунку 4.47 досягнення постановленого значення точності відбулося при пізніших епохах ніж при batch = 64.

\

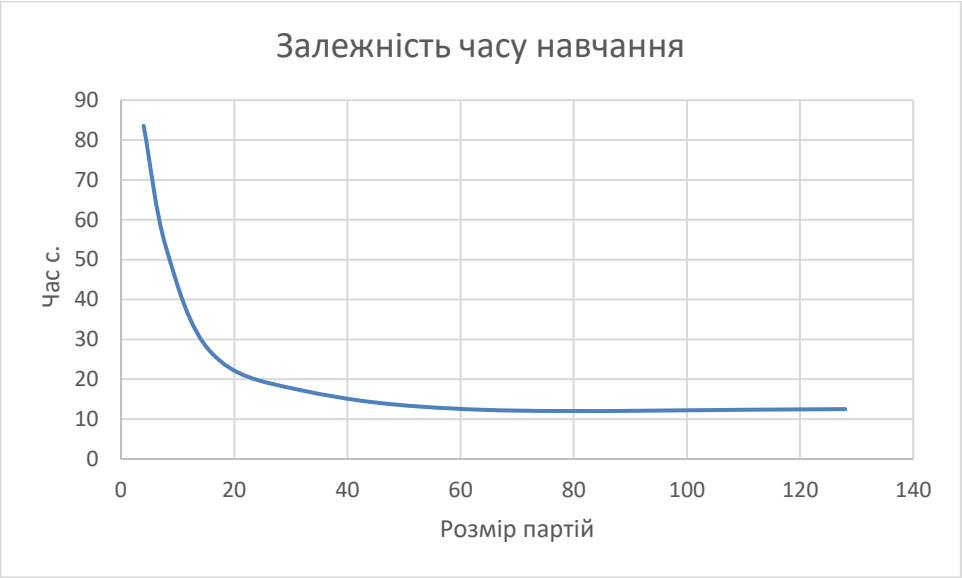

Рисунок 4.49– Графік залежності часу навчання від розміру партії

На рисунку 4.49 чітко можна прослідити взаємозалежність часу навчання від розміру партії, коли збільшення за 60 перестає явно впливати на параметр часу.

Висновки за розділом

Під час навчання великої мережі буде момент, коли модель припинить узагальнювати і почне вивчати статистичний шум у навчальному наборі даних. Збільшення кількості навчальних епох дозволяє досягнути більшої точності передбачення, що можна побачити на основі тестів. При використанні значень більших за 300 епох, показники точності перестали явно збільшуватися відносно числа повторів навчального набору. Тому ґрунтовно використовувати для тренування не більше 300 циклів повтору що дозволяє зекономити апаратно-часові ресурси машини. В той же час занадто мала кількість повторів не дозволить досягнути поставленого рівня точності, наприклад, значення менші за 100.

Для пошуку мінімальних точок використовуються різні алгоритми оптимізації та їх покращенні версії. Для випадку розпізнавання квітів ірису було розглянуто використання алгоритму ADAM та SGD.

У SGD для кожної ітерації випадковим чином вибирається лише один зразок із набору даних, шлях алгоритму для досягнення мінімумів зазвичай шумніший, ніж ваш типовий алгоритм Gradient Descent.

ADAM - це алгоритм оптимізації, який може використовувати замість класичної процедури стохастичного градієнта спуску для оновлення

ітераційних мережевих ваг на основі даних навчальних екземплярів. Проведення навчання з різними коефіцієнтами альфа показало що найоптимальніші значення для даної мережі наближенні до 0,001. Сам алгоритм більш точний порівняно з стохастичним градієнтним спуском. Тому використання саме ADAM буде кращим рішенням для представленої нейронної мережі.

Партії - це гіперпараметр, який визначає кількість проб, до яких потрібно опрацювати перед оновленням внутрішніх параметрів моделі. В кінці партії прогнози порівнюються з очікуваними вихідними змінними та обчислюється помилка. З цієї помилки алгоритм оновлення використовується для вдосконалення моделі, наприклад. рухатися вниз по градієнту помилки. Оптимальними значеннями для великих нейронних мереж є показники 32, 64, 128. Для даної мережі було розглянуто значення від 4 до 128, на основі отриманих даних найоптимальніший параметр для оптимізації часу навчання є 64, оскільки збільшення значення партії не зменшує час навчання .

ВИСНОВКИ

Розробка нейронної мережі з нуля досить складний процес. Якщо взяти до уваги велику кількість платформ то API створених для розробки та навчання нейронних мереж.

Типова нейронна мережа представляє собою від кількох десятків до сотень, тисяч або навіть мільйонів штучних нейронів, які називаються одиницями, розташованими за серією шарів, кожен з яких з'єднується з шаром з обох сторін. Деякі з них, відомі як одиниці введення, призначені для отримання різноманітних форм інформації з боку зовнішнього світу, які мережа спробує сприйняти, розпізнати чи інакше обробити. Інші підрозділи знаходяться на протилежній стороні мережі та сигналізують, як вона відповідає отриманій інформації; вони відомі як вихідні одиниці. Між пристроями вводу та вихідними блоками є один або декілька шарів прихованих блоків.

З боку програмного забезпечення розробленні потужні рішення для прискорення розробки компаніями Microsoft та Nvidia такі як Azure ML, бібліотеки Cuda. Головний принцип розробки платформ та бібліотек було поширення використання машинного навчання, спрощення вхідного порогу створення нових рішень.

Важливим фактором для навчання та тестування моделей нейронних мереж – підтримка на апаратному рівні GPU. Необхідність використання cuDNN 2.0 – GPU– прискорення бібліотека для задач глибокого навчання від NVIDIA, для масштабних проектів дозволяє задіяти потужність графічного прискорювача.

Апаратна частина надзвичайно важлива і правильно вибрати потрібну нелегко. Згідно даних практичних замірень виграш не дають поєднання багатьох дешевших відеокарт проти меншої кількості з потужніших, за умови складності реалізації програмного розпаралелення. Таке твердження справедливе для великих проектів нейроних мереж і найефективніше апаратне забезпечення для розробки ШНМ стануть відеокарти 900 та 1000 серії GTX компанії Nvidia.

TensorFlow – високомасштабована система машинного навчання, здатна працювати на різних платформах. Використання TensorFlow має широкий діапазон як розпізнання мови або вирішення завдань класифікації. Завдяки гнучкості системи, конструювання і навчання нейромереж значно пришвидшується.

Для кращого розуміння роботи TensorFlow було роглянуто приклад класифікації квітів ірису трьох окремих видів, тому що розгляд більшої

кількість видів значно ускладнює задачу. Як і багато аспектів машинного навчання, підбір найкращої форми нейронної мережі вимагає суміші знань та експериментів. Як правило, збільшення кількості прихованих шарів і нейронів зазвичай створює більш потужну модель, для якої потрібні додаткові дані для ефективного навчання. Основна концепція розпізнавання грунтується на отриманні прогнозу ймовірності, що вхідні дані відповідають конкретному виду ірису.

Завершаючим став етап тестування даних, що містять функцію але не мітку, тобто прикладах які не використовувалися при навчанні. Були побудовані графіки похибки які показували незначні величини через перенавчання мережі.

Під час навчання великої мережі буде момент, коли модель припинить узагальнювати і почне вивчати статистичний шум у навчальному наборі даних. Збільшення кількості навчальних епох дозволяє досягнути більшої точності передбачення, що можна побачити на основі тестів. При використанні значень більших за 300 епох, показники точності перестали явно збільшуватися відносно числа повторів навчального набору. Тому ґрунтовно використовувати для тренування не більше 300 циклів повтору що дозволяє зекономити апаратно-часові ресурси машини. В той же час занадто мала кількість повторів не дозволить досягнути поставленого рівня точності, наприклад, значення менші за 100.

Для пошуку мінімальних точок використовуються різні алгоритми оптимізації та їх покращенні версії. Для випадку розпізнавання квітів ірису було розглянуто використання алгоритму ADAM та SGD.

У SGD для кожної ітерації випадковим чином вибирається лише один зразок із набору даних, шлях алгоритму для досягнення мінімумів зазвичай шумніший, ніж ваш типовий алгоритм Gradient Descent.

ADAM - це алгоритм оптимізації, який може використовувати замість класичної процедури стохастичного градієнта спуску для оновлення ітераційних мережевих ваг на основі даних навчальних екземплярів. Проведення навчання з різними коефіцієнтами альфа показало що найоптимальніші значення для даної мережі наближенні до 0,001. Сам алгоритм більш точний порівняно з стохастичним градієнтним спуском. Тому використання саме ADAM буде кращим рішенням для представленої нейронної мережі.

Партії - це гіперпараметр, який визначає кількість проб, до яких потрібно опрацювати перед оновленням внутрішніх параметрів моделі. В кінці партії прогнози порівнюються з очікуваними вихідними змінними та обчислюється

помилка. З цієї помилки алгоритм оновлення використовується для вдосконалення моделі, наприклад. рухатися вниз по градієнту помилки. Оптимальними значеннями для великих нейронних мереж є показники 32, 64, 128. Для даної мережі було протестованно значення від 4 до 128, на основі отриманих даних найоптимальніший параметр для оптимізації часу навчання є 64, оскільки збільшення значення партії не зменшує час навчання .

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Описання розробки Alphago компанією Google [Електронний ресурс] // googleblog.com – 2016. – Режим доступу до ресурсу :https://research.googleblog.com /2016/01/alphago-mastering-ancient-game-of-go.html

2. Нейронні мережі для початківців[Електронний ресурс] // habrahabr.ru – 2016. – Режим доступу до ресурсу https://habrahabr.ru/post/312450/

3. Нейроні мережі, основні моделі [Електронний ресурс] – Режим доступу: http://www.nncourse.chat.ru/course.pdf

4. Найпростіший вид штучних нейронних мереж – персептрони [Електронний ресурс] – Режим доступу: https://neuralnet.info/chapter/%D0%BF%D0%B5%D1%80%D1%81%D0%B5%D0%BF%D1%82%D1%80%D0%BE%D0%BD%D1%8B/#C#

5. Рекурентні Електронний ресурс] – Режим доступу: https://cyberleninka.ru/article/v/navchannya-rekurentnih-neyronnih-merezh-metodom-psevdoregulyarizatsiyi-dlya-bagatokrokovogo-prognozuvannya-chasovih-ryadiv

6. Динамічні рекурентні нейроні мережі [Електронний ресурс] – Режим доступу: http://www.immsp.kiev.ua/publications/articles/2009/2009_3/Reznik_03_2009.pdf

7. Нейронні мережі на основі радіально-симетричних функцій [Електронний ресурс] – Режим доступу: http://neuronus.com/theory/960-nejronnye-seti-na-osnove-radialno-simmetrichnykh-funktsij.html

8. Опис сгорткових нейроних мереж [Електронний ресурс] – Режим доступу: http://ru.datasides.com/code/cnn-convolutional-neural-networks/

9. Лекція по сгортковим нейроним мережам [Електронний ресурс] – Режим доступу: http://www.machinelearning.ru/wiki/images/1/1b/DL16_lecture_3.pdf

10. Абетка ШИ «Рекурентні нейроні мережі» [Електронний ресурс] – Режим доступу: https://nplus1.ru/material/2016/11/04/recurrent-networks

11. Глубоке навчання на відеокартах Nvidia [Електронний ресурс] – Режим доступу:http://www.nvidia.ru/object/blog-nvidia-digits-devbox-ru.html

12. Tensorflow від Google [Електронний ресурс] – Режим доступу:https://googleblog.blogspot.com/2015/11/tensorflow-smarter-machine-learning-for.html

13. Графіки замірів прискорення [Електронний ресурс] – Режим доступу:https://developer.nvidia.com/cudnn

14.Опис API Keras [Електронний ресурс] – Режим доступу:https://keras.io/

15. Введення в Microsoft Azure[Електронний ресурс] – Режим доступу: https://msdn.microsoft.com/ru-ru/magazine/dn781358.aspx

16. Заміри швидкості апаратного забеспечення для глубокого навчання [Електронний ресурс] – Режим доступу: http://timdettmers.com/2017/04/09/which-gpu-for-deep-learning

17. Ресурси для розробників CUDA [Електронний ресурс] – Режим доступу:https://developer.nvidia.com/cuda-zone

18. Бібліотека cuDNN для швидкого навчання нейронних мереж на GPU [Електронний ресурс] – Режим доступу: https://www.asozykin.ru/deep_learning/2017/03/20/cuDNN-with-theano-and-keras.html

19. Хайкин С. Нейроні мережі повний курс [пер. с англ] С. Хайкин 2-е вид. — М.: Вильямс, 2006. — 1104 с.

20. База даних класифікації ірисів за Фішером [Електронний ресурс] – Режим доступу: https://archive.ics.uci.edu/ml/datasets/iris

21. Опис алгориму оптимізації ADAM [Електронний ресурс] – Режим доступу: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

22. Магазин розробників костюму захоплення руху [Електронний ресурс] – Режим доступу: https://www.nansense.com/suits/

23. Стаття про результат http://theorangeduck.com/page/phase-functioned-neural-networks-character-control

24. Кореляція і види статистичних досліджень: [Електронний ресурс] – Режим доступу:https://statpsy.ru/correlation/correlation/

25. Різниця між пакетними та епоховими нейроними мережами [Електронний ресурс] – Режим доступу: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

26. Опис роботи стохастичного градієнта [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/

27. Порогова функція softmax [Електронний ресурс] – Режим доступу: https://towardsdatascience.com/softmax-function-simplified-714068bf8156

28. Явище перенавчання нейронних мереж [Електронний ресурс] – Режим доступу: https://machinelearningmastery.com/early-stopping-to-avoid-overtraining-neural-network-models/

29. Введення до функції SULE’s [Електронний ресурс] – Режим доступу: https://towardsdatascience.com/gentle-introduction-to-selus-b19943068cd9