Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет електроніки та комп’ютерних технологій
Кафедра оптоелектроніки та інформаційних технологій

Пояснювальна записка
до курсової роботи

“ Аналіз хаотичних станів у згорткових нейронних мережах за різними методами оптимізації процесу навчання”

Виконав:
студент групи ФЕІм-13с
Кричевець Олександр Ілліч
Науковий керівник:
Свелеба С. А.

Львів – 2023

Зміст

1. Поняття хаосу.

2. Механізми переходу до хаотичних станів.

3.Оптимізація навчання нейронної мережі. Алгоритм градієнтного спуску.

4. Огляд оптимізаційних методів.

5. Візуалізація та порівняння оптимізаційних методів.

1.1. Поняття хаосу

Хаос — це поняття, яке широко використовується в різних областях, включаючи математику, фізику та загальну теорію систем. Це відноситься до стану випадковості та непередбачуваності в системі, яка дотримується детермінованих законів.

Різні галузі знань виділяють різні види хаосу:

- Інформаційна
- Термодинамічна
- Диференційна
- Культурна

У математиці теорія хаосу має справу з поведінкою динамічних систем, які дуже чутливі до початкових умов. Навіть за незначних змін початкових умов майбутня поведінка системи може кардинально змінитися. Хаотичні системи демонструють так званий «ефект метелика», коли невеликі зміни початкових умов можуть призвести до значних відмінностей у результатах з часом.

У фізиці хаос часто виникає в складних системах, які мають велику кількість взаємодіючих компонентів. Приклади включають динаміку рідини, погодні умови та поведінку небесних тіл. Хаотичні системи можуть демонструвати нерегулярну, здавалося б, випадкову поведінку, що ускладнює їх довгострокові прогнози.

Хаотичні системи характеризуються відсутністю довгострокової передбачуваності, оскільки навіть незначні зміни або невизначеності початкових умов або параметрів системи можуть призвести до різних результатів. Однак хаос не є синонімом випадковості. Хаотичні системи все ще дотримуються детермінованих правил, але їхня чутливість до початкових умов

робить їхню поведінку випадковою або непередбачуваною.

Вивчення хаосу дало цінну інформацію про розуміння складних явищ і має практичне застосування в різних сферах, таких як прогнозування погоди, проектування безпечних систем зв'язку та розуміння динаміки біологічних систем.

1.2. Механізми переходу до хаотичних станів

Аналіз механізмів переходу порядку до хаосу в реальних системах і різних моделях виявив універсальність відносно кількох сценаріїв. Перехід до хаосу можна представити у вигляді біфуркаційної діаграми (термін «біфуркація» використовується для позначення якісної перебудови системи з появою нового режиму її поведінки).

Входження системи в непередбачуваний режим описується каскадом біфуркацій, що наступають одна за одною. Каскад біфуркацій призводить послідовно до появи вибору між двома рішеннями, потім чотирма і так далі, система починає коливатися в хаотичному режимі послідовного подвоєння числа можливих значень.

1.3. Оптимізація навчання нейронної мережі. Алгоритм градієнтного спуску.

Оптимізатори – важливий елемент архітектури нейронних мереж. Вони відіграють важливу роль у процесі навчання нейронних мереж, допомагаючи їм робити все більш точні прогнози. Оптимізатори визначають оптимальний набір параметрів моделі, таких як вага та зсув, щоб при вирішенні конкретної задачі модель видавала найкращі результати. Найпоширенішою технікою оптимізації, яка використовується більшістю нейронних мереж, є алгоритм градієнтного спуску. Більшість популярних бібліотек глибокого навчання, наприклад PyTorch і Keras, мають безліч вбудованих оптимізаторів, що базуються на використанні алгоритму градієнтного спуску, наприклад, SGD, Adadelta, Adagrad, RMSProp, Adam та ін.

Розглянемо на 3D-зображенні, як працює стандартний алгоритм градієнтного спуску:

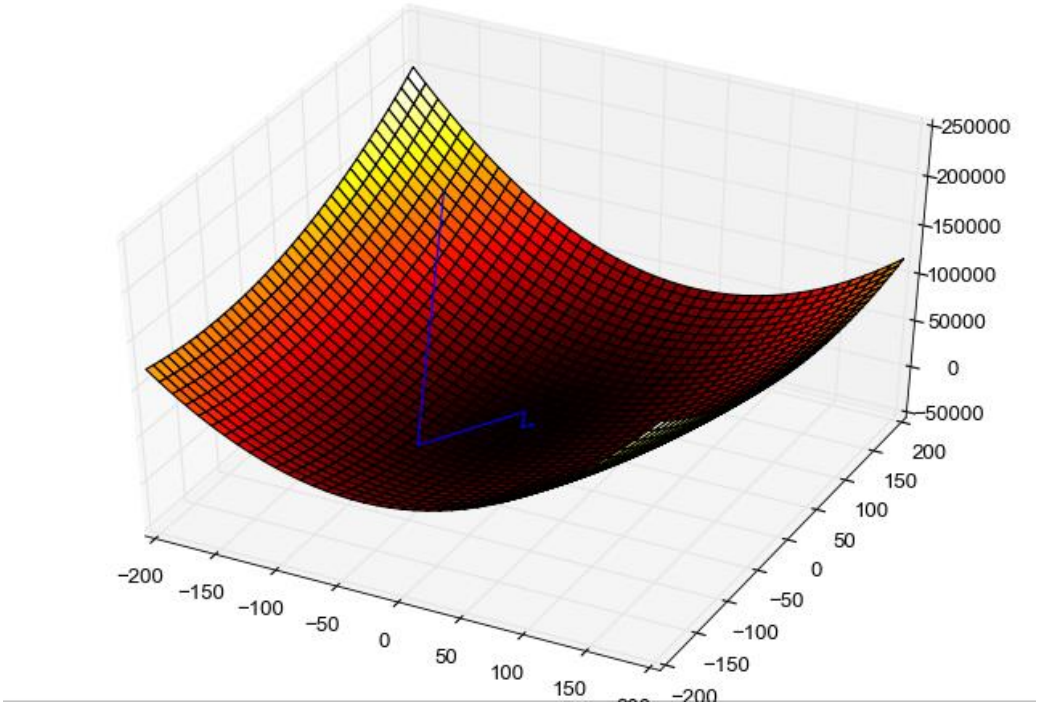

Рис 1.1 - Крива втрат для алгоритму градієнтного спуску

На малюнку показано мережу з двома ваговими параметрами:

- На горизонтальній площині розміщуються дві осі для ваг w1 і w2 відповідно.
- На вертикальній осі відкладаються значення втрат.

Форма кривої є відображенням “ландшафту втрат” у нейронній мережі. Крива представляє втрати для різних значень ваг при фіксованому набору вхідних даних.

Синя лінія відповідає траєкторії алгоритму градієнтного спуску при оптимізації:

- Робота алгоритму починається з вибору деяких випадкових значень ваг та обчислення втрат.
- Після кожної ітерації вагові значення будуть оновлюватися і алгоритм

буде рухатись вздовж кривої до наступної точки.

- Зрештою, алгоритм досягає мети, тобто входить у нижню точку кривої — у точку з найнижчим значенням втрат.

1.4. Проблеми при оптимізації градієнтного спуску

Локальні мінімуми. На стандартній кривій втрат, крім глобального мінімуму, може трапитися безліч локальних мінімумів. Головним завданням алгоритму градієнтного спуску, як випливає з його назви, є спуск дедалі «нижче». Але варто йому спуститися до локального мінімуму — і піднятися звідти часто стає непосильним завданням. Алгоритм може просто застрягти у локальному мінімумі, так і не потрапивши на глобальний мінімум.

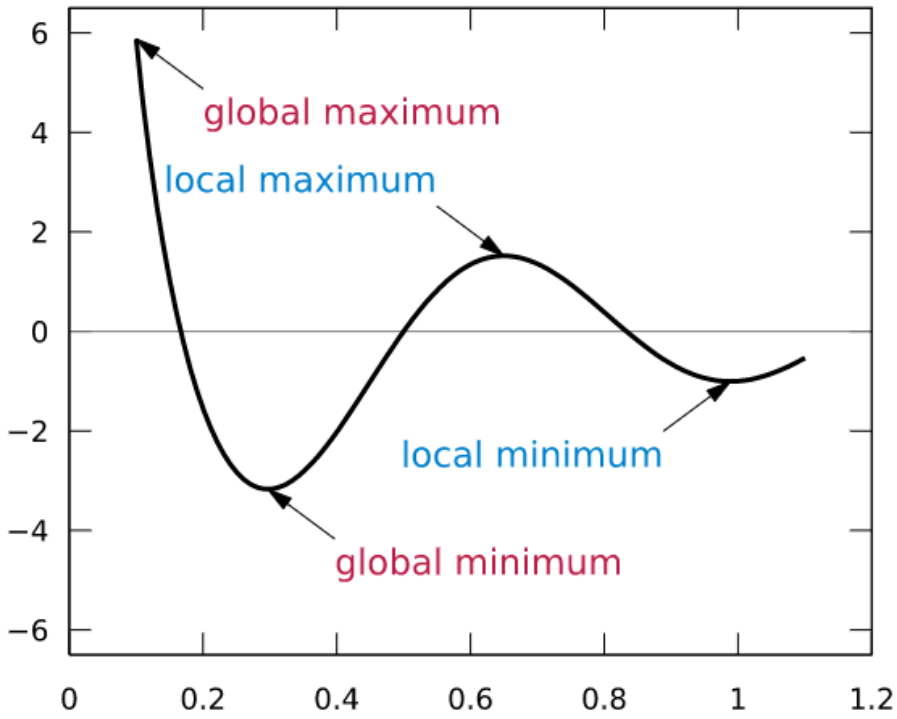

Рис 1.2 – Локальні і глобальні мінімуми

Сідлові точки. Ще однією важливою проблемою є проходження алгоритмом "сідлових точок". Сідловою називають точку, в якій в одному напрямку, що відповідає одному параметру, крива знаходиться на локальному мінімумі; у другому напрямку, відповідному іншому параметру, крива знаходиться на локальному максимумі. Небезпека в тому, що область, що безпосередньо оточує

сідлову точку, як правило, досить плоска, вона нагадує плато. Плоска область означає майже нульові градієнти. Оптимізатор починає коливатися навколо сідлової точки в напрямку першого параметра, не здогадуючись спуститися вниз по ухилу в напрямку другого параметра.

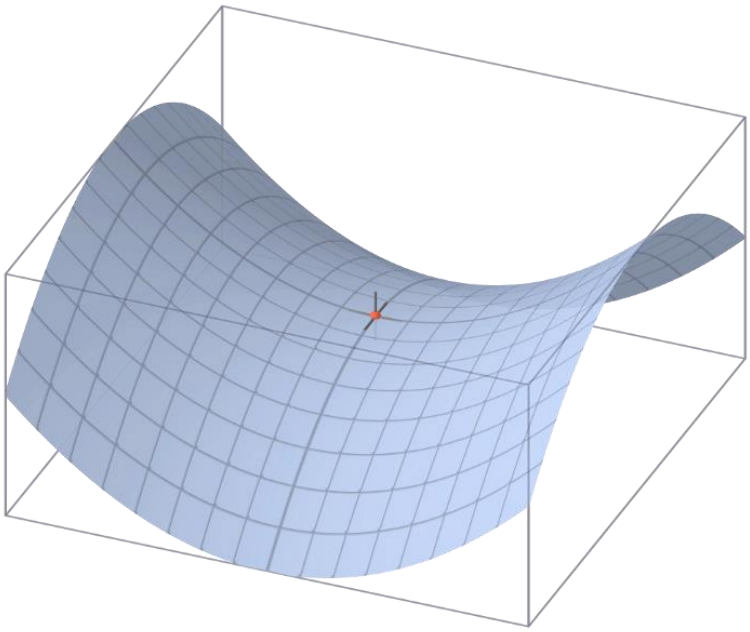

Рис 1.3 – Сідлова точка

Яри. Яр – це протяжна вузька долина, що має крутий ухил в одному напрямку (тобто по сторонах долини) і плавний ухил в іншому (тобто вздовж долини). Досить часто яр призводить до мінімуму. Оскільки навігація за такою формою є досить складною, таку криву часто називають патологічним викривленням (Pathological Curvature).

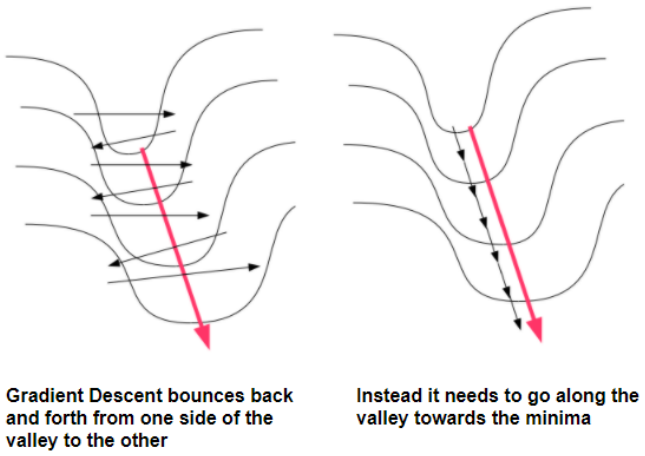

Рис 1.4 – Яри

1.5. Огляд оптимізаційних методів

Momentum (Імпульс). SGD (стохастичний градієнтний спуск) має проблеми з навігацією по ярах, тобто областям, де поверхня вигинається набагато крутіше в одному вимірі, ніж в іншому , що часто зустрічається навколо локальних оптимумів. У таких випадках SGD коливається вздовж схилів ущелини, лише нерішуче просуваючись по дну до місцевого оптимуму (Рис 1.6)

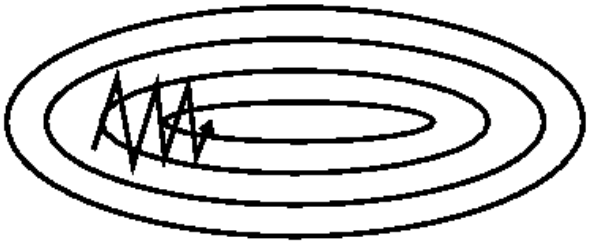

Рис 1.6 – SGD без імпульсу

Momentum - це метод, який допомагає прискорити SGD у відповідному напрямку та гасить коливання, як можна побачити на зображенні (Рис 1.7). Він робить це шляхом додавання частки γγ вектора оновлення минулого кроку до поточного вектора оновлення.

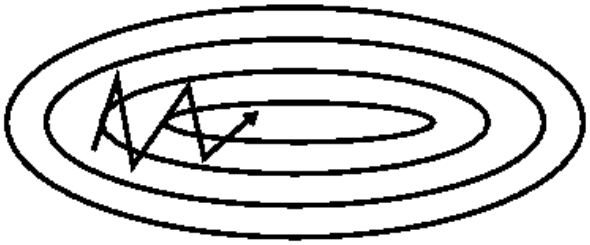

Рис 1.7 – SGD з імпульсом

По суті, використовуючи імпульс, ми штовхаємо м’яч з пагорба. М’яч накопичує імпульс, коли котиться вниз, стаючи все швидшим і швидшим на шляху (поки не досягне своєї кінцевої швидкості, якщо є опір повітря, тобто γ<1γ<1). Те ж саме відбувається з нашими оновленнями параметрів: термін імпульсу збільшується для розмірів, градієнти яких спрямовані в однакових напрямках, і зменшує оновлення для розмірів, градієнти яких змінюють напрямки. У результаті ми отримуємо швидшу конвергенцію та зменшуємо коливання.

NAG (Nesterov accelerated gradient). Однак м'яч, який котиться наосліп, слідуючи схилу, є незадовільним для нас результатом. Ми хотіли б мати розумніший м’яч, який має уявлення про те, куди він рухається, щоб він знав, що потрібно сповільнитися, перш ніж пагорб знову підніметься.

Прискорений градієнт Нестерова (NAG) — це спосіб надати нашому імпульсу такого роду передбачення. Ми знаємо, що буде використовуватися наш імпульс γvt−1γvt−1 для переміщення параметрів θθ. Таким чином, обчислення θ−γvt−1θ−γvt−1 дає нам наближення до наступної позиції параметрів, приблизне уявлення про те, де будуть наші параметри. Тепер ми можемо ефективно дивитися наперед, обчислюючи градієнт до наших поточних параметрів θθ.

Ми встановлюємо значення імпульсу γγ приблизно 0,9. Momentum спочатку обчислює поточний градієнт (маленький синій вектор на Рис. 1.8), а потім робить великий стрибок у напрямку оновленого накопиченого градієнта (великий синій вектор), NAG спочатку робить великий стрибок у напрямку попереднього накопиченого градієнта (коричневий вектор), вимірює градієнт, а потім вносить корекцію (червоний вектор), що призводить до повного оновлення NAG (зелений вектор). Це випереджувальне оновлення запобігає надшвидкій роботі та забезпечує підвищену швидкість реагування, що значно підвищило продуктивність RNN у ряді завдань.

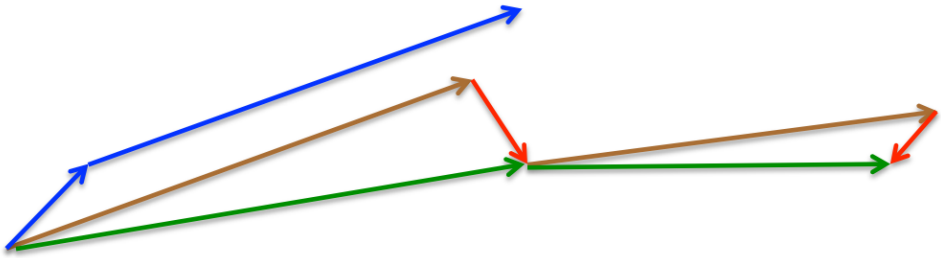

Рис. 1.8 – NAG

AdaGrad — це алгоритм оптимізації на основі градієнта, суть якого полягає в адаптації швидкості навчання до параметрів, виконуючи менші оновлення

(тобто низька швидкість навчання) для параметрів, пов’язаних із функціями, що часто зустрічаються, і більші оновлення (тобто висока швидкість навчання) для параметрів, пов’язаних із функціями, які рідко зустрічаються. З цієї причини він добре підходить для роботи зі sparse data. Було виявлено, що AdaGrad значно покращив надійність SGD і його часто використовують для навчання великих нейронних мереж Google, які, серед іншого, навчилися розпізнавати котів у відео на Youtube.

Основною слабкістю AdaGrad є накопичення квадратів градієнтів у знаменнику:

оскільки кожен доданий член додатний, накопичена сума продовжує зростати під час навчання. Це, у свою чергу, призводить до того, що швидкість навчання зменшується і зрештою стає нескінченно малою, після чого алгоритм більше не може отримати додаткові знання. Наступні алгоритми спрямовані на усунення цієї вади.

AdaDelta. AdaDelta — це метод стохастичної оптимізації, який дозволяє використовувати метод швидкості навчання для SGD для кожного виміру. Це розширення AdaGrad, яке прагне зменшити його агресивну, монотонно знижену швидкість навчання. Замість накопичення всіх минулих градієнтів у квадраті AdaDelta обмежує вікно накопичених минулих градієнтів фіксованим розміром w. Замість неефективного зберігання вагами попередніх квадратичних градієнтів, сума градієнтів рекурсивно визначається як спадне середнє всіх минулих квадратичних градієнтів. Середнє значення $E[g2]_t$ на кроці часу t тоді залежить лише від попереднього середнього та поточного градієнта:

$$E\left[g^2\right]_t = \gamma E\left[g^2\right]_{t-1} + (1-\gamma) g_t^2$$

Зазвичай, γ приблизно дорівнює 0,9. Перезапис оновлень SGD у векторі оновлення параметрів:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

AdaDelta приймає форму:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Головна перевага AdaDelta полягає в тому, що нам не потрібно встановлювати швидкість навчання за замовчуванням.

RMSprop. RMSprop — це неопублікований адаптивний метод швидкості навчання, запропонований Джеффом Хінтоном у лекції його курсу Coursera.

RMSprop і Adadelta були розроблені незалежно один від одного приблизно в той самий час, що випливає з необхідності вирішити проблему радикального зниження швидкості навчання Adagrad. RMSprop фактично ідентичний першому вектору оновлення Adadelta, який ми отримали вище. RMSprop також ділить швидкість навчання на експоненціально спадаюче середнє квадратичних градієнтів. Хінтон пропонує встановити γγ на 0,9, тоді як підходяще значення за замовчуванням для швидкості навчання ηη становить 0,001.

Adam (Adaptive Moment Estimation). Ще один метод, який обчислює адаптивні швидкості навчання для кожного параметра. На додаток до зберігання експоненціально спадного середнього минулих квадратичних градієнтів , як Adadelta та RMSprop, Адам також зберігає експоненціально спадне середнє минулих градієнтів, подібно до імпульсу. Тоді як імпульс можна розглядати як кулю, що біжить по схилу, Адам поводиться як важка куля з тертям, яка, таким чином, віддає перевагу рівним мінімумам на поверхні помилки.

AMSGrad. Оскільки методи адаптивної швидкості навчання стали нормою навчання нейронних мереж, практики помітили, що в деяких випадках, наприклад розпізнавання об’єктів або машинного перекладу вони не в змозі наблизитися до оптимального рішення.

Редді та інші вчені формалізували це питання та вказали на експоненціальне ковзне середнє минулих квадратичних градієнтів як причину поганої поведінки адаптивних методів навчання. Введення експоненціального середнього було добре вмотивованим: воно повинно запобігти тому, щоб темпи навчання ставали нескінченно малими в ході навчання, що є ключовим недоліком алгоритму Adagrad. Однак ця короткочасна пам’ять про градієнти стає перешкодою в інших випадках.

У налаштуваннях, де Адам збігається до неоптимального рішення, було помічено, що деякі minibatch забезпечують великі та інформативні градієнти, але оскільки ці minibatch трапляються рідко, експоненціальне усереднення зменшує їхній вплив, що призводить до поганої збіжності. Автори надають приклад

простої задачі опуклої оптимізації, де така ж поведінка може спостерігатися для Адама. Щоб виправити цю поведінку, автори пропонують новий алгоритм AMSGrad, який використовує максимум попередніх квадратичних градієнтів а не експоненціальне середнє для оновлення параметрів. Замість безпосереднього використання vtvt (або його виправленої зміщенням версії), тепер ми використовуємо попередній vt−1vt−1, якщо він більший за поточний.

Таким чином, AMSGrad не призводить до збільшення розміру кроку, що дозволяє уникнути проблем, яких зазнає Adam. Для простоти автори також видаляють debiasing step, який ми бачили в Adam. Автори спостерігають покращену продуктивність порівняно з Adam на малих наборах даних і на CIFAR-10. Однак інші експерименти показують подібні або гірші результати, ніж Adam.

Nadam. Nadam поєднує елементи двох популярних алгоритмів оптимізації, прискореного градієнта Нестерова (NAG) і адаптивної оцінки моменту (Adam), щоб досягти швидшої конвергенції та кращої продуктивності. Він використовує підхід NAG для розрахунку майбутнього напрямку градієнта та підхід Адама для коригування швидкості навчання для кожного параметра. Подібно до Adam, Nadam адаптує швидкість навчання для кожного параметра окремо на основі оцінок першого та другого моментів градієнтів. Він підтримує окрему швидкість навчання для кожного параметра, дозволяючи алгоритму динамічно регулювати розмір кроку під час навчання. Nadam виконує корекцію зміщення для першого та другого моментів градієнтів. Ця корекція компенсує зсув, який виникає на етапах ініціалізації навчання. Це допомагає стабілізувати процес оптимізації та дає точніші оцінки моментів. Nadam набув популярності та широко використовується для навчання нейронних мереж. Він продемонстрував ефективну роботу з різними завданнями, включаючи класифікацію зображень, обробку природної мови та навчання з підкріпленням. Хоча Nadam є популярним алгоритмом оптимізації, він має деякі потенційні проблеми та обмеження:

- Чутливість до швидкості навчання. Продуктивність Nadam може залежати від вибору швидкості навчання. Встановлення занадто високої швидкості навчання може призвести до нестабільного навчання або розбіжності, тоді як встановлення занадто низької може призвести до повільної конвергенції або застрягання на неоптимальних рішеннях. Пошук відповідної швидкості навчання може бути складним і може вимагати ретельного налаштування.
- Обчислювальна складність. Nadam вимагає додаткових обчислень порівняно з простішими алгоритмами оптимізації, такими як стохастичний градієнтний спуск (SGD). Він підтримує оцінки першого та другого моментів для кожного параметра та виконує корекцію зміщення. Ці додаткові обчислення можуть призвести до збільшення обчислювальної складності, особливо для великомасштабних моделей або наборів даних.
- Чутливість до гіперпараметрів. Nadam, як і багато інших алгоритмів оптимізації, має гіперпараметри, які необхідно встановити відповідним чином для оптимальної продуктивності. Ці гіперпараметри включають швидкість навчання, термін імпульсу та експоненціальні швидкості спаду для оцінок моменту. Чутливість Nadam до цих гіперпараметрів може зробити його більш складним для точного налаштування та може вимагати значних експериментів для досягнення хороших результатів.
- Потенційне перенавчання. Як і будь-який алгоритм оптимізації, Nadam може постраждати від перенавчання, якщо його не врегулювати належним чином. Перенавчання відбувається, коли модель стає надто складною та починає адаптувати навчальні дані, що призводить до поганого узагальнення тестових даних. Такі методи нормалізації як weight decay або dropout можуть бути використані, щоб уникнути проблеми перенавчання.

Інші потимізатори. Після AMSGrad було запропоновано ряд інших оптимізаторів. Серед них AdamW, який фіксує зниження ваги в Адамі; QHAdam , який усереднює стандартний крок SGD з кроком SGD імпульсу; та AggMo, який об'єднує кілька членів імпульсу γγ.

1.6. Візуалізація та порівняння оптимізаційних методів

Наступні дві анімації (автор: Алек Редфорд) надають деяке уявлення про поведінку більшості представлених методів оптимізації.

На Рис. 1.9 видно їхню поведінку на контурах поверхні втрат (функція Біла) з часом. Зауважте, що Adagrad, Adadelta та RMSprop майже відразу прямують у правильному напрямку та зближуються так само швидко, тоді як Momentum та NAG збиваються з траси, створюючи образ м'яча, що котиться з пагорба. NAG, однак, здатний швидко коригувати свій курс завдяки підвищеній чутливості, дивлячись вперед і прямуючи до мінімуму.

Рис. 1.10 показує поведінку алгоритмів у сідловій точці, тобто точці, де один вимір має додатний нахил, а інший вимір має від'ємний нахил, що створює труднощі для SGD, як вже зазначено раніше. Видно, що SGD, Momentum і NAG важко порушити симетрію, хоча двом останнім врешті-решт вдається уникнути сідлової точки, тоді як Adagrad, RMSprop і Adadelta швидко спускаються вниз по негативному схилу.

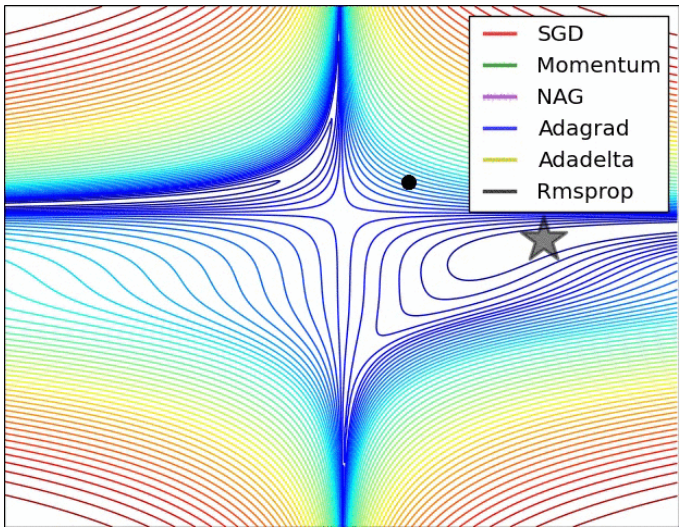

Рис 1.9 – Візуалізація оптимізаційних методів

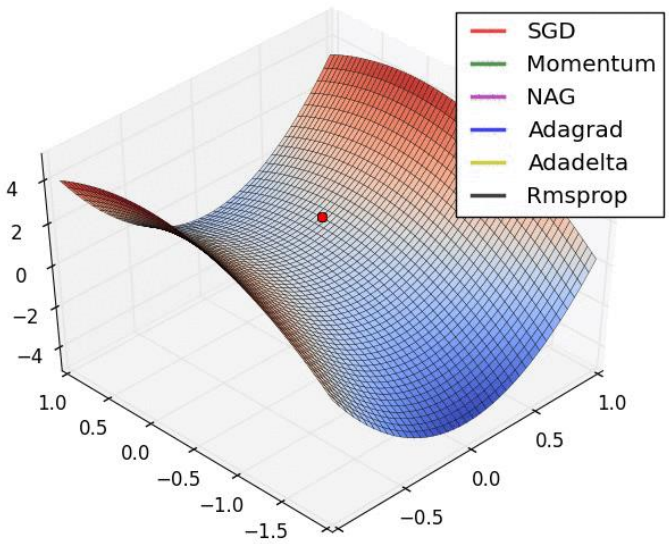

Рис. 1.10 - Візуалізація оптимізаційних методів у сідловій точці

2. Практична частина

2.1. Модулі Keras і TensorFlow

TensorFlow і Keras — популярні бібліотеки Python, які використовуються для завдань глибокого та машинного навчання. TensorFlow — це бібліотека з відкритим кодом, розроблена Google Brain, а Keras — це високорівневий API для нейронних мереж, який працює на основі TensorFlow (а також інших фреймворків глибокого навчання). TensorFlow надає гнучку структуру для створення та навчання різних моделей машинного навчання, включаючи нейронні мережі.

Ключові особливості TensorFlow:

- Обчислювальні графіки. TensorFlow використовує обчислювальний графік для визначення та ефективного виконання складних математичних операцій.
- Автоматична диференціація. TensorFlow може автоматично обчислювати градієнти для алгоритмів оптимізації, таких як градієнтний спуск, що полегшує навчання моделей.
- Підтримка графічних процесорів. TensorFlow легко інтегрується з графічним

процесором, дозволяючи швидше навчатися та робити висновки на паралельному обладнанні.

- Розподілені обчислення. TensorFlow підтримує розподілені обчислення, дозволяючи навчати моделі на кількох пристроях або машинах.
- TensorBoard. TensorFlow поставляється з TensorBoard, набором інструментів для візуалізації, який допомагає відстежувати та аналізувати навчальний процес.

Ключові особливості Keras:

- Зручний синтаксис. Keras пропонує інтуїтивно зрозумілий і простий синтаксис для визначення моделей нейронних мереж, що робить його доступним для початківців.
- Модульність. Keras підтримує модульний підхід, що дозволяє складати рівні та з’єднувати їх для створення складних мережевих архітектур.
- Легке налаштування моделі: Keras дозволяє легко налаштовувати моделі, додаючи або змінюючи шари.
- Попередньо підготовлені моделі. Keras надає колекцію попередньо підготовлених моделей, які можна використовувати для різних завдань, таких як класифікація зображень і обробка природної мови.
- Інтеграція з TensorFlow. Keras можна використовувати як високорівневий API поверх TensorFlow, виграючи від його обчислювальної потужності та екосистеми.

2.2. Опис програми за допомогою бібліотеки Keras

Набори позначених даних (датасети)

У Keras існує багато датасетів, які можна імпортувати та завантажувати. До прикладу, такі датасети як CIFAR-10 та MNIST.

Шари та параметри

Keras містить численні шари та параметри, такі як функції втрат, оптимізатори, метрики оцінки. Вони використовуються для створення, налаштування, тренування та оцінки нейронних мереж. Дана бібліотека також підтримує одно- та двомірні згорткові та рекурентні нейронні мережі

Методи попередньої обробки даних

В Keras у наявності багато інструментів для обробки даних, до прикладу, Keras.np_utils.to_categorical() для унітарного кодування y_train та y_test.

Метод add()

Щоб імпортувати шари нейронної мережі достатньо скористатися методом add() та прописати певні параметри.

Метод compile()

Перед початком навчання потрібно налаштувати його процес, що і робиться за допомогою даного методу, приклад:

model.compile(loss='categorical_crossentropy',optimizer=Adam(), metrics=['accuracy'])

Метод fit()

За допомогою даного метода відбувається тренування моделі. Приклад: model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5, batch_size=200) У параметрах вказується кількість епох та розмір датасета.

Оцінка моделі

Після тренування моделі результати перевіряються на нових даних, або за допомогою predict_classes() або evaluate().

Опис програми:

Імпорт потрібних модулів

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
%matplotlib inline
from keras.models import Sequential
from keras.layers import Dense,Dropout,Flatten,Conv1D,MaxPooling1D
from keras import utils
from keras.datasets import fashion_mnist
from keras.utils import np_utils
import array as arr
```

Імпорт та підключення датасета. Далі крок попередньої обробки даних. Так як дана нейромережа буда реалізована для розпізнавання двомірних та вище даних, потрібно зменшити розмірність даних, замість параметрів (28, 28, 1) треба прописати (784, 1), де 784 – результат множення 28*28, а 1 -кількість каналів, які необхідно згорнути за допомогою фільтрів. Перевод правильних відповідей у представлення one-hot за допомогою np_utils.to_categorical.

```
(x_train,y_train),(x_test,y_test) = fashion_mnist.load_data()

x_train3D = x_train.reshape(x_train.shape[0],784,1).astype('float32')
x_test3D = x_test.reshape(x_test.shape[0],784,1).astype('float32')

x_train3D_normalize = x_train3D / 255
x_test3D_normalize = x_test3D / 255

y_trainOneHot = np_utils.to_categorical(y_train)
y_testOneHot = np_utils.to_categorical(y_test)
```

Далі ініціалізуємо Sequential модель та додаємо потрібні нам шари.

```
model = Sequential()
model.add(Conv1D(filters=16,
                 kernel_size=25,
                 padding = 'same',
                 input_shape = (784,1),
                 activation = 'relu'))
model.add(MaxPooling1D(pool_size = 2, strides=1))

model.add(Conv1D(filters=10,
                 kernel_size=25,
                 padding = 'same',
                 activation = 'relu'))

model.add(MaxPooling1D(pool_size = 2, strides=1))

model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128,activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy',
                    optimizer=keras.optimizers.Adam(learning_rate=(train_speed/1000) ),
                    metrics=['accuracy'])
```

Дана згорткова нейромережа має наступну архітектуру:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 784, 16)           416

 max_pooling1d (MaxPooling1D  (None, 783, 16)          0
 )

 conv1d_1 (Conv1D)           (None, 783, 10)           4010

 max_pooling1d_1 (MaxPooling  (None, 782, 10)          0
 1D)

 dropout (Dropout)           (None, 782, 10)           0

 flatten (Flatten)           (None, 7820)              0

 dense (Dense)               (None, 128)               1001088

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
```

Наступним кроком буде навчання данної нейромережі, перевірка результатів за допомогою методу model.evaluate() та збереження архітектури з вагами в окремі файли.

```
train_history=model.fit(x=x_train3D_normalize,
                        y=y_trainOneHot,validation_split=0.3,
                        epochs=epoch_number, batch_size=batch_size, verbose=2)

scores = model.evaluate(x_test3D_normalize,y_testOneHot)

print("\nTest data accuracy: %.2f%%" % (scores[1]*100))

model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
model.save_weights("model.h5")
```

2.3. Функція відображення

Відображення - це функція, яка показує залежність наступних значень параметрів системи від попередніх значень.

Метою цього завдання є створення алгоритму автоматичного підбору швидкості навчання для нейронних мереж. Для досягнення цієї мети було використано Python та бібліотеку Keras для розробки програми, яка працює зі згортковою нейронною мережею. Ця програма дозволяє встановлювати та змінювати параметри швидкості навчання.

Ми розглядатимемо згорткову нейронну мережу як детерміновану систему, для якої було проведено дослідження діаграми розгалуження з використанням функції відображення виду:

xn+1=η-xn-xn2
де n — крок, η = альфа — параметр, який визначає швидкість навчання.

ЇЇ нерухомі точки:
x 1,2 = −1±(η +1)1/2,
власні значення, які можна обчислити наступним чином:
ρ1 =1− 2 (η +1)1/2

Розглянемо вплив швидкості навчання на функцію похибки даної згорткової нейронної мережі. Важливу роль в процесі навчання та розпізнавання згорткових нейромереж відіграють оптимізаційні методи. В даній роботі використовувалися такі оптимізатори Adam, SGD та AdaGrad.

Результати для оптимізатора SGD:

10 епох

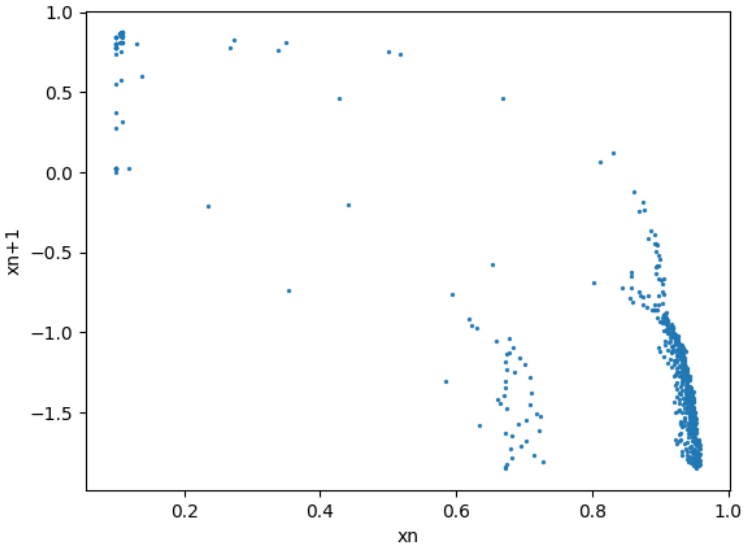

Рис. 2.1 – Діаграма логістичної функції

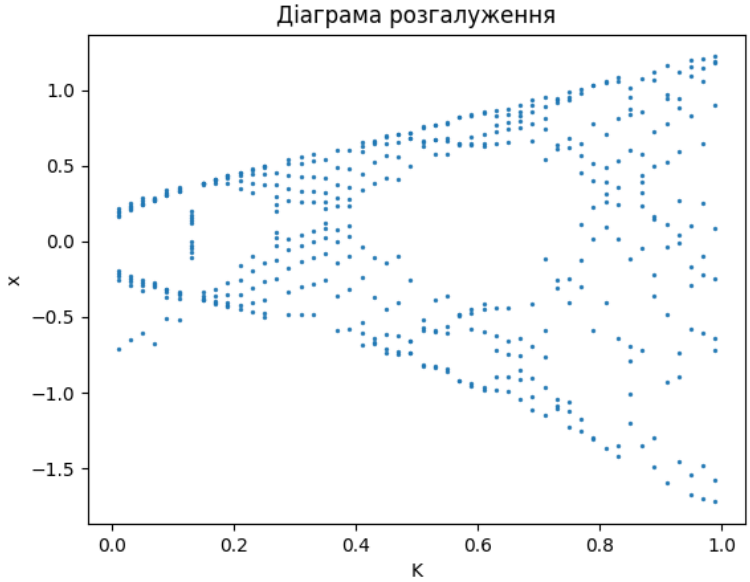

Рис 2.2 – Діаграма розгалуження

Результати для оптимізатора AdaGrad:

10 епох

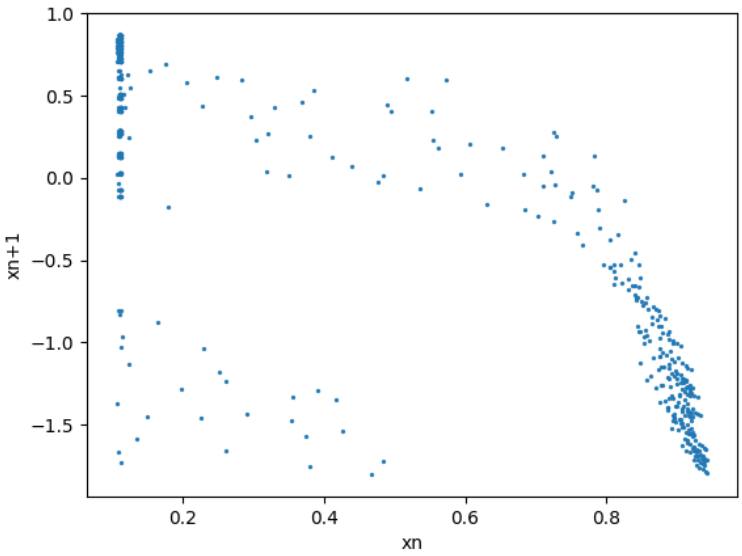

Рис. 2.3 - Діаграма логістичної функції

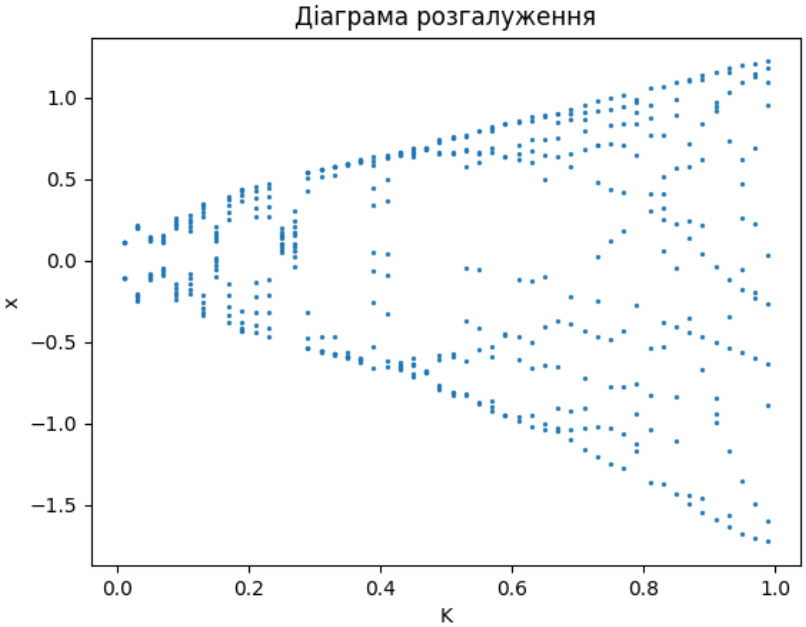

Рис. 2.4 – Діаграма розгалуження

Результати для оптимізатора Adam:

10 епох

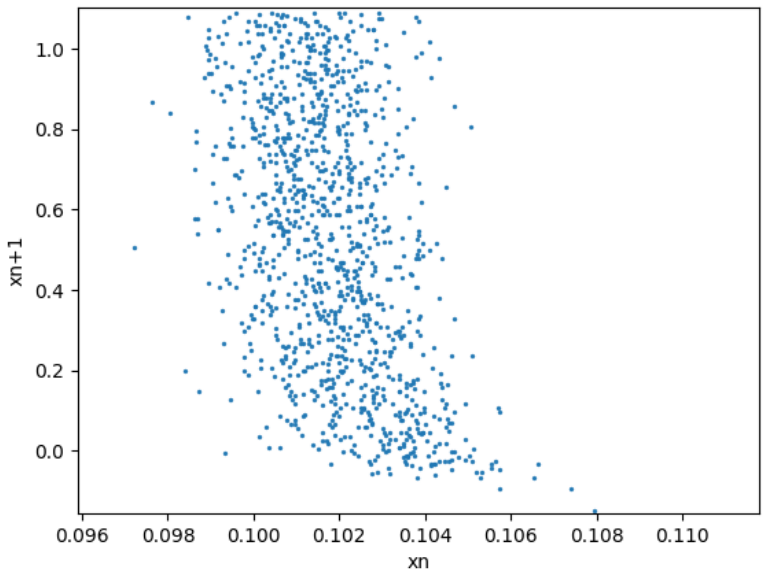

Рис. 2.5 – Діаграма логістичної функції

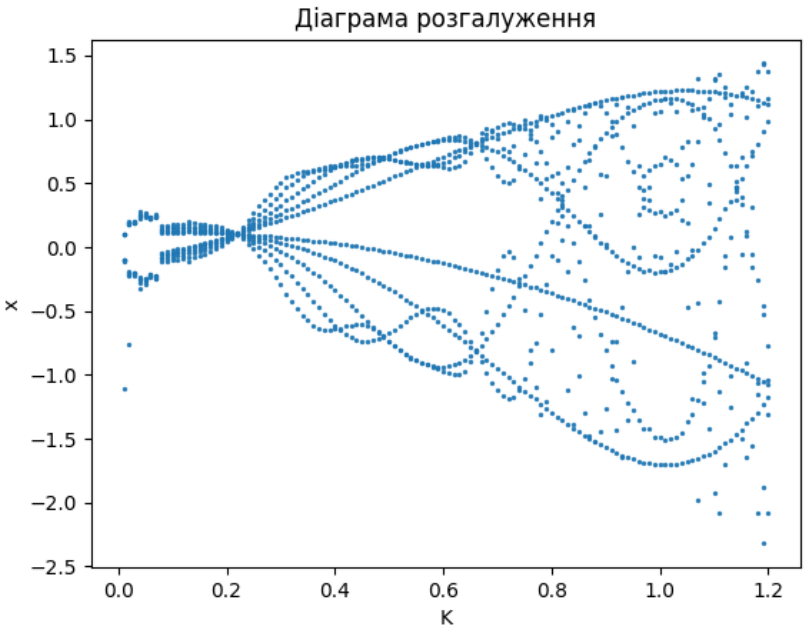

Рис. 2.6 – Діаграма розгалуження

100 епох

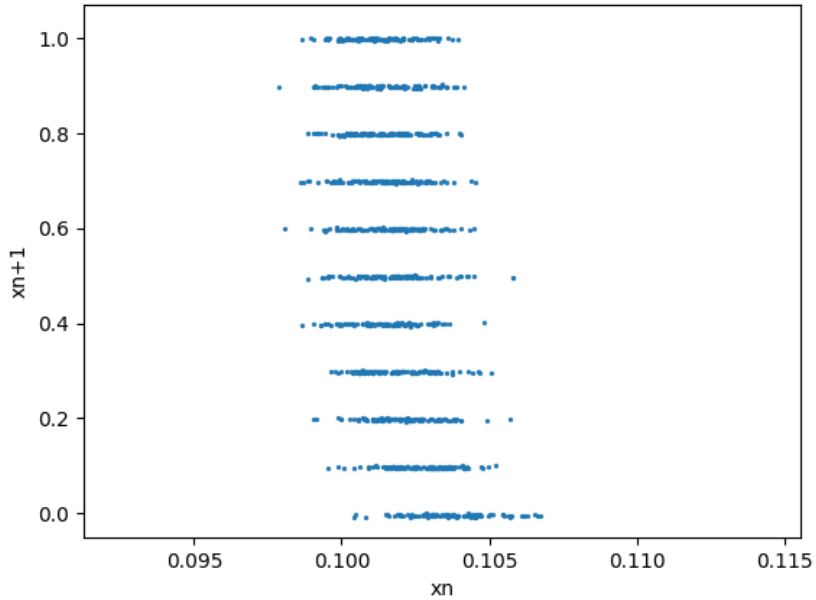

Рис. 2.7 – Діаграма логістичної функції

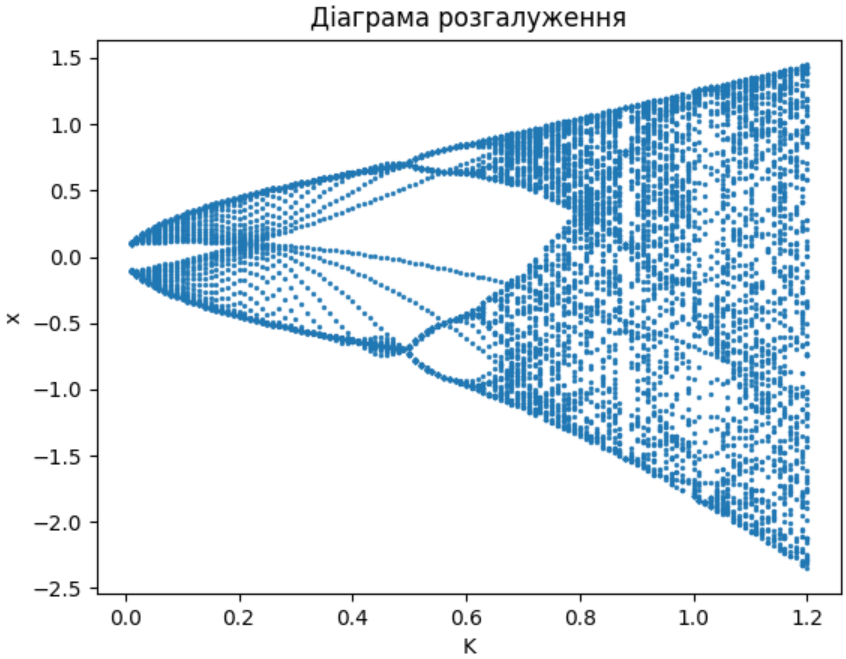

Рис 2.8 – Діаграма розгалуження

Наразі найбільш дослідженим є оптимізатор Adam. Для всіх трьох оптимізаторів 10 епох явно недостатньо для задовільного процесу навчання та розпізнавання. Дані вище діаграми розгалуження засвідчують наявність процесів подвоєння локальних мінімумів. Така поведінка діаграм розгалуження свідчить про наявність хаотичних станів в даній системі.

Для оптимізатора Adam процес подвоєння локальних мінімумів з'являється в околі альфа ≈ 0,6

Для оптимізатора AdaGrad процес подвоєння локальних мінімумів з'являється в околі альфа ≈ 0,8

Для оптимізатора SGD процес подвоєння локальних мінімумів з'являється в околі альфа ≈ 0,4 або альфа ≈ 0,5.

Також, виходячи з діаграм розгалуження для 10 епох наведених вище, для оптимізаторів Adam та SGD в околі альфа ≈ 0,1 простежується аномальна поведінка функції похибки від швидкості навчання.

Як було вже вище зазначено, входження системи в непередбачуваний режим описується каскадом біфуркацій, які йдуть одна за одною. Каскад біфуркацій веде послідовно до появи вибору між двома рішеннями, потім чотирма і так далі, система починає коливатися в хаотичному, турбулентному режимі послідовного подвоєння кількості можливих значень

Висновки

Було детально проаналізовано згорткову нейромережу, яка була реалізована за допомогою модуля Keras на мові програмування Python.

Було реалізовано функцію відображення, за допомогою якої було отримано залежність функції похибки від параметрів загорткової нейронної мережі разом з діаграмами логістичної функції та діаграмами розгалуження.

Виходячи з діаграм розгалуження, можна зробити висновок, що при використанні усіх трьох оптимізаторів (SGD, Adam, AdaGrad) у розглянутій згортковій нейронній мережі можуть виникати хаотичні стани, які зумовлені збільшенням швидкості навчання та характеризуються подвоєнням локальних мінімумів при підході до глобального. Для отримання більш детальної інформації щодо процесу виникнення хаотичних станів при використанні оптимізаторів SGD та AdaGrad слід продовжити роботу, тобто провести навчання зі значенням кількості епох = 100.

Виходячи з отриманих даних, можна зробити висновок, що найкраще себе показав оптимізатор AdaGrad. При його використанні подвоєння локальних мінімумів з'являється в околі альфа ≈ 0,8. Даний позитивний, відносно інших оптимізаторів результат, може пояснюватися адаптивністю швидкості навчання цього оптимізатора, тобто даний оптимізаційний метод адаптує швидкість навчання для кожного параметра на основі історії градієнтів цього параметру.

Також, слід зазначити появу аномальної поведінки функції похибки від швидкості навчання околі альфа ≈ 0,1 при використанні оптимізаторів Adam та SGD.

Список використаних джерел

1. Могильний С. Б. Машинне навчання з використанням мікрокомп’ютерів: навч.-метод. посіб. / за ред. О. В. Лісового та ін. – К., 2019. – 226 с.
2. Браунли Дж. - Прогнозування на основі нейронної мережі моделі довгої короткострокової пам’яті на Python (Jason Brownlee - Long Short-Term Memory Networks With Python, 2017)
3. Равичандиран С. – Глибоке навчання з підкріпленням на Python (Sudharsan Ravichandiran - Hands-On Reinforcement Learning with Python, 2018)
4. Гудфеллоу Я., Бенджио И., Курвилль А. - Глубокое обучение (Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning, 2017)
5. Траск Э. – «Грокаємо» глибоке навчання (Andrew W. Trask - Grokking Deep Learning, 2019)
6. Руденко, О. Г. Штучні нейронні мережі / О. Г. Руденко, Є. В. Бодянський. – Харків : Компанія СМІТ, 2006. – 404 с.
7. Субботін, С. О. Нейронні мережі : навчальний посібник / С. О. Субботін, А. О. Олійник ; під заг. ред. проф. С. О. Субботіна. – Запоріжжя : ЗНТУ, 2014. – 132 с
8. Yurij Olenych, Sergiy Sveleba, Ivan Katerynchuk,Ivan Kunio, Ivan Karpa “Features of deep studyneural network” https://openreviewhub.org/lea /paper2019/features-deep-study-neural-network

Додатки

Нижче представлений код функції відображення

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
import array as arr
import numpy as np
import matplotlib.pyplot as plt
import keras.optimizers

# Створення масивів
cz=arr.array('d',[])
yy=arr.array('d',[])
x=arr.array('d',[])
amplid=arr.array('d',[])
omega=arr.array('d',[])
chastota=arr.array('d',[])
xx=arr.array('d',[])
xxx=arr.array('d',[])
chastota_b=arr.array('d',[])
yy=arr.array('d',[])

# завантажуємо дані MNIST
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# перетворюємо кожну картинку в двовимірний масив
batch_size, img_rows, img_cols = 64, 28, 28
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

# приводимо дані до типу float32 і нормалізуємо їх від 0 до 1
X_train = X_train.astype("float32")
X_test = X_test.astype("float32")
X_train /= 255
X_test /= 255

# переводимо правильні відповіді в one-hot уявлення
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

```
# створюємо модель, задаємо зміну масиву—швидкості або параметра Dropout
for c1 in np.arange(0.01,0.995,0.02):
    model = Sequential()

    # додаємо згорткові шари
    model.add(Activation("relu"))
    model.add(Convolution1D(32,10, 10, padding='same', input_shape=input_shape))
    model.add(MaxPooling1D(pool_size=(3, 3), strides=(3, 3), padding='same'))
    model.add(Convolution1D(64, 10,10, padding='same', input_shape=input_shape))
    model.add(Activation("relu"))
    model.add(MaxPooling1D(pool_size=(2, 2), strides=(2, 2), padding='same'))

    # додаємо повнозв'язні шари
    model.add(Flatten())
    model.add(Dense(128))
    model.add(Activation("relu"))
    model.add(Dropout(c1))
    model.add(Dense(10))
    model.add(Activation("softmax"))

    # компілюємо і запускаємо навчання
    model.compile(loss='categorical_crossentropy',
            optimizer=keras.optimizers.Adam(learning_rate=(0.6) ),metrics=['accuracy'])
    #model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])

    train_history=model.fit(X_train, Y_train, batch_size=batch_size, epochs=100,
verbose=1, validation_data=(X_test, Y_test))
    score = model.evaluate(X_test, Y_test, verbose=1)

    # виводимо результати
    #створюємо функцію дя виведення результатів
    print(score)
    data0=arr.array('d',[])
    def show_train_history(train_history,train,validation):
        return train_history.history[train]
    #print(type(train_history.history[train]))
    A=show_train_history(train_history,'accuracy', 'val_acc')
    #A=show_train_history(train_history,'loss','val_loss')
    n=len(A)
    print(len(A),c1)
    # Створюємо масив значень похибки при кожній ітерації
    data0=[]
```

```
    for i in np.arange(n):
        data0.append(A[i])
    data0=np.array(data0,float)
    print('data0=',data0, len(data0))

    #print(data0)
  # Розрахунок та створення масиву дя побудови логістичної функції
    for i in range(n):
        chastota.append(c1-data0[i-1]-data0[i-1]*data0[i-1])
        x.append(data0[i])
        xx.append(c1)
    #print(len(chastota))
# Розрахунок діаграми логістичної функції
    chast=[]
    data00=[]
    for i in range(n):
        chast.append(c1-data0[i-1]-data0[i-1]*data0[i-1])
        data00.append(data0[i])
    data=(min(chast)+max(chast))/2
    k=len(chast)
    #print('data',data,'chast',chast,'c1',c1)

# Створення масиву для діаграми розгалуження
    for data0 in data00:
        data0=data
        data0=c1-data0-data0*data0
        chastota_b.append(data0)
        xxx.append(c1)
        #print('data0',data0,c1)
        data=data0
# Побудова логістичної функції та діаграми розгалуження
chastota=np.array(chastota)
chastota_b=np.array(chastota_b)
x=np.array(x)
xx=np.array(xx)
xxx=np.array(xxx)
print(len(chastota_b),len(xxx))
plt.scatter(x, chastota, s=2, alpha=0.9)
#plt.plot(xx, chastota,'r.')
plt.xlabel(u"xn")
plt.ylabel(u"xn+1")
plt.show()

plt.scatter(xxx, chastota_b, s=2, alpha=0.9)
plt.title("Діаграма розгалуження")
```

```
plt.xlabel(u"K")
plt.ylabel(u"x")
plt.show()
```