МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

КУРСОВА РОБОТА

з дисципліни

“МАШИННЕ НАВЧАННЯ”

на тему:

Проблема вибуху градієнта і евристика gradient clipping.

Студента 309 групи
спеціальності 122 “Комп’ютерні науки”
Скворцова Іллі Володимировича
Керівник
к. е. н., доц. Бойко Н.І.

Кількість балів:_____Оцінка _____

Члени комісії

_____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

_____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

_____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Львів – 2020

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ І СКОРОЧЕНЬ

MLP - багатошаровий перцептрон;
NaN - Не число;
SGD - стохастичний градієнтний спуск;
LSTM - довго-короткострокова пам’ять

ВСТУП

В даній курсовій роботі буде розглянута одна з проблем, що з’являється знову і знову, під час процесу тренування штучних нейронних мереж. Це проблема нестабільних градієнтів, яка також є відомою під назвами вибухаючих та зникаючих градієнтів. Оскільки ці проблеми виникають за схожих обставин, то обидві ці проблеми будуть розглянуті в даній роботі.

АНАЛІЗ МАТЕРІАЛІВ ТА МЕТОДІВ

Актуальність

Вибух градієнтів може спричинити проблеми при тренуванні штучних нейронних мереж. Якщо виникають вибухові градієнти може виникнути нестабільна мережа і неможливо буде завершити навчання . Значення ваг також можуть стати надто великими, що призведе переповнення і до того, що називається значенням Not A Number(NaN), що перекладається як не число - що значить, що дані значення не є визначеними. Корисно знати, як визначити вибухові градієнти, щоб виправити тренування.

З іншої сторони проблема зникаючих градієнтів може призвести до того, що значення ваг мережі будуть змінюватися на надзвичайно малі значення, що може призвести як до уповільнення процесу навчання мережі, так і до того, що вона може “застрягти” я певному значенні, що фактично значить те, що мережа може застрягнути в одному етапі навчання.

Мета

Мета даної роботи полягає у розгляді феноменів вибухаючих та зникаючих градієнтів, розгляді їх проблематики, причин появи та методів їх вирішення.

Розділ 1. Градієнтний спуск

Градієнтний спуск в одному вимірі

Для розуміння феноменів градієнтного вибуху та зникнення, слід для початку розглянути, що таке градієнтний спуск.

Для початку, розглянемо випадок, в якому в нас є один параметр. Також припустимо, Що наша цільова функція присвоює кожному значенню цього параметра якесь певне значення. Формально, можна сказати, що ця цільова функція має вигляд $f : \mathrm{R} \to \mathrm{R}$.

Зауважимо, що область f є одновимірною. Відповідно до його розширення серії Тейлора, як показано у вступній главі, маємо

$$f(x+\epsilon) \approx f(x) + f'(x)\epsilon.$$

Замінаючи ϵ на $-\eta f'(x)$, де η є сталою, маємо

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)2.$$

Якщо η встановлено як мале додатне значення, отримаємо

$$f(x - \eta f'(x)) \leq f(x).$$

Іншими словами, оновлення x як

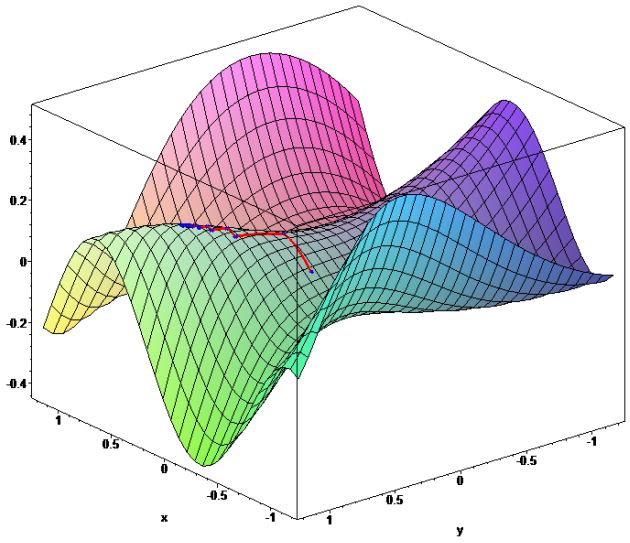
$$x := x - \eta f'(x)$$

може зменшити значення f(x), якщо його поточне значення похідної f '(x) ≠ 0. Оскільки похідна f '(x) є особливим випадком градієнта в одновимірній області, то вищенаведене оновлення x - це градієнтний спуск в одновимірній області.

Позитивний скаляр η називається швидкістю навчання або розміром кроку. Варто зауважити, що більша швидкість навчання збільшує шанс оминути глобальний мінімум та може призвести до осцелювання. Однак якщо рівень навчання занадто малий, то зближення може бути

дуже повільним. На практиці зазвичай підбирають належну швидкість навчання за допомогою експериментів.

Градієнтний спуск з багатовимірними параметрами



Reference: https://en.wikipedia.org/wiki/Gradient_descent

Розглянемо цільову функцію f: Rd → R, яка приймає будь-який багатовимірний вектор x = [x1, x2, ..., xd] ⊤ як свій вхід. Градієнт f (x) по відношенню до x визначається вектором часткових похідних:

$$\nabla \mathbf{x} f(\mathbf{x}) = [\partial f(\mathbf{x}) \partial x1, \partial f(\mathbf{x}) \partial x2, \ldots, \partial f(\mathbf{x}) \partial xd]^\top.$$

Щоб зберегти наше позначення компактним, ми можемо використовувати позначення ∇f (x) та ∇xf (x) взаємозамінно, коли немає двозначності щодо параметрів, під якими ми оптимізуємо. Простіше кажучи, кожен елемент градієнта ∂f (x) / ∂xi вказує на швидкість зміни для f у точці x по відношенню лише до входу xi. Для вимірювання швидкості зміни f у будь-якому напрямку, який представлений одиничним вектором u, в багатовимірному обчисленні визначаємо похідну спрямованості f при x у напрямку u як

$$D\mathbf{u} f(\mathbf{x}) = \lim h \to 0 f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x}) h,$$

які можна переписати відповідно до правила ланцюжка як

$$D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

Оскільки Duf (x) дає швидкості зміни f у точці x у всіх можливих напрямках, щоб мінімізувати f, нам цікаво знайти напрямок, де f може бути зменшено найшвидше. Таким чином, ми можемо мінімізувати направлену похідну Duf (x) відносно u. Оскільки Duf (x) = // ∇f (x) // · // u // · cos (θ) = // ∇f (x) // · cos (θ), де θ - кут між ∇f (x) та u, мінімальне значення cos (θ) дорівнює -1, коли θ = π. Тому Duf (x) мінімізується, коли u знаходиться в протилежному напрямку градієнта ∇f (x). Тепер ми можемо ітеративно зменшити значення f за допомогою наступного оновлення спуску градієнта:

$$\mathbf{x} := \mathbf{x} - \eta \nabla f(\mathbf{x}),$$

де позитивний скаляр η називається швидкістю навчання або розміром кроку.

Стохастичний градієнтний спуск

Однак алгоритм спуску градієнта може виявитися нездійсненним, коли розмір даних тренувань є надто великим. Таким чином, натомість часто використовується стохастична варіація алгоритму.

Варто звернути увагу на те, що при тренуванні моделей глибокого навчання часто розглядається цільова функція, як сума кінцевої кількості функцій:

$$f(\mathbf{x})=1n\sum i=1nfi(\mathbf{x}),$$

де fi (x) - функція втрат на основі екземплярі тренувальних даних, індексованих через i. Важливо підкреслити, що обчислювальна вартість за ітерацію в градієнтному спуску лінійно залежить від розміру набору навчальних даних n. Отже, коли n є надто великим, обчислювальна вартість градієнтного спуску за ітерацію є дуже високою.

З огляду на це, стохастичний градієнтний спуск пропонує кращу альтернативу. При кожній ітерації, радше, ніж обчислювати градієнт ∇f (x), стохастичний градієнт спуск випадковим чином обирає зразки i та обчислює ∇fi (x). Суть полягає в тому, що стохастичний градієнтний спуск використовує ∇fi (x) як неупереджу оцінку ∇f (x), оскільки

$$Ei\nabla fi(\mathbf{x})=1n\sum i=1n\nabla fi(\mathbf{x})=\nabla f(\mathbf{x}).$$

В узагальненому випадку, при кожній ітерації міні-вибірки B, що складається з індексів для екземплярів тренувальних даних, може бути відібрана в одній формі з заміною. Аналогічно ми можемо використовувати

$$\nabla fB(\mathbf{x})=1|B|\sum i\in B\nabla fi(\mathbf{x})$$

Що оновити **x** як

$$\mathbf{x} := \mathbf{x} - \eta \nabla f_B(\mathbf{x}),$$

де | B | позначає кардинальність міні-вибірки, а позитивний скаляр η - швидкість навчання або розмір кроку. Аналогічно,стохастичний градієнт міні-вибірки $\nabla f_B(x)$ є неупередженою оцінкою для градієнта $\nabla f(x)$:

$$E_B \nabla f_B(\mathbf{x}) = \nabla f(\mathbf{x}).$$

Цей узагальнений стохастичний алгоритм також називають стохастичним градієнтим спуску міні-вибірки, і, зазвичай, просто зветься, як стохастичний градієнтний спуск. Обчислювальна вартість за повторення становить

$$O(|B|).$$

Таким чином, коли розмір міні-вибірки є невеликим, обчислювальна вартість при кожній ітерації також є невеликою.

Є й інші практичні причини, які можуть зробити стохастичний градієнтний спуск більш привабливим, ніж градієнтний спуск. Якщо набір навчальних даних має багато зайвих екземплярів даних, стохастичні градієнти може бути настільки близькими до справжнього градієнта $\nabla f(x)$, що невелика кількість ітерацій знайде корисні рішення проблеми оптимізації. Насправді, коли набір даних про тренування є досить великим, то стохастичному градієнтному спуску потрібна лише невелика кількість ітерацій, щоб знайти корисні рішення,
і щоб загальна обчислювальна вартість була нижчою, ніж для градієнтного спуску навіть лише після однієї ітерації. Крім того, стохастичний градієнтний спуск може розглядатися як такий, що надає ефект регуляризації, особливо коли розмір міні-вибірки є невеликим через випадковість і шум у відборі міні-вибірки.

Розділ 2. Зникаючі та вибухові градієнти

Зникаючі градієнти

Проблема зникаючого градієнта - це проблема, яка викликає великі труднощі при навчанні нейронної мережі. Більш конкретно, це проблема, яка стосується ваг в попередніх шарах мережі. Під час тренування, стохастичний градієнтний спуск (або SGD) працює для обчислення градієнта втрат відносно ваг у мережі. Так, іноді, градієнт ваг у перших шарах мережі може стати надзвичайно малим.

Коли SGD обчислює градієнт відносно певної ваги, він використовує це значення для оновлення цієї ваги, і вага оновлюється деяким способом, пропорційним градієнту. Якщо градієнт є надзвичайно малим, то це оновлення, в свою чергу, буде також надзвичайно малим.

Отже, якщо це нещодавно оновлене значення ваги ледве перемістилося від його початкового значення, то це не дуже добре для мережі. Ця зміна погано вплине на мережу та ніяк не допоможе зменшити втрати, оскільки вона ледве змістилася з того місця, де була до оновлення.

Як результат, ця вага наче застрягає, ніколи насправді не оновлюється достатньо, щоб навіть наблизитися до оптимального значення, що несе негативні наслідки для решти мережі праворуч від цієї ваги та погіршує здатність мережі навчатися.

Отже як виникає ця проблема? Відомо, що градієнт втрат по відношенню до будь-якої ваги є результатом певних похідних, які залежать від компонент, які в перебувають далі у мережі.

З огляду на це, можна зробити висновок, що чим раніше в мережі знаходиться вага, тим більше значень знадобиться в добутку, про який тільки що було згадано, щоб отримати градієнт втрат відносно цієї ваги.

Очевидно, що якщо монжники цього добутку або хоча б деякі з них є невеликими значеннями, то результатом їх множення буде ще менше число.

Як ми вже згадували раніше, тепер береться цей результат і використовується для оновлення певної ваги. Також варто зауважити, що це оновлення, спочатку множиться на на нашу швидкість навчання, що саме по собі є невеликим числом, зазвичай коливаючись між .01 і .0001.Таким чином, результатом цього добутку буде ще менше число. Після отримання цього числа ми віднімаємо його від ваги, і кінцевим результатом цієї різниці буде значення оновленої ваги.

Застрягання ваг

Якщо градієнт, який було отримаємо відносно цієї ваги, вже є досить малим, тоді після помноження його на швидкість навчання, їх добуток стане ще меншим, і після віднімання цього числа від ваги, вона практично не зміниться.

Таким чином, вага може фактично застрягнути на одному місці. Тоді вона не буде зміщуватися та не буде “навчатися”, і тому ніяк не допомагатиме досягти загальної мети, а саме мінімізації втрат мережі.

Так, перші ваги в мережі є найбільш вразливими до цієї проблемі. Тому що, як було вище згадано, чим раніше у мережі знаходиться вага, тим більше множників буде включено в добуток для обчислення градієнта.Чим більше множників ми множимо разом на число менше, ніж один, тим швидше градієнт зникне.

Вибух градієнта

Тепер буде розглянута дана проблема у зворотному напрямку. А саме, не як градієнт, який зникає, а скоріше, градієнт, що вибухає.

Візьмемо як приклад, проблему зникаючого градієнта, з вагами на початку мережі, що виникає через добуток, принаймні деяких, порівняно невеликих значень.

Тепер розглянемо обчислення градієнта відносно тієї самої ваги, але, замість малих множників, будуть взяті досить великі. Під “досить великими” маються на увазі множники, що є більшими за одиницю.

Очевидно, що якщо перемножити декілька множників, кожен з яких є більшим, від одиниці, то ми отримаємо число більше, ніж один, а можливо, навіть набагато більше, ніж один.

Тут тримається той самий аргумент, що про який було згадано, під час розглядання зникаючих градієнтів, де, чим раніше в мережі існує вага, тим більше множників буде використано в добутку, про який тільки що було згадували.

Як результат, можна побачити, що чим більше цих великих членів ми множимо разом, тим більший буде градієнт, таким чином, фактично, вибухаючи за розміром.

З даним градієнтом буде пройдено такий самий процес, як із градієнтом з попереднього розділу, ним буде оновлена відповідна вагу, про яку було згадано раніше.

Однак цього разу, замість того, щоб ледь перемістити свою вагу за допомогою цього оновлення, вага буде сильно зміщуватись, настільки, що оптимального значення для цієї ваги може не буде досягнуто, оскільки відношення, до якої вагу оновлюється кожної епохи є просто

занадто великою і продовжує рухатися все далі і далі від оптимального значення.

Вирішення проблеми вибухаючих та зникаючих градієнтів

Є декілька методів, які дозволяють вирішити проблему вибухових градієнтів. Однак в даній роботі буде розглянутий лише метод відсікання градієнта.

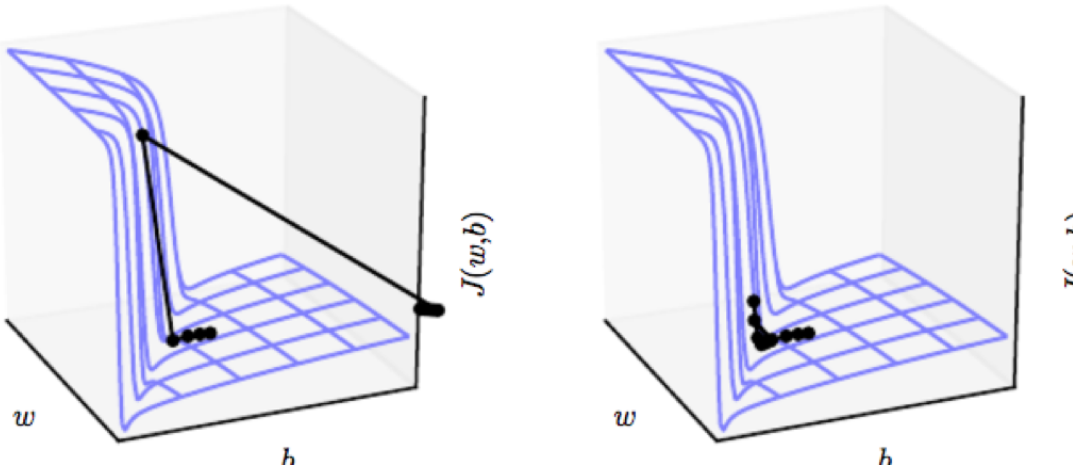
Відсікання Градієнта - це техніка, яка справляється із вибуховими градієнтами. Ідея відсікання градієнта дуже проста: якщо градієнт стає занадто великим, то його масштаб підлягає зміні таким чином, щоб він став меншим. Точніше, якщо $\|g\| \geq c$, то

$$\mathbf{g} \leftarrow c \cdot \mathbf{g} / \|\mathbf{g}\|$$

де c - гіпер параметр, $\mathbf{g}$ - градієнт, а $\|\mathbf{g}\|$ - норма $\mathbf{g}$. Оскільки $\mathbf{g}/\|\mathbf{g}\|$ - одиничний вектор, то після масштабування новий $\mathbf{g}$ матиме норму c. Варто зауважити, що якщо $\|\mathbf{g}\| < c$, то нічого не потрібно робити.

Градієнтне відсікання забезпечує, що градієнтний вектор $\mathbf{g}$ має норму не більшу, ніж c. Це допомагає градієнтному спуску мати більш відповідну поведінку, навіть якщо поле втрат моделі є нерегулярним. На наступному малюнку показаний приклад надзвичайно крутої скелі на полі втрат. Без обрізання параметри роблять величезний крок спуску і покидають "хороший" регіон. За допомогою обрізання розмір кроку спуску обмежується, а параметри залишаються у «хорошому» регіоні.

Без обрізання З обрізанням



Reference: Ian Goodfellow et. al, “Deep Learning”, MIT press, 2016

З іншої сторони, основною проблема зникаючих градієнтів полягає не в тому, що вони наближаються до нуля зі збільшенням розмірів / розгортання шарів, а у відносних розмірах градієнтів між мережевими шарами.
Щоб вирішити цю проблему з використанням відсікання потрібні будуть окремі правила / параметри для кожного шару. Однак це потребує ретельної настройки параметрів. Більш надійним способом досягти цього буде використання спеціальних архітектур.
Однією з таких архітектур є Багатошаровий персептрон(Multilayer Perceptron(MLP)) з k прихованими шарами.
У такому випадку, верхні шари, що знаходяться близько до виходу, будуть мати більші градієнти, ніж нижні шари, що є ближче до входу, якщо використовується насичуюча нелінійність, як tanh. Таким чином, глобальна швидкість навчання для всіх ваг завжди віддаватиме перевагу найвищим параметрам.
Іншою валідною архітектурою є LSTM. Обидві архітектури будуть розглянуті нижче.

ЕКСПЕРИМЕНТИ

Багатошаровий персептрон(MLP) з вибухаючим та зникаючим градієнтами

Для того, щоб продемонструвати вплив вибухаючого градієнта на процес навчання нейронної мережі та як обрізання градієнта може покращити навчання мережі, буде використано Багатошаровий персептрон(MLP) для проблеми регресії.

Модель буде продемонстрована на необроблених даних без будь-якого масштабування вхідних чи вихідних змінних. Це хороший приклад для демонстрації вибухових градієнтів, оскільки модель, навчена прогнозувати немасштабіровану цільову змінну, призведе до помилкових градієнтів зі значеннями в межах сотень чи навіть тисяч, залежно від розміру вибірки, що використовується під час тренування. Такі великі градієнтні значення, ймовірно, можуть призвести до нестабільного навчання або переповнення значень ваги.

Після запуску прикладу, будується модель та обчислюється середня квадратична помилка в тренувальних та тестових вибірках.

Однак далі модель не в змозі коректно навчатися, що призводить до передбачень значень NaN(Not a Number). Ваги моделі вибухнули під час тренувань, що було спричинено дуже великими розмірами помилок і я наслідок градієнтів, обчислених для оновлення ваг.

```
Train: nan, Test: nan
```

Це свідчить про необхідність певного втручання стосовно цільової змінної для вивчення цієї проблеми.

Лінійний графік історії підготовки був успішно створений, однак він нічого не показує, оскільки модель майже одразу призводить до середньоквадратичної помилки NaN.

Традиційним рішенням було б змінити масштаб цільової змінної, використовуючи або стандартизацію, або нормалізацію, і такий підхід рекомендується для MLP. Тим не менш, альтернативою, яка буде використана в цьому випадку - градієнтне відсікання.

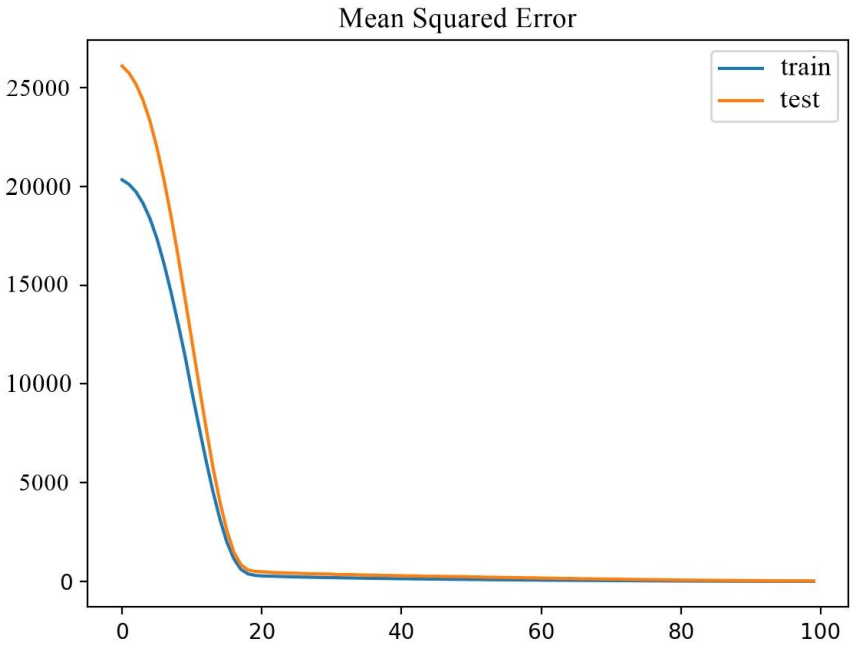
Далі, до попередньої моделі, буде додано масштабування градієнтної норми.

Після запуску прикладу, будується модель та обчислюється середня квадратична помилка в тренувальних та тестових вибірках.

У цьому випадку ми можемо побачити, що масштабування градієнта з векторною нормою 1,0 призвело до стабільної моделі, здатної вивчити задачу та сходитись на розв’язанні.

Створюється також графік ліній, що показує середньоквадратичні втрати помилок у тренувальному та тестовому наборах даних протягом навчальних епох.

Графік показує, як втрати знижувались від великих значень понад (20 000) до малих значень нижче (100) швидко протягом 20 епох.

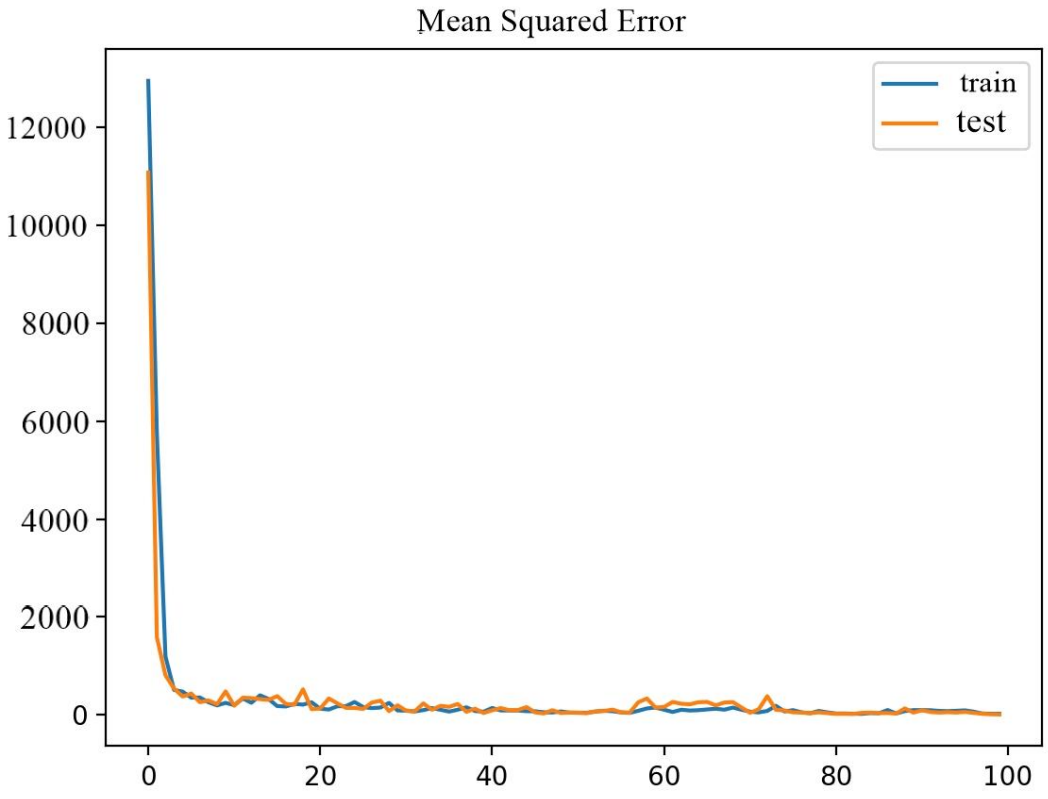


Різниця між першим та другим версіями мережі є надзвичайними, адже в другій версій було не просто успішно пройшов процес навчання нейронної мережі, а й було досягнуто досить непоганих результатів.

Наостанок, буде оновлено навчання MLP для використання градієнтного відсікання.

Можна бачити, що в цьому випадку модель здатна вивчити проблему без вибуху градієнтів, досягаючи значення середньоквадратичної похибки нижче 10, як під час тренування, так і на тестових наборах.

Створюється також графік ліній, що показує середньоквадратичні втрати помилок у тренувальному та тестовому наборах даних протягом навчальних епох.



Графік показує, Що модель вчиться досить швидко, досягаючи середньоквадратичних втрат помилок, менших, ніж 100, після лише кількох епох.

В даній ітерації мережі, було досягнуто ще більшої швидкості навчання, що дозволяє пришвидшити процес навчання мережі й також досягти кращих результатів.

В даному прикладі навмисно було обрано архітектуру MLP, для того, щоб уникнути зникаючих градієнтів, однак варто зауважити, що під час обрізання встановлюються як верхня, так і нижня межі

Використання різних функцій активації для усунення проблеми зникаючих градієнтів

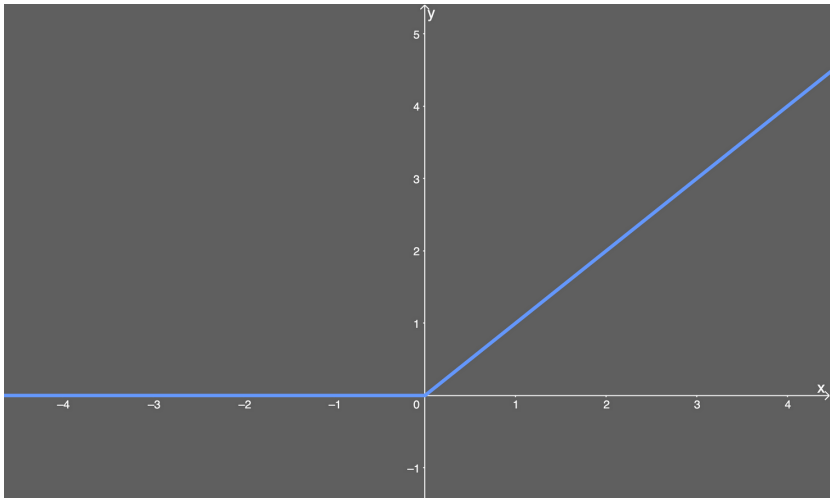
Також використання певних функцій активації таких, “ReLU” чи Leaky ReLU” замість “Sigmoid” чи “ELU” може призвести до покращення рейту навчання та можливо усуне виникнення зникаючих та вибухаючих градієнтів, якщо детальніше:

ReLU

Випрямлена лінійна одиниця може бути одним із способів вирішення проблеми зникнення градієнта, однак може спричинити інші проблеми.
рівняння для ReLU:
ReLU (x) = max (0, x)
Це рівняння говорить про те, що:

1. Якщо input x < 0, встановити input = 0
2. Якщо input > 0, встановіть input, = input

Ось так можна представити ReLU у вигляді графа:



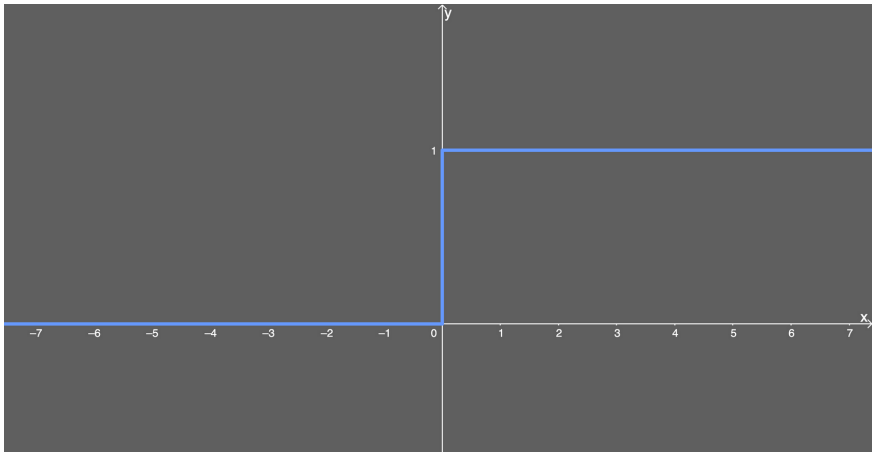
Але як це стосується зникаючої градієнтної проблеми? По-перше, ми маємо диференціальне рівняння:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Це нам говорить про наступне:

1. Якщо вхід x > 0, тоді вхід = 1
2. Якщо вхід <= 0, тоді вхід = 0

Графік:



Так з надзвичайно малі значення просто не можливо отримати при використанні функції активації ReLU, як, наприклад, 0.0000000438 від сигмоподібної функції. Натомість це або 0, або 1.

Однак це може породити ще одну проблему, проблему «мертвих» градієнтів.

Що станеться, якщо занадто багато значень є меншими від 0, при обчисленні градієнтів? Отримується велика кількість ваг, які не оновлюються, оскільки оновлення дорівнює нулю.

Позначимо ReLU як R у цьому рівнянні, де ми просто замінили кожну сигмоїду σ на R.

$$\frac{\partial C}{\partial b1} R'(z1)w2R'(z2)w3R'(z3)w4R'(z4) \frac{\partial C}{\partial a4}$$

Нехай, випадковий вхід z для диференційованого ReLU < 0, тоді функція призведе до того, що ця упередженість 'вмре'. Нехай R'(z3) = 0:

$$\frac{\partial C}{\partial b1} R'(z1)w2R'(z2)w3\ 0\ w4R'(z4) \frac{\partial C}{\partial a4} = 0$$

У свою чергу, тоді вийде, що R ′ (z3) = 0, решта значень множимо, і вийде нуль, через що ця упередженість вмирає. Відомо, що новим

значенням зміщення є зміщення мінус ступінь навчання мінус градієнт, а значить, отримається оновлення в нуль.

Коли вводиться функція ReLU в нейронну мережу, також вводиться велика розрідженість. Що ж насправді означає цей термін розрідженість?

розріджені - невеликі за кількістю, часто розкидані на великій площі. У нейронних мережах це означає, що в матриці активацій присутня велика кількість нулів.

Що можливо отримати за допомогою цієї розрідженості? Коли деякий відсоток (наприклад, 50%) активацій насичується, то таку нейронну мережу можна назвати розрідженою. Це призводить до підвищення ефективності щодо часової та просторової складності - постійні значення (часто) вимагають менше місця і є менш обчислювальними.

Йошуа Бенджо та ін. помітили, що цей компонент ReLU фактично робить нейромережу ефективнішою.

Плюси:

1. Менша часова та просторова складність через нерівномірність, і порівняно з сигмоїдною, вона не розвиває експоненціальні операції, які дорожчі.
2. Уникає зниклої градієнтної проблеми.

Мінуси:

1. Представляється проблема мертвих градієнтів, коли компоненти мережі, швидше за все, ніколи не оновлюються до нового значення. Іноді це може бути і плюсом.
2. ReLU не уникає проблеми з градієнтом, що вибухає.

ELU

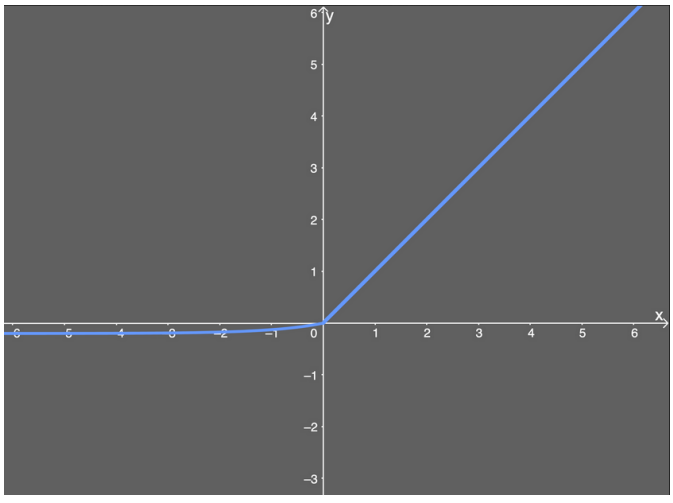
Експоненціальна лінійна одиниця. Ця функція активації виправляє деякі проблеми з ReLU та зберігає деякі позитивні моменти. Для цієї функції активації вибирається значення альфа α; стандартне значення становить від 0,1 до 0,3

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

Якщо введене значення x > 0, то воно залишиться тим самим, що і ReLU. Але на цей раз, якщо вхідне значення x < 0, то отримане буде значення дещо нижче від нуля.

Отримане значення y залежить як від введення значення x, так і від параметра альфа-α, яку можна налаштувати за потреби. Крім того, вводиться експонента e^x, що означає, що ELU є обчислювально дорожчим, ніж ReLU.

Функція ELU побудована нижче з α = 0,2.

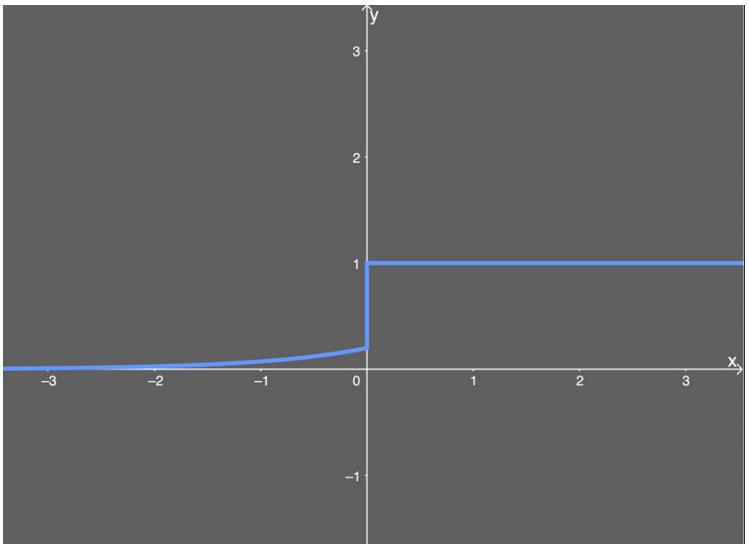


Похідна:

$$\mathrm{ELU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \mathrm{ELU}(x) + \alpha & \text{if } x \leq 0 \end{cases}$$

Вихідне значення у дорівнює 1, якщо x > 0. Вихід є функцією ELU (не диференційованою) плюс альфа-значення, якщо вхід x < нуля.

Графік:



Тут уникається проблема мертвих градієнтів та зберігається деяка обчислювальна швидкість, набута функцією активації ReLU - тобто у все ще будуть деякі мертві компоненти.

Плюси

1. Уникається проблема мертвих градієнтів.
2. Отримуються негативні результати, що допомагає мережі підтягувати ваги та ухили в правильних напрямках.
3. Під час обчислення градієнта виробляються не нульові активації.

Мінуси

1. Більший час обчислення через включену експоненту.
2. Не уникає проблеми з градієнтом, що вибухає
3. Нейронна мережа не вивчає значення альфа

Leaky ReLU

Ця функція активації також має значення альфа α, яке зазвичай становить від 0,1 до 0,3. Функція активації Leaky ReLU часто використовується, але вона має деякі недоліки порівняно з ELU, але також і деякі плюси порівняно з ReLU.

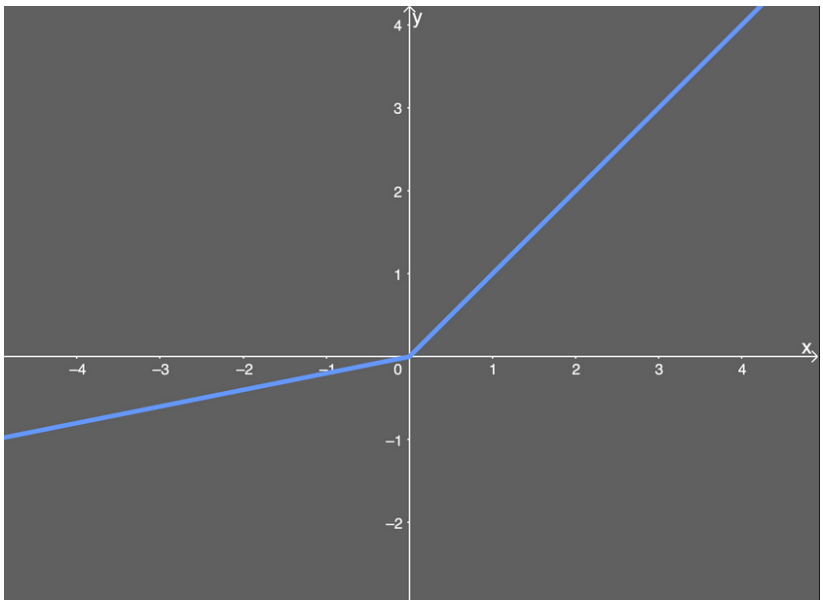
Leaky ReLU приймає цю математичну форму

$$\mathrm{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Отже, якщо вхід x > 0, то = x. Якщо вхід < 0, вихід буде рівен альфа-α разів введення.

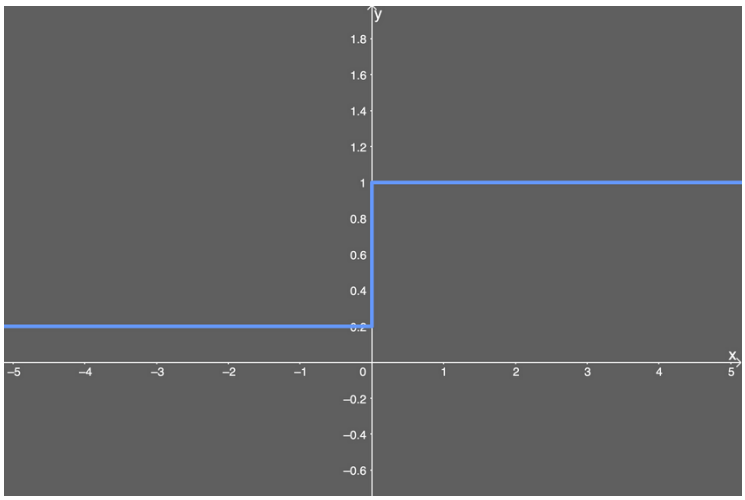
Це означає, що так вирішуємо проблема мертвих градієнтів, оскільки значення градієнтів вже не можуть затримуватися на нулі - також ця функція дозволяє уникнути зниклої градієнтної проблеми.

Графік Leaky ReLU за α = 0,2:



похідна функції Leaky ReLU:

$$\mathrm{LReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$$



Плюси

1. Як і ELU, уникаєтсья проблема мертвих градієнтів, оскільки допускається невеликий градієнт при обчисленні похідної.

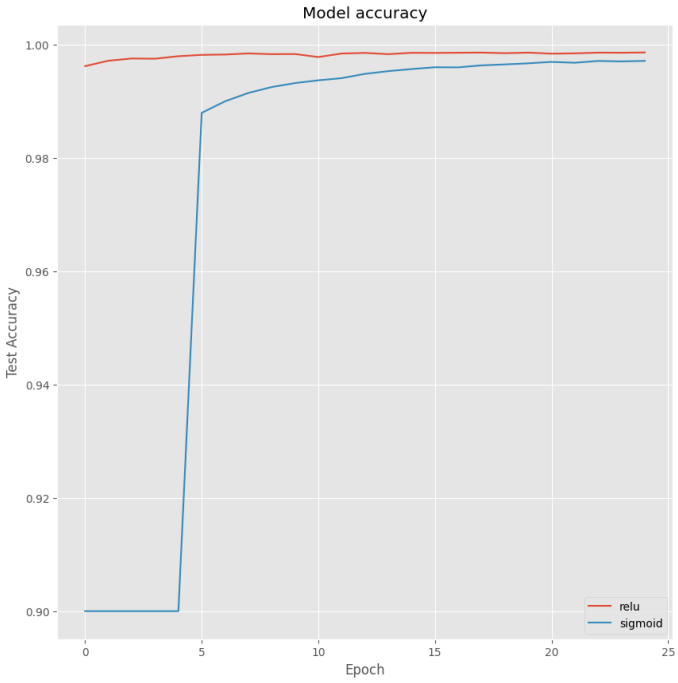
2. Швидше обчислюється, ніж ELU, оскільки відсутня експонента.

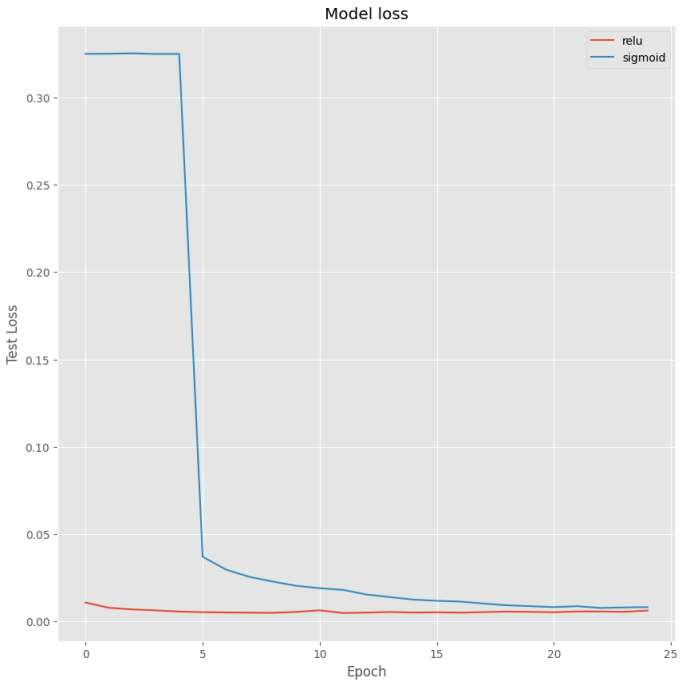
Мінуси

1. Не уникає проблеми з градієнтом, що вибухає
2. Нейронна мережа не вивчає значення альфа

Нижче наведемо вплив різних функцій активації на нейронну мережу. За основу візьмемо CNN, що буде працювати з MNIST датасетом. Мережа буде навчатися на протязі 25 епох для кожної функції активації.

Нижче наведені результати тренування мережі:





Як видно з другого графіка, у випадку сигмоїда в перших чотирьох епохах, кількість втрат практично не зменшувалася, що може вказувати на появу зникаючих градієнтів. Як і очікувалося, у випадку ReLU, такої поведінки не було, натомість модель швидко досягла відмінних значень. Таке швидке досягнення такого низького рівня втрат, швидше за все, спричинено добре підібраною архітектурою мережі.

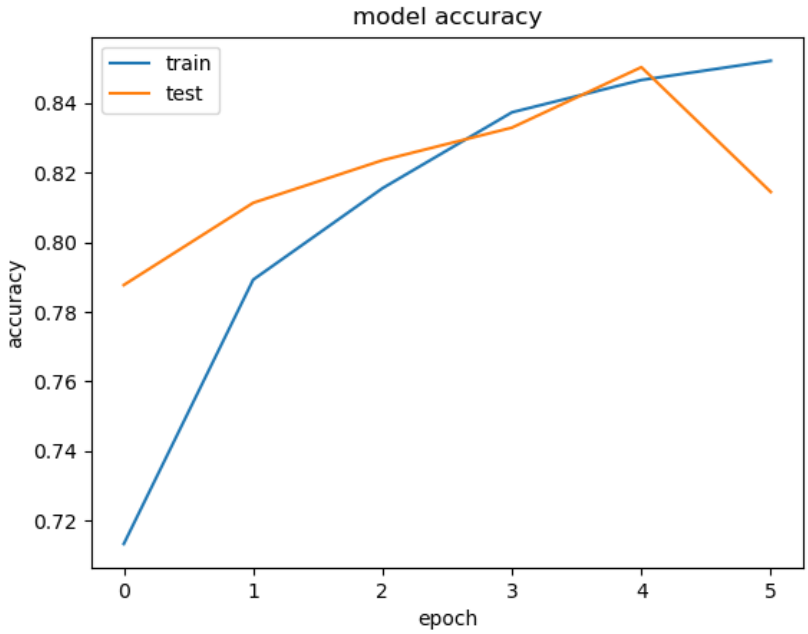
Обрізання градієнта для оптимізації навчання

Оскільки обрізання градієнта призводить до зведення значень градієнтів до певних меж, якщо вони за них виходять, то теоретично обрізання можна використовувати для пришвидшення навчання нейронних Мереж.

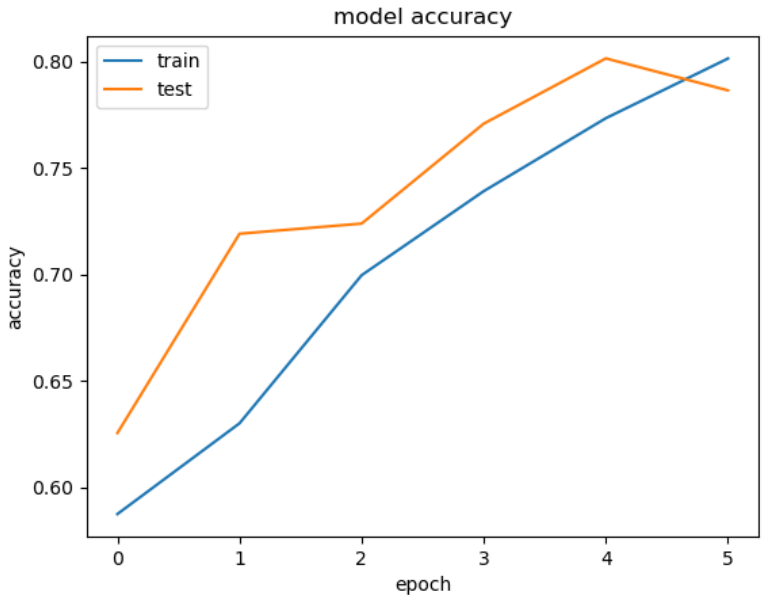
Нижче наведено результати тренування LSTM мережі, що оцінює користувацькі коментарі до фільмів і оцінює чи вони позитивні, чи негативні.

Аналогічно до моделі з архітектурою MLP, використовується архітектура, що сама по собі може нівелювати зникаючі градієнти. Також було впроваджено обрізання верхньої та нижньої меж.

Без використання обрізання(A)



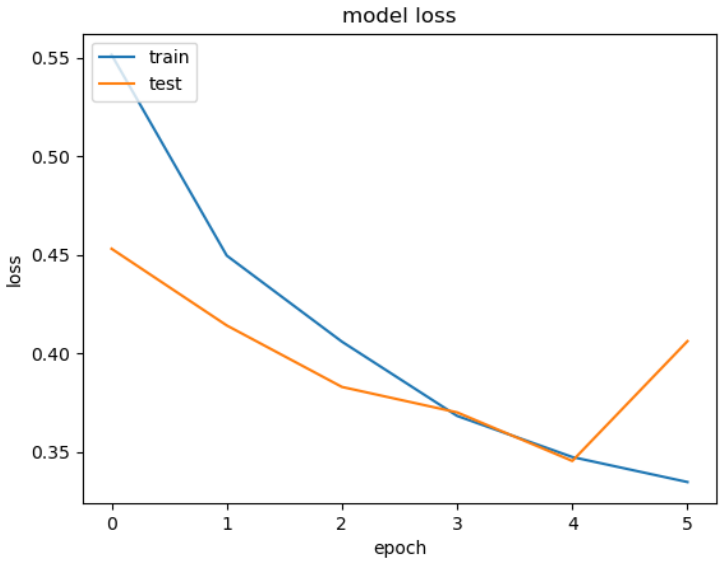
З використанням обрізання(B)



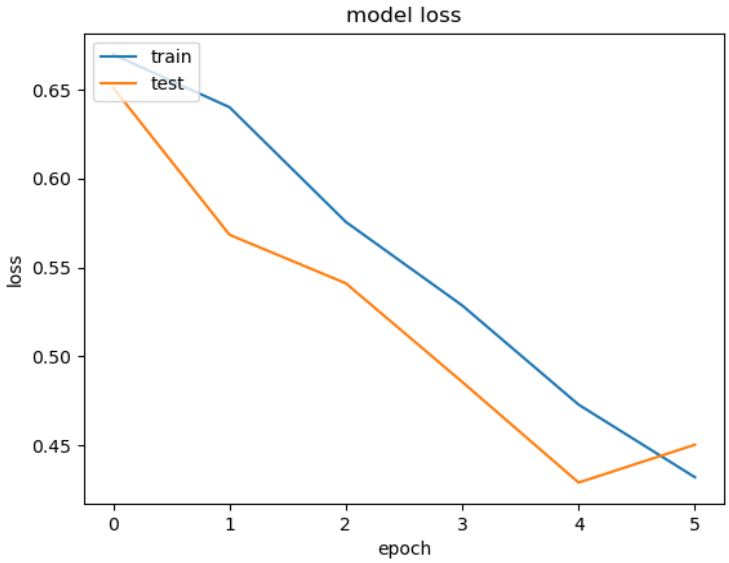
Точність моделі

З допомогою обрізання було досягнуто кращої точності, як видно з графіків. Цілком імовірно, але не точно, що різке падіння в точності в графіку А під час тестування, було спричинене певним вибухом градієнта, що призвело до надто великого зсуву ваг і, в результаті, до різкої зміни точності, в даному випадку у гіршу сторону.

Без використання обрізання(A)



З використанням обрізання(B)



Втрати моделі

Аналогічно до ситуації з точністю, різке збільшення втрат, імовірно, було спричинене вибухом градієнта, чого не відбулося у випадку, де було використано обрізання.

Як видно з вищенаведених графіків, точності обидвох варіацій моделі досягли досить схожих точностей, а саме 0.8141 у випадку моделі, де не використовувалось обрізання градієнтів та 0.7832, де використовувалось.

В даному випадку, дана оптимізація не є надто корисною, оскільки дана мережа є досить простою та й розмір датасету не був надзвичайно великим. Однак для складніших мереж дана модифікація може зекономити досить багато часу.

Що цікаво, модель В завершила процес тренування набагато швидше, ніж модель А.

ОбГОВОРЕННЯ РЕЗУЛЬТАТІВ ЕКСПЕРИМЕНТІВ

Багатошаровий персептрон(MLP) з вибухаючим та зникаючим градієнтами

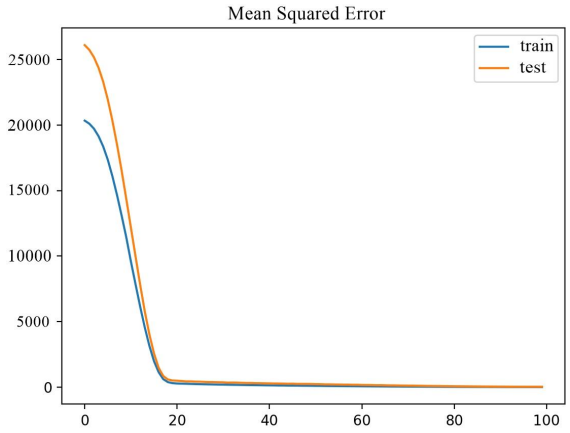
Для того, щоб продемонструвати вплив вибухаючого градієнта на процес навчання нейронної мережі та як обрізання градієнта може покращити навчання мережі, було використано Багатошаровий персептрон(MLP) для проблеми регресії.

```
Train: nan, Test: nan
```

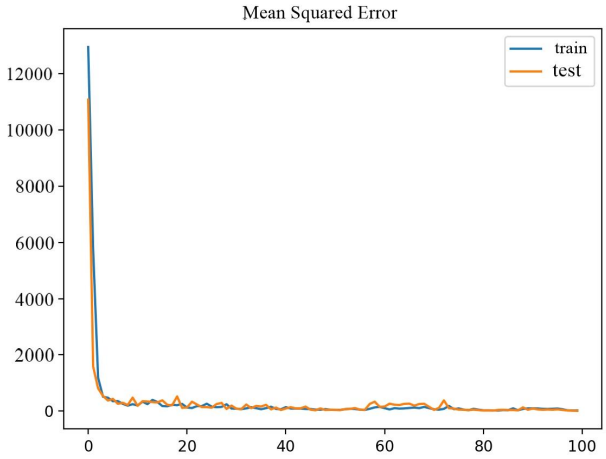
З першого результату видно, що без використання масштабування градієнтної норми чи обрізання градієнта, значення швидко перетворилися в NaN(Not a Number). Це свідчить про необхідність певного втручання стосовно цільової змінної для вивчення цієї проблеми. Для того, щоб вирішити дану штучну проблему було використано відповідно масштабування градієнтної норми та обрізання градієнта.

Різниця між першим та другим версіями мережі є надзвичайними, адже в другій версій було не просто успішно пройшов процес навчання нейронної мережі, а й було досягнуто досить непоганих результатів.

Перший графік показав, Що модель вчиться досить швидко, досягаючи середньоквадратичних втрат помилок, менших, ніж 100, після лише кількох епох.



У другому експерименті можна бачити, що в даному випадку модель здатна вивчити проблему без вибуху градієнтів, досягнувши значення середньоквадратичної похибки нижче 10, як під час тренування, так і на тестових наборах.

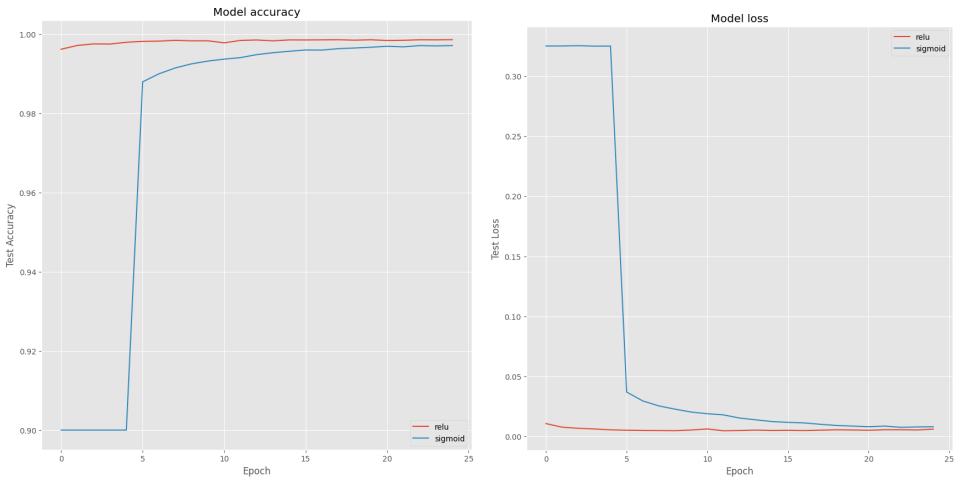


В даній ітерації мережі, було досягнуто ще більшої швидкості навчання, що дозволяє пришвидшити процес навчання мережі й також досягти кращих результатів.

В даному прикладі навмисно було обрано архітектуру MLP, для того, щоб уникнути зникаючих градієнтів, однак варто зауважити, що під час обрізання встановлюються як верхня, так і нижня межі

Використання різних функцій активації для усунення проблеми зникаючих градієнтів

результати тренування мережі:



В даному розділі було наведемо вплив різних функцій активації на нейронну мережу. За основу було взято CNN, що працювало з датасетом MNIST. Мережа навчалася на протязі 25 епох для кожної функції активації.

Як було видно з другого графіка, у випадку сигмоїда в перших чотирьох епохах, кількість втрат практично не зменшувалася, що може вказувати на появу зникаючих градієнтів. У випадку з ReLU, такої поведінки не було, натомість модель швидко досягла відмінних значень. Таке швидке досягнення такого низького рівня втрат, швидше за все, спричинено добре підібраною архітектурою мережі.

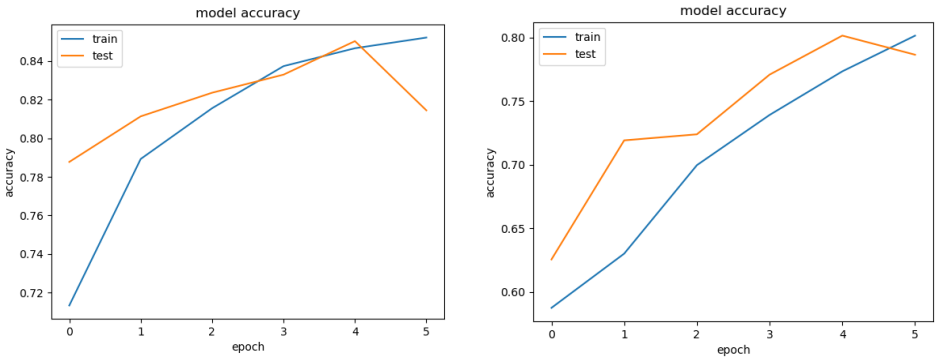
Обрізання градієнта для оптимізації навчання

Тут розглядалося використання обрізання для пришвидшення навчання нейронних Мереж.

В якості архітектури було використано LSTM мережу та за датасет було взято датасет IMDB, що містить коментарі юзерів для фільмів. Мережа розглядає користувацькі коментарі до фільмів і оцінює чи вони позитивні, чи негативні.

Аналогічно до моделі з архітектурою MLP, використовується архітектура, що сама по собі може нівелювати зникаючі градієнти. Також було впроваджено обрізання верхньої та нижньої меж.

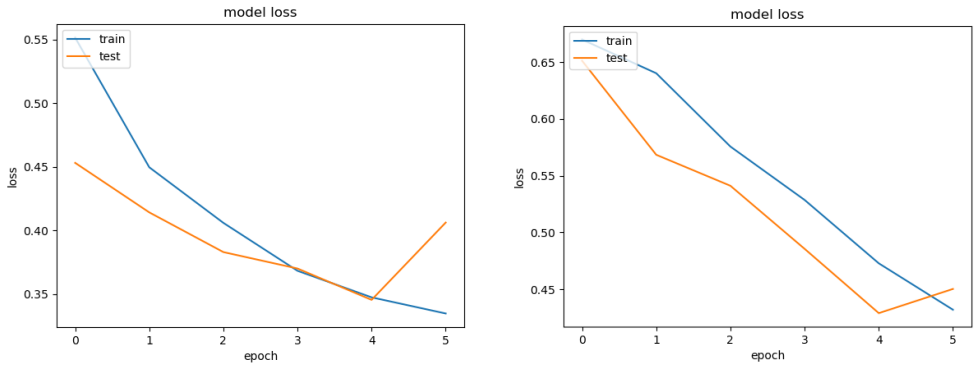
Без використання обрізання(A) З використанням обрізання(B)



Точність моделі

З допомогою обрізання було досягнуто кращої точності, як видно з графіків в попередньому розділі. Цілком імовірно, але не точно, що різке падіння в точності в графіку А під час тестування, було спричинене певним вибухом градієнта, що призвело до надто великого зсуву ваг і, в результаті, до різкої зміни точності, в даному випадку у гіршу сторону.

Без використання обрізання(A) З використанням обрізання(B)



Втрати моделі

Аналогічно до ситуації з точністю, різке збільшення втрат, імовірно, було спричинене вибухом градієнта, чого не відбулося у випадку, де було використано обрізання.

Як видно з вищенаведених графіків, точності обидвох варіацій моделі досягли досить схожих точностей, а саме 0.8141 у випадку моделі, де не використовувалось обрізання градієнтів та 0.7832, де використовувалось.

В даному випадку, дана оптимізація не є надто корисною, оскільки дана мережа є досить простою та й розмір датасету не був надзвичайно великим. Однак для складніших мереж дана модифікація може зекономити досить багато часу.

Що цікаво, модель B завершила процес тренування набагато швидше, ніж модель A.

ВИСНОВКИ

В даній курсовій роботі було розглянуто проблематику феномену зникаючих та вибухових градієнтів, причини їх появи та потенційні способи вирішення цих проблем, серед яких основна увага була зосереджена на методі обрізання градієнтів. Також було розглянуто концепт використання методу обрізання градієнтів з метою прискорення тренування нейронних мереж.

Проблема зникаючого градієнта - це проблема, яка стосується ваг в попередніх шарах мережі. Під час тренування, стохастичний градієнтний спуск (або SGD) працює для обчислення градієнта втрат відносно ваг у мережі. Так, іноді, градієнт ваг у перших шарах мережі може стати надзвичайно малим, що може призвести до уповільнення або, навіть, зупинки процесу навчання мережі.
Проблема вибуху градієнта хоч і спричиняється тими ж самими факторами, однак її наслідки сильно відрізняються. Вибух градієнта може призвести до того, що значення градієнтів перетворяться на NaN, через що мережа втратить можливість навчатися. Альтернативно, значення градієнтів можуть і не перетворитися на NaN, однак тоді ваги ніколи не досягнуть своїх оптимальних значень.

Є декілька методів, які дозволяють вирішити ці проблеми. В даній роботі було розглянуто метод відсікання градієнта та використання певний архітектур, а саме MLP і LSTM.

Відсікання Градієнта - це техніка, яка справляється із вибуховими градієнтами. Ідея відсікання градієнта дуже проста: якщо градієнт стає занадто великим, то його масштаб підлягає зміні таким чином, щоб він став меншим.

Відсікання Градієнта, також можливо використовувати з метою оптимізації часу навчання мережі, однак це сильно залежить від її архітектури і розміру датасету. Також така модифікація може призвести до незначної втрати точності мережі.

Список використаних джерел

1. Avoiding the vanishing gradients problem using gradient noise addition - Abien Fred Agarap
2. Gradient descent - wikipedia.org
3. Vanishing And Exploding Gradient Problems - Jekfine
4. Stochastic gradient descent - wikipedia.org
5. What is Gradient Clipping? - Wanshun Wong
6. Exploding and Vanishing Gradients - Roger Grosse
7. A Gentle Introduction to Exploding Gradients in Neural Networks - Jason Brownlee
8. The Vanishing Gradient Problem - Chi-Feng Wang
9. Reference: Ian Goodfellow et. al, “Deep Learning”, MIT press, 2016

Додаток

1. Багатошаровий персептрон(MLP) з вибухаючим та зникаючим градієнтами, програмна реалізація:

```
# mlp with unscaled data for the regression problem
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

```python
# mlp with unscaled data for the regression problem with gradient norm scaling
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot

# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
model.compile(loss='mean_squared_error', optimizer=opt)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

```
# mlp with unscaled data for the regression problem with gradient clipping
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot

# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
opt = SGD(lr=0.01, momentum=0.9, clipvalue=5.0)
model.compile(loss='mean_squared_error', optimizer=opt)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

2. Використання різних функцій активації для усунення проблеми зникаючих градієнтів, програмна реалізація:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D, Activation, LeakyReLU
from keras.layers.noise import AlphaDropout
from keras.utils.generic_utils import get_custom_objects
from keras import backend as K
from keras.optimizers import Adam

(x_train, y_train), (x_test, y_test) = mnist.load_data()


def preprocess_mnist(x_train, y_train, x_test, y_test):
    # Normalizing all images of 28x28 pixels
    x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
    x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
    input_shape = (28, 28, 1)

    # Float values for division
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')

    # Normalizing the RGB codes by dividing it to the max RGB value
    x_train /= 255
    x_test /= 255

    # Categorical y values
    y_train = to_categorical(y_train)
    y_test = to_categorical(y_test)

    return x_train, y_train, x_test, y_test, input_shape


x_train, y_train, x_test, y_test, input_shape = preprocess_mnist(x_train, y_train, x_test, y_test)


def build_cnn(activation,
              dropout_rate,
              optimizer):
    model = Sequential()
```

```
    if (activation == 'selu'):
        model.add(Conv2D(32, kernel_size=(3, 3),
                         activation=activation,
                         input_shape=input_shape,
                         kernel_initializer='lecun_normal'))
        model.add(Conv2D(64, (3, 3), activation=activation,
                         kernel_initializer='lecun_normal'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(AlphaDropout(0.25))
        model.add(Flatten())
        model.add(Dense(128, activation=activation,
                        kernel_initializer='lecun_normal'))
        model.add(AlphaDropout(0.5))
        model.add(Dense(10, activation='softmax'))
    else:
        model.add(Conv2D(32, kernel_size=(3, 3),
                         activation=activation,
                         input_shape=input_shape))
        model.add(Conv2D(64, (3, 3), activation=activation))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))
        model.add(Flatten())
        model.add(Dense(128, activation=activation))
        model.add(Dropout(0.5))
        model.add(Dense(10, activation='softmax'))

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

    return model



act_func = ['leaky-relu', 'relu', 'sigmoid']
result = []
```

```
for activation in act_func:
    print('\nTraining with -->{0}<-- activation function\n'.format(activation))

    model = build_cnn(activation=activation,
                      dropout_rate=0.2,
                      optimizer=Adam())

    history = model.fit(x_train, y_train,
                        validation_split=0.20,
                        batch_size=128,  # 128 is faster, but less accurate. 16/32 recommended
                        epochs=25,
                        verbose=1,
                        validation_data=(x_test, y_test))

    result.append(history)

    K.clear_session()
    del model

print(result)

new_act_arr = act_func[1:]

new_results = result[1:]


def plot_act_func_results(results, activation_functions=[]):
    plt.figure(figsize=(10, 10))
    plt.style.use('fast')

    # Plot validation accuracy values
    for act_func in results:
        plt.plot(act_func.history['val_accuracy'])

    plt.title('Model accuracy')
    plt.ylabel('Test Accuracy')
    plt.xlabel('Epoch')
    plt.legend(activation_functions)
    plt.show()

    # Plot validation loss values
    plt.figure(figsize=(10, 10))

    for act_func in results:
        plt.plot(act_func.history['val_loss'])
```

```
    plt.title('Model loss')
    plt.ylabel('Test Loss')
    plt.xlabel('Epoch')
    plt.legend(activation_functions)
    plt.show()


plot_act_func_results(new_results, new_act_arr)
```

3. Обрізання градієнта для оптимізації навчання, програмна реалізація:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import re
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers.core import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import Tokenizer

movie_reviews = pd.read_csv("datasets\IMDB Dataset.csv")
movie_reviews.isnull().values.any()
print("movie_reviews.shape: ", movie_reviews.shape)

print(movie_reviews.head(5))

print(movie_reviews["review"][3])


def preprocess_text(sen):
    # Removing html tags
    sentence = remove_tags(sen)

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Single character removal
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence)

    # Removing multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence)

    return sentence
```

```
TAG_RE = re.compile(r'<[^>]+>')


def remove_tags(text):
    return TAG_RE.sub('', text)


X = []
sentences = list(movie_reviews['review'])
for sen in sentences:
    X.append(preprocess_text(sen))

print(X[3])

y = movie_reviews['sentiment']

y = np.array(list(map(lambda x: 1 if x == "positive" else 0, y)))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(X_train)

X_train = tokenizer.texts_to_sequences(X_train)
X_test = tokenizer.texts_to_sequences(X_test)

# Adding 1 bcuz of the reserved 0 index
vocab_size = len(tokenizer.word_index) + 1

maxlen = 100


X_train = pad_sequences(X_train, padding='post', maxlen=maxlen)
X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)

from numpy import array
from numpy import asarray
from numpy import zeros

embeddings_dictionary = dict()


glove_file = open('datasets/glove.6B/glove.6B.100d.txt', encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary[word] = vector_dimensions
glove_file.close()
```

```
embedding_matrix = zeros((vocab_size, 100))
for word, index in tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix[index] = embedding_vector

print(embedding_matrix.shape)

model = Sequential()
embedding_layer = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=maxlen, trainable=False)
model.add(embedding_layer)
model.add(LSTM(128))
model.add(Dense(1, activation='relu'))
# compile model
# opt = SGD(lr=0.01, momentum=0.9, clipvalue=5.0)
model.compile(optimizer="adam", loss='binary_crossentropy', metrics=['acc'])  # adam
print(model.summary())

history = model.fit(X_train, y_train, batch_size=128, epochs=6, verbose=1, validation_split=0.2)


score = model.evaluate(X_test, y_test, verbose=1)
print("Test Score: ", score[0])
print("Test Accuracy: ", score[1])

plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])

plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train','test'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train','test'], loc='upper left')
plt.show()


instance = X[23]
print(instance)
instance = tokenizer.texts_to_sequences(instance)
```

```
flat_list = []
for sublist in instance:
    for item in sublist:
        flat_list.append(item)

flat_list = [flat_list]
instance = pad_sequences(flat_list, padding='post', maxlen=maxlen)
prediction = model.predict(instance)
print(prediction)
```