МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Факультет електроніки та інформаційних технологій
Кафедра прикладної математики та моделювання складних систем

«До захисту допущено»

Завідувач кафедри

_____ Ігор КОПЛИК
(підпис) (Ім'я ПРІЗВИЩЕ)

___ _____ 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»

зі спеціальності 113 «Прикладна математика»,

освітньо-професійної програми

«Наука про дані та моделювання складних систем»

на тему: «Порівняння сучасних функцій активації у нейронних мережах»

Здобувача групи ПМ-01 Литвиненка Івана Далійовича
(шифр групи) (прізвище, ім'я, по батькові)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Іван ЛИТВИНЕНКО
(підпис) (Ім'я та ПРІЗВИЩЕ здобувача)

Керівник доцент, кандидат фіз-мат. наук Аліна ДВОРНИЧЕНКО _____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Суми – 2024

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Факультет	**електроніки та інформаційних технологій**
Кафедра	**прикладної математики та моделювання складних систем**
Рівень вищої освіти	**перший (бакалаврський)**
Галузь знань	**11 «Математика та статистика»**
Спеціальність	**113 «Прикладна математика»**
Освітня програма	**освітньо-професійна** «Наука про дані та моделювання складних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри ПМтаМСС
Ігор КОПЛИК _____
«___»_____ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧЕВІ ВИЩОЇ ОСВІТИ

Литвиненку Івану Далійовичу

1. Тема роботи: Порівняння сучасних функцій активації у нейронних мережах.
   Керівник роботи Дворниченко Аліна Василівна, доцент, кандидат фіз.-мат.наук, затверджено наказом по факультету ЕлІТ від «05» квітня 2024 р. № 0349-VI
2. Термін подання роботи здобувачем «4» червня 2024 р.
3. Вихідні дані по роботі: Монографії, ресурси інтернету, наукові статті, періодичні видання з теми дослідження; датасети MNIST та CIFAR-10.

4. Зміст розрахунково-пояснювальної записки (перелік питань, для розроблення): Робота з літературою, вибір даних та форми експериментів, прототипування експериментів.
5. Перелік графічного матеріалу: Графіки функцій активації, варіації експериментів, результати експериментів.
6. Консультанти проєкту (роботи) із зазначенням розділів проєкту, що їх стосується

Розділ	Прізвище, ініціали та посада консультатнта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «09» квітня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№	Вид робіт	Термін виконання	Примітка
1	Робота з літературою	14.04.2024 р.	
2	Вибір даних та форми експериментів	16.04.2024 р.	
3	Прототипування експериментів	22.04.2024 р.	
4	Рефакторинг коду	20.05.2024 р.	

Здобувач вищої освіти _____ Литвиненко І.Д.

Керівник роботи _____ Дворниченко А.В.

РЕФЕРАТ

Кваліфікаційна робота: 94 сторінки, 25 рисунків, 14 формул, 6 таблиць, 5 джерел.

Мета роботи: проведення комп'ютерних експериментів для порівняння сучасних функцій активації у нейронних мережах.

Об’єкт дослідження: властивості функцій активації, що є важливими при навчанні та експлуатації нейронних мереж.

Предмет дослідження: вплив вибору функцій активації на швидкість збіжності навчання та точність прогнозів нейронної мережі.

Методи дослідження: літературний аналіз, проектування та прототипування, комп’ютерний експеримент.

Ключові слова: НЕЙРОННА МЕРЕЖА, ФУНКЦІЯ АКТИВАЦІЇ, RECTIFIED LINEAR UNIT, EXPONENTIAL LINEAR UNIT, GAUSSIAN ERROR LINEAR UNIT, SMOOTH LOGARITHMIC UNIT, КОМП’ЮТЕРНИЙ ЕКСПЕРИМЕНТ, АПРОКСИМАЦІЯ ФУНКЦІЙ, КЛАСИФІКАЦІЯ ЗОБРАЖЕНЬ.

ЗМІСТ

ВСТУП

Нейронні мережі стали потужним інструментом у різних сферах, революціонізуючи галузі від фінансів і до охорони здоров'я. Оскільки нейронні мережі стають все більш важливими для вирішення складних проблем, пошук методів отримання кращих показників набуває першочергового значення. Одним з аспектів проектування нейронних мереж є вибір функцій активації, які відіграють важливу роль у визначенні здатності мережі до навчання та узагальнення даних.

Пошук кращих функцій активації зумовлений прагненням підвищити швидкодію мереж, ефективність та здатність мереж до узагальнення. Краща функція активації може суттєво вплинути на здатність мережі вивчати складні закономірності, зменшити швидкість навчання і підвищити точність прогнозування.

Підібрати єдину функцію активації для всіх ситуацій неможливо. Одні функції обчислювально дорожчі за інші, що особливо помітно у достатньо масштабних нейронних мережах, і що стає критичним у небезпечних сферах, де секундна затримка може коштувати комусь життя. У випадках же, коли навчання мережі для поставлених цілей є дуже дорогим, пріоритетом у виборі архітектури, а отже і функції активації, може стати швидкість навчання з ціллю мінімізації витрат на оренду комп'ютерного обладнання. Не варто забувати і про точність прогнозування, яку в деяких ситуаціях ставлять понад усе.

Дана робота спрямована на огляд сучасних функцій активацій та порівняння їх характеристик у ряді експериментів. В ній також запропоновано власну функцію з огляду на деякі практичні міркування, а також на властивості відомих функцій, що вже гарно себе показали.

РОЗДІЛ 1. ОГЛЯД СУЧАСНИХ ФУНКЦІЙ АКТИВАЦІЇ

Функції активації відіграють вирішальну роль у штучних нейронних мережах, вносячи нелінійність у модель, що дозволяє їй вивчати складні закономірності та взаємозв'язки в даних. Наприклад, вихід повнозв’язного шару можна обчислити як:

$$Y = f(XW + b), \tag{1}$$

де W – матриця ваг нейронів; b – вектор вільних членів; X – матриця, що була подана на вхід шару; f – функція активації.

Без функцій активації в нейронній мережі вона по суті перетворюється на лінійну регресійну модель. Вихід кожного шару стає лінійною комбінацією входів без будь-якої нелінійності. Як наслідок, виражальні можливості мережі будуть обмежені лінійними перетвореннями.

В останні роки було запропоновано кілька нових функцій активації для покращення продуктивності та ефективності моделей глибокого навчання. Далі будуть розглянуті деякі з найпопулярніших сучасних функцій активації, що використовуються в глибокому навчанні, обговорено їхні властивості, переваги та недоліки.

1.1. Rectified Linear Unit (ReLU)

Функція була вперше запропонована для обмеженої машини Больцмана у 2010 році [1] та має наступний вигляд:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2}$$

Дана функція прийшла на заміну сигмоїді (*sigmoid*) та вирішила проблему затухаючого градієнта (*vanishing gradient problem*), коли градієнт дуже швидко ставав настільки маленьким, що майже не доходив до перших

шарів у глибоких нейронних мережах, через що перші шари фактично не навчались.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

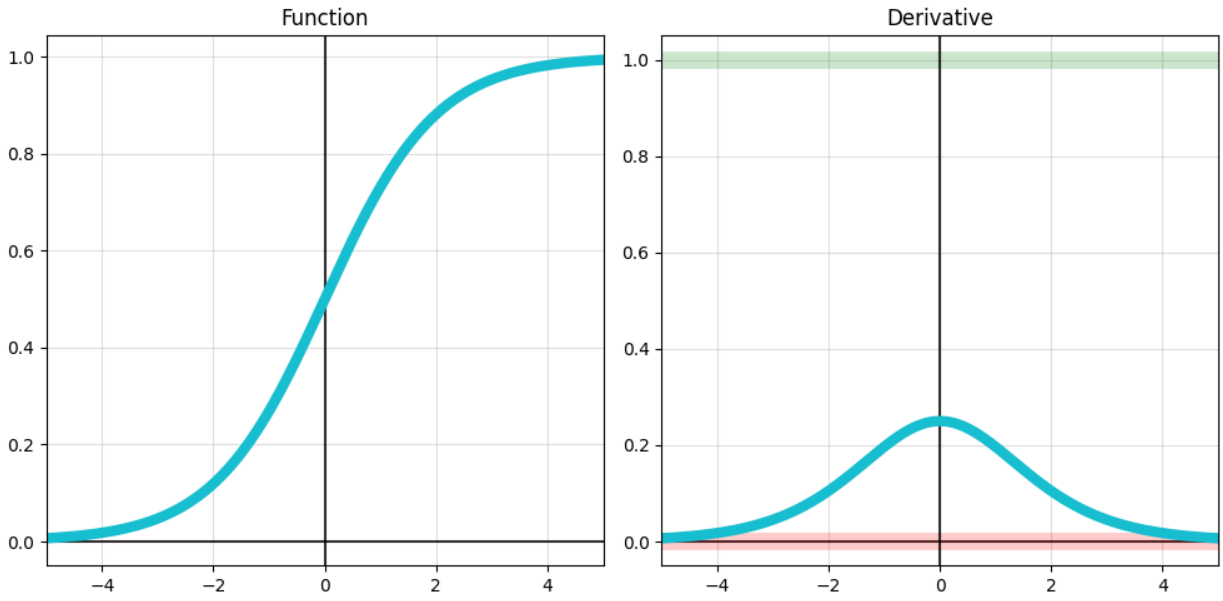

Рисунок 1. Графіки сигмоїди та її похідної.

Як можна побачити, похідна сигмоїди має найбільше значення 0.25. На практиці це означає, що при зворотньому розповсюдженні помилки кожен шар зменшує значення градієнта у 4 рази як мінімум, що і спричиняло проблему затухаючого градієнта.

Значення ReLU 0 при від'ємних значеннях аргумента дозволяє мережі користуватись властивістю розрідженості (*sparsity*) [2]. Серед її переваг розрізняють наступні:

- "Розплутування" інформації: малі коливання вхідних даних з меншою ймовірністю змістять багато ваг у мережі, якщо багато з них дорівнюють нулю.
- Ефективне представлення змінного розміру: зміна кількості активних нейронів дозволяє моделі контролювати ефективну розмірність представлення для заданих вхідних даних.

- Лінійна відокремлюваність: наявність багатьох нулів у високорозмірному представленні робить його більш легким для відокремлення за допомогою лінійних меж.
- "Розподілені, але розріджені": хоча розріджені представлення мають меншу "виразність", ніж щільні, розподіленість потенційно робить їх експоненціально кращими.

Ще однією перевагою ReLU є простота обчислення значення функції, а також її похідної. Дійсно, порівняти значення з нулем значно швидше, ніж порахувати сигмоїду. Як сигмоїда, так і її похідна вимагають розрахунку експоненти, що значно сповільнює обчислення.

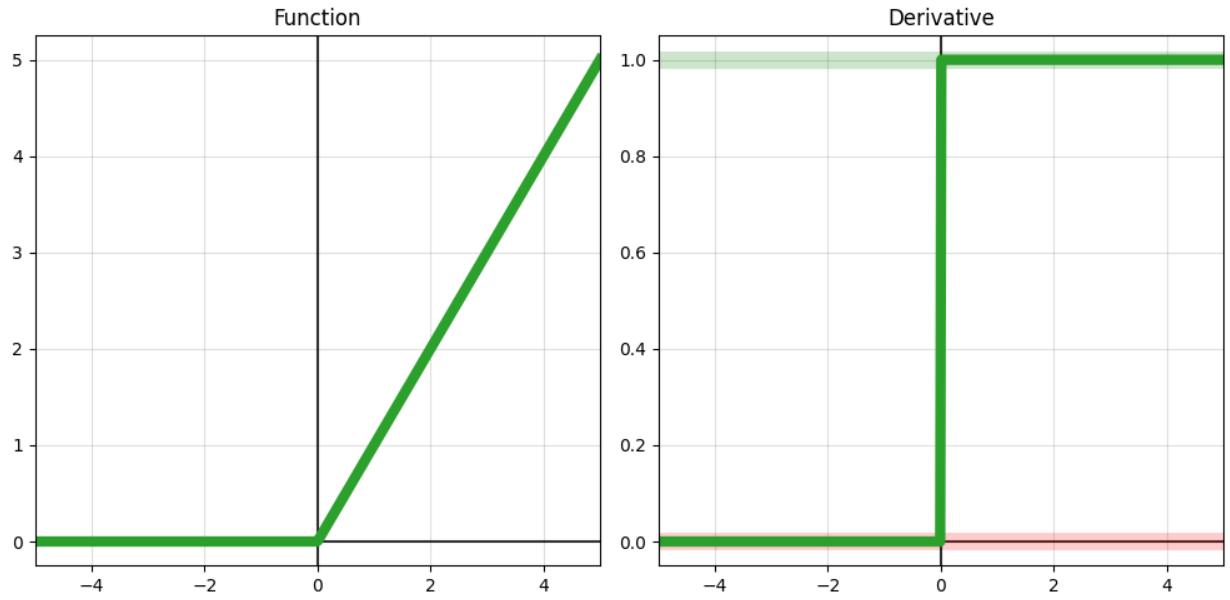

Рисунок 2. Графіки ReLU та її похідної.

На графіку похідної ReLU видно, що градієнт при переході через дану функцію або не зміниться, або стане нулем. З одного боку така властивість вирішує проблему затухаючого градієнта, а з іншого породжує проблему мертвих нейронів (*dead neurons*).

Проблема полягає в тому, що якщо раптом у процесі навчання в мережі складеться ситуація, при якій нейрон завжди буде видавати від'ємні значення, то після активації ці значення обертатимуться на нуль, а разом з ними і

градієнт, що буде через цей нейрон проходити. У результаті нейрон перестане навчатися та зменшить швидкість навчання нейронів у шарах, що стоять за ним. В залежності від глибини мережі та способу ініціалізації нейронів стається, що велика частка нейронів "обнуляється" та ніяк не впливає на роботу моделі [3].

Також на графіку похідної ReLU можна побачити причину ще одного недоліку: функція не є гладкою. Дуже малі зміни значення x біля 0 призводять до різкої зміни похідної з 0 на 1. На практиці це відображається наступним чином: градієнт для одного і того ж нейрону між ітераціями може значно відрізнятись, що зменшує стабільність навчання.

Ще однією небажаною властивістю є те, що значення ReLU є невід'ємними і, отже, мають середню активацію, більшу за нуль. Даний ефект має назву *bias shift* та призводить до вповільнення навчання [4].

Не дивлячись на деякі недоліки, ReLU завдяки своїй простоті та ефективності найчастіше обирається в якості функції активації для прихованих шарів нейронних мереж.

1.2. Exponential Linear Unit (ELU)

У 2015 році вийшла стаття “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)” [4], у якій було запропоновано активаційну функцію ELU. У роботі демонструвалось підвищення якості класифікації зображень, а також пришвидення навчання при використанні даної активації.

$$\mathrm{ELU}(x) = \begin{cases} x & x \geq 0 \\ \alpha(\exp(x) - 1) & x < 0 \end{cases} \quad (4)$$

За замовчуванням значення α дорівнює 1. При такому значенні гіперпараметра функція є гладкою, що позитивно позначається на навчанні.

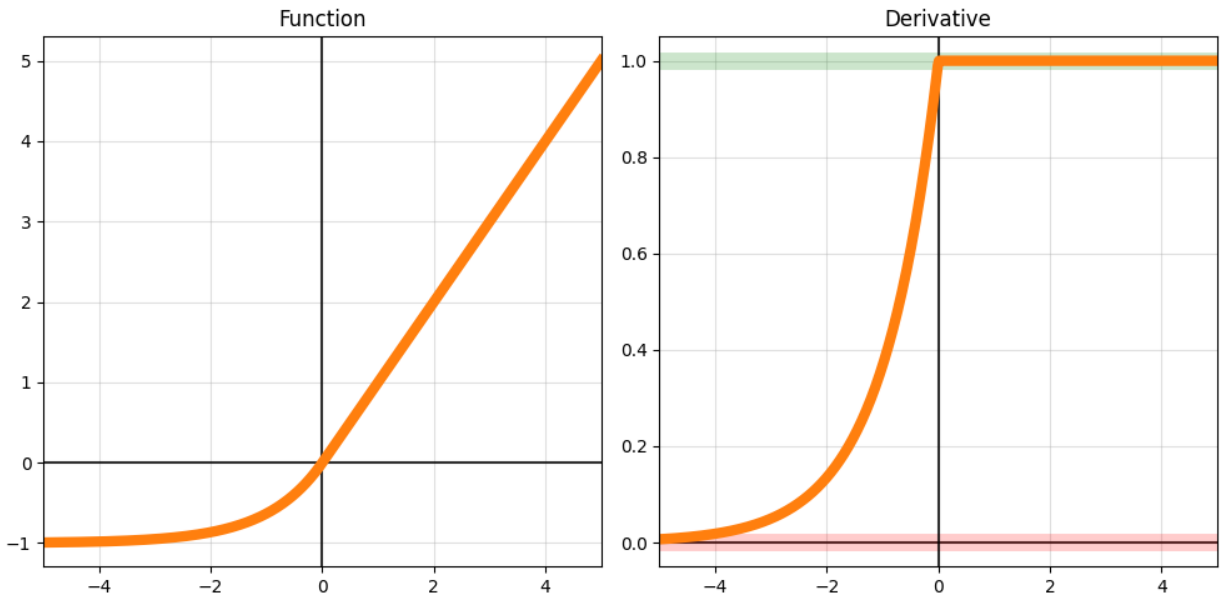

Рисунок 3. Графіки ELU та її похідної ($\alpha = 1$).

На відміну від ReLU, ELU може приймати від'ємні значення, що дозволяє їй наближати середнє значення активації до нуля, подібно до пакетної нормалізації (*batch normalization*), але з меншою обчислювальною складністю. Зсув середнього до нуля прискорює навчання, наближаючи нормальний градієнт до природного градієнта через зменшення ефекту bias shift [4].

Незважаючи на нелінійність у своїй лівій частині, ELU все ще забезпечує шумостійкий стан деактивації. ELU досягає майже фіксованого від'ємного значення при менших входах і тим самим зменшує варіацію та інформацію, що поширюється вперед.

З іншого боку, ELU втрачає властивість sparse activation. Також як сама функція, так і її похідна має експоненціонування у своєму складі, що робить її обчислення та навчання з нею повільнішими. І хоча загальна швидкість збіжності навчання компенсує уповільнення кожної окремої ітерації навчання (ітерація займає більше часу, проте навчання потребує менше ітерацій), повільність обчислення нікуди не дівається. Це може бути критичним для задач, де складність завдання вимагає використовувати глибоку мережу, а його характер потребує видавати результат дуже швидко

(такими задачами можуть бути, наприклад, керування авто або детекція небезпечних ситуацій на виробництві). У подібних випадках використання ReLU є набагато доцільнішим.

1.3. Gaussian Error Linear Unit (GELU)

У 2016 році вийшла стаття “Gaussian Error Linear Units (GELUs)” [5], яка запропонувала ще одну функцію активації з зовсім інших міркувань: GELU. Ідея полягала в тому, що входи нейронів мають тенденцію до нормального розподілу, особливо при використанні проміжної нормалізації. Було отримано гладку функцію з неперервною похідною, яка використовувала даний факт:

$$\mathrm{GELU}(x) = x\Phi(x), \quad \Phi(x) = \mathrm{P}(x \leq X), \quad X \sim \mathcal{N}(0,1) \tag{5}$$

За даною формулою ми масштабуємо значення x у стільки разів, наскільки воно більше за інші вхідні дані. Через це для достатньо малих та великих значень функція веде себе як ReLU, що надає їй деякі корисні властивості останньої, наприклад властивість розрідженості.

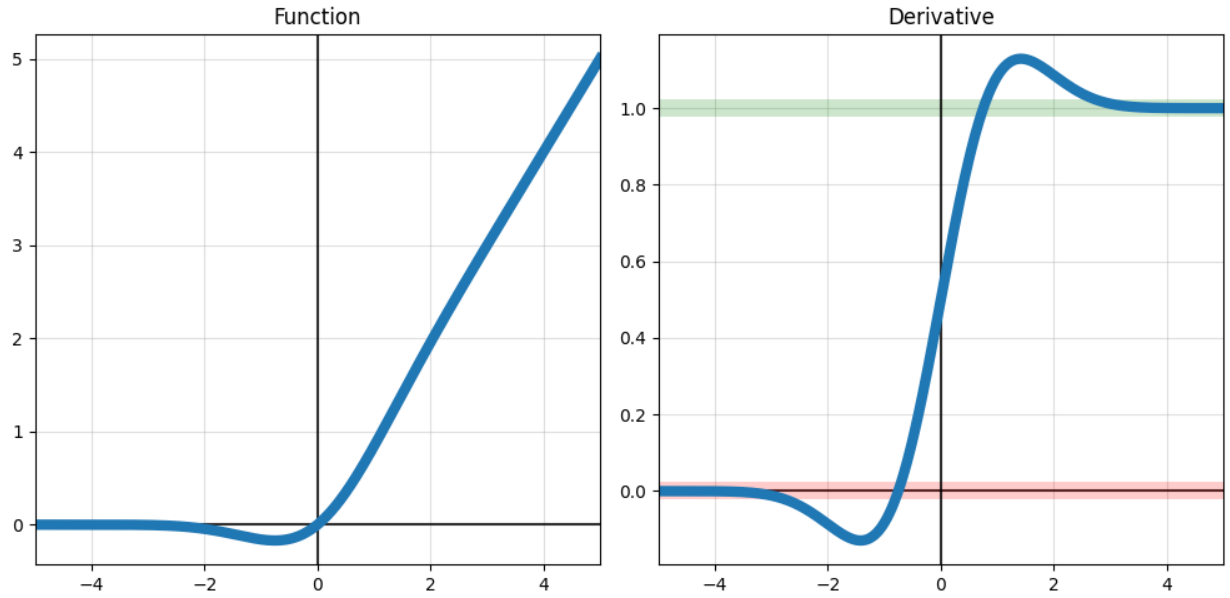

Рисунок 4. Графіки GELU та її похідної ($\mu = 0,\ \sigma = 1$).

GELU неопукла, немонотонна, не є лінійною в додатній області і має кривизну в усіх точках. Натомість ReLU та ELU, які є опуклими та монотонними активаціями, є лінійними в додатній області і, таким чином, можуть не мати кривизни. Автори оригінальної роботи наголошують, що підвищена кривизна і немонотонність може дозволити GELU апроксимувати складні функції легше, ніж ReLU або ELU.

У роботі автори порівняли ефективність ReLU, ELU та GELU на багатьох типах завдань та прийшли до висновку, що GELU стабільно випереджає за точністю ELU та ReLU, що робить її гідною альтернативою при виборі функції активації.

РОЗДІЛ 2. УЗАГАЛЬНЕННЯ БАЖАНИХ ВЛАСТИВОСТЕЙ ФУНКЦІЇ АКТИВАЦІЇ

2.1. Узагальнені властивості функцій активації

Усі з перелічених функцій мають деякі спільні властивості.

По-перше, для досить великих додатніх значень аргументу всі функції поводяться як лінійні. Це можна формалізувати як:

$$\lim_{x \to +\infty} \frac{\partial f(x)}{\partial x} = 1. \tag{6}$$

По-друге, для досить великих від'ємних значень аргументу всі функції поводяться як константні. Це можна формалізувати як:

$$\lim_{x \to -\infty} \frac{\partial f(x)}{\partial x} = 0. \tag{7}$$

Також бажано, щоб функція була гладкою. Приклад ELU показує, що така властивість може дещо стабілізувати процес навчання. Достатньо буде виконання наступної умови:

$$\forall x_0 \in \mathbb{R}: \lim_{x \to x_0 - 0} \frac{\partial f(x)}{\partial x} = \lim_{x \to x_0 + 0} \frac{\partial f(x)}{\partial x} = \frac{\partial f(x_0)}{\partial x}. \tag{8}$$

Ще одну бажану властивість можна отримати з практичних міркувань.

Якщо уявно розділити нейромережу на останній шар і всі шари, що йшли перед ним, вийде feature extractor і деяка проста модель, що приймає на вхід більш інформативні подання вихідних даних.

З інтуїтивної точки зору, природно буде припустити, що в деяких випадках ми б не хотіли, щоб різні значення якоїсь властивості ставилися у відповідність одному проміжному поданню, як це відбувається, наприклад, у ReLU, де половина всіх можливих значень аргументу відображаються в 0.

З іншого боку, позбавлятися такої можливості теж би не хотілося: відомо, що всі перераховані вище функції активації вже її мають і на сьогоднішній день є стандартом.

Це приводить нас до думки, що непогано було б мати функцію, яка в різних ситуаціях може бути або не бути бієктивною. Також хотілося б, щоб модель сама в процесі навчання визначала, чи потрібна їй ця властивість: і справді, вручну визначати для кожного шару, які проміжні представлення даних мають бути рівнопотужними, непрактично навіть для невеликих моделей, не кажучи вже про індустріальні масштаби.

Із попередньо перерахованих властивостей стає зрозумілим, що бієктивною функція може бути лише коли вона є зростаючою: принаймні на тому проміжку, де буде розташована переважна більшість значень аргументу. Ми не очікуємо, що проміжні представлення даних у мережі будуть обчислюватися сотнями чи тисячами: навпаки, така поведінка сигналізує про потребу в нормалізації даних, значного зменшення кроку навчання або вказує на інші проблеми з мережею. Орієнтуватися на подібні ситуації буде неправильно, тому ми не накладатимемо обмеження на монотонність у повному сенсі слова.

2.2. Розробка функції активації на основі сформульованих гіпотез. Smooth Logarithmic Unit (SLU)

Методом перебору функцій, що задовольняли би перерахованим умовам, було отримано формулу Smooth Logarithmic Unit (SLU).

$$\mathrm{SLU}(x) = \begin{cases} x + k\ln^2(x+1) & x \geq 0 \\ k\ln^2(-x+1) - \ln(-x+1) & x < 0 \end{cases} \tag{9}$$

Формулу можна перетворити для зручності.

$$A = \ln(1+|x|) \qquad B = kA^2 \qquad \mathrm{SLU}(x) = \begin{cases} x + B & x \geq 0 \\ B - A & x < 0 \end{cases}$$

Похідна SLU виглядає наступним чином:

$$\frac{\partial}{\partial x}\mathrm{SLU}(x) = \begin{cases} 1 + \dfrac{2k\ln(x\ +\ 1)}{x+1} & x \geq 0 \\ \dfrac{1 - 2k\ln(-x\ +\ 1)}{-x+1} & x < 0 \end{cases} \tag{10}$$

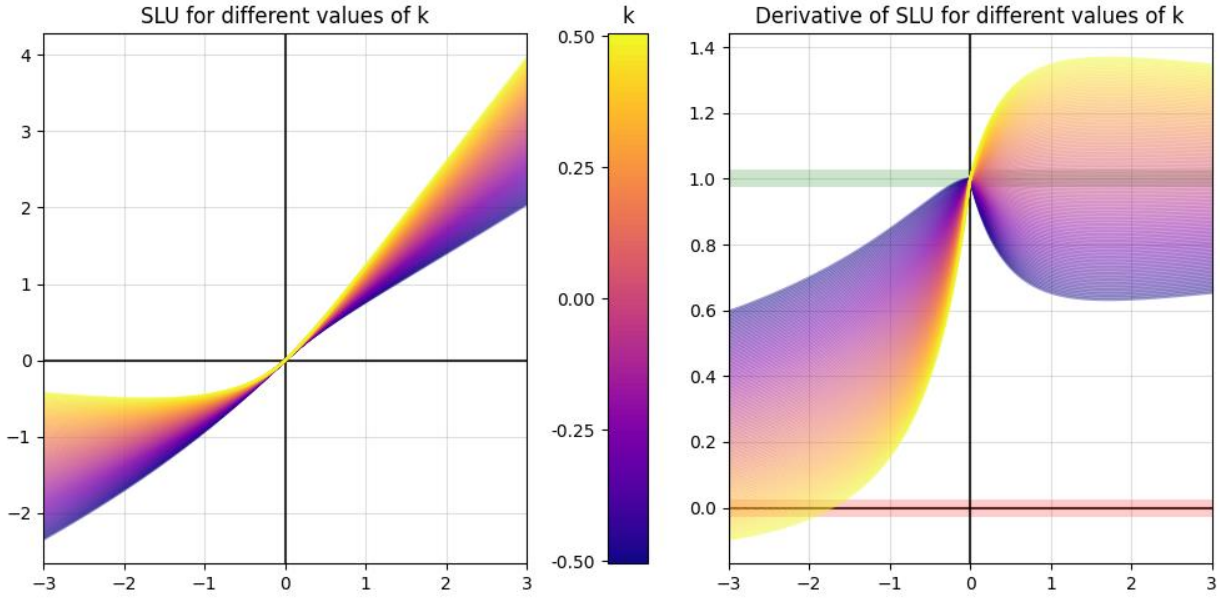

Рисунок 5. Графіки SLU та її похідної на проміжку $[-3; 3]$
для різних значень параметра k.

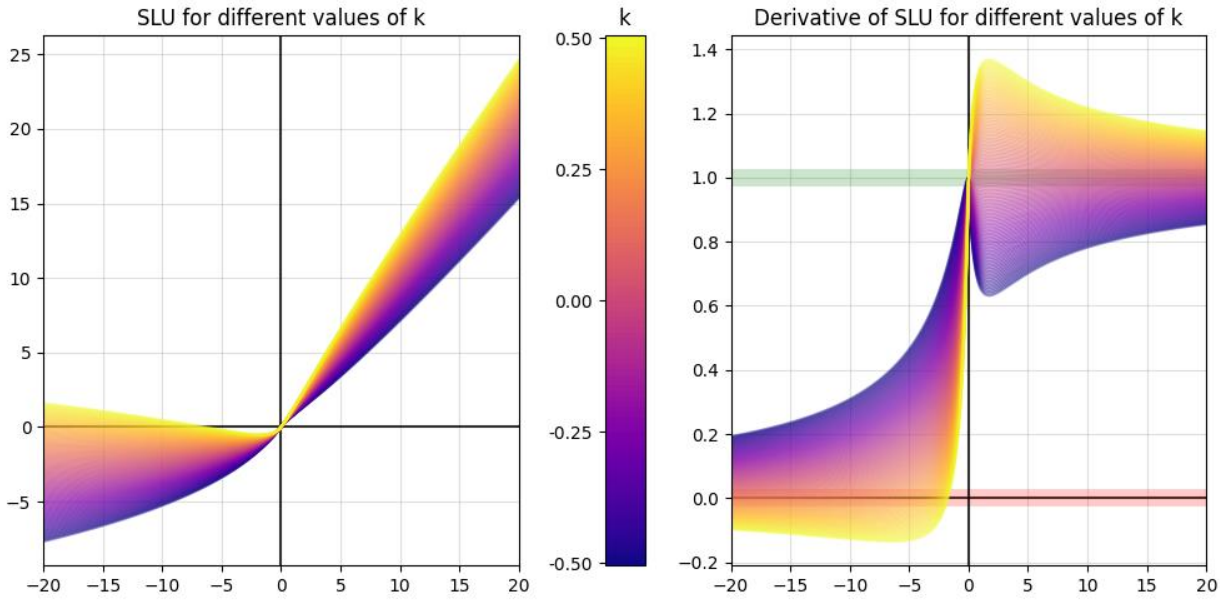

Рисунок 6. Графіки SLU та її похідної на проміжку $[-20; 20]$
для різних значень параметра k.

Продемонструємо виконання поставлених вище умов:

1. Лінійність для досить великих додатніх значень аргументу

$$\lim_{x\to+\infty}\frac{\partial f(x)}{\partial x}=1$$

$$\lim_{x\to+\infty}\left(1+\frac{2k\ln(x+1)}{x+1}\right)=1+2k\lim_{x\to+\infty}\frac{\ln(x+1)}{x+1}=1$$

2. Константність для досить великих від'ємних значень аргументу

$$\lim_{x\to-\infty}\frac{\partial f(x)}{\partial x}=0$$

$$\lim_{x\to-\infty}\left(\frac{1-2k\ln(-x+1)}{-x+1}\right)=\lim_{x\to-\infty}\frac{1}{-x+1}-2k\lim_{x\to-\infty}\frac{\ln(-x+1)}{-x+1}$$

$$=0$$

3. Гладкість

$$\forall x_0\in\mathbb{R}\colon\lim_{x\to x_0-0}\frac{\partial f(x)}{\partial x}=\lim_{x\to x_0+0}\frac{\partial f(x)}{\partial x}=\frac{\partial f(x_0)}{\partial x}$$

SLU складена з двох функцій, що є гладкими на заданих проміжках. Перевірити лишається лише точку, де ці функції поєднуються:

$$\lim_{x\to-0}\frac{\partial f(x)}{\partial x}=\lim_{x\to+0}\frac{\partial f(x)}{\partial x}=f(0)$$

$$\lim_{x\to-0}\frac{\partial f(x)}{\partial x}=\lim_{x\to-0}\left(\frac{1-2k\ln(-x\ +\ 1)}{-x+1}\right)=1$$

$$\lim_{x\to+0}\frac{\partial f(x)}{\partial x}=\lim_{x\to+0}\left(1+\frac{2k\ln(x\ +\ 1)}{x+1}\right)=1$$

4. Опціональна бієктивність для більшості значень аргументу

$$\frac{\partial}{\partial x}SLU>0$$

1) $x\geq 0$

$$\frac{\partial}{\partial x}\mathrm{SLU}_+=1+\frac{2k\ln(x+1)}{x+1}>0$$

$$k>-\frac{x+1}{2\ln(x+1)}$$

$$\max_{x\in\mathbb{R}_+}\left(-\frac{x+1}{2\ln(x+1)}\right)=-\frac{e}{2}$$

$$k>-\frac{e}{2}\approx-1.359$$

2) $x<0$

$$\frac{\partial}{\partial x}\mathrm{SLU}_-=\frac{1-2k\ln(-x+1)}{-x+1}>0$$

$$k<\frac{1}{2\ln(-x+1)}$$

$$\min_{x\in\mathbb{R}_-}\left(\frac{1}{2\ln(-x+1)}\right)\to 0$$

$$k\leq 0$$

Похідна дорівнює нулю лише в одній точці, можемо включити $-e/2$. Таким чином, функція бієктивна при $k\in[-e/2;\ 0]$.

Варто зазначити, що верхня межа k дорівнює нулю, лише якщо ми хочемо накладати обмеження на всі від'ємні значення x, що не завжди відповідає дійсності. Як було вказано вище, достатньо буде зробити функцію бієктивною там, де ми очікуємо отримувати всі значення аргументу. Максимальне значення k по x та мінімальне значення x по k можна знайти за наступними формулами:

$$k_{max}(x_{min})=\frac{1}{2\ln(-x_{min}+1)};\tag{11}$$

$$x_{min}(k_{max})=-\exp\frac{1}{2k_{max}}+1.\tag{12}$$

Наприклад, при $x_{min}=-3$ отримаємо $k_{max}=0.361$.

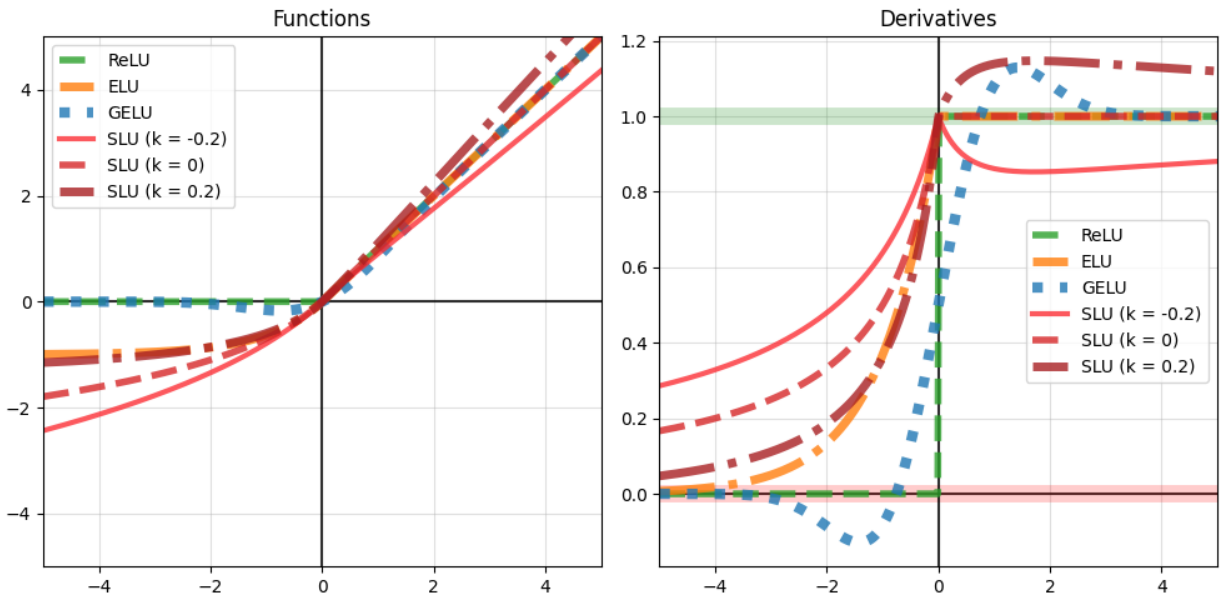

Рисунок 7. Візуальне порівняння розглянутих на даний момент функцій активації.

Звернемо увагу на те, що візуально графіки ELU та SLU доволі схожі. Щоб це пояснити, розкладемо обидві функції в ряд Тейлора (для простоти візьмемо $k = 0$ та $\alpha = 1$):

$$\mathrm{ELU}_{\alpha=1}(x) = \begin{cases} x & x \geq 0 \\ \exp(x) - 1 & x < 0 \end{cases}$$

$$\mathrm{SLU}_{k=0}(x) = \begin{cases} x & x \geq 0 \\ -\ln(-x\ +\ 1) & x < 0 \end{cases}$$

$$\exp(x) - 1 \approx x + \frac{x^2}{2!} + \frac{x^3}{3!}$$

$$-\ln(-x + 1) \approx x + \frac{x^2}{2} + \frac{x^3}{3}$$

Таким чином, різниця між значеннями функцій становить $x^3/6$ для від'ємних значень аргумента та 0 для додатних. Робимо висновок, що SLU за необхідністю може повторювати поведінку ELU для достатньо малих значень аргумента.

Знайдемо k, при якому відстань між ELU та SLU буде мінімальною. Вимірювати будемо у межах $\pm 3\sigma$ для стандартного нормального розподілу (на відрізку $x \in [-3; 3]$).

Відстань обчислюватимемо за формулою:

$$D = \sqrt{\int_a^b \big(f(x) - g(x)\big)^2 dx} \tag{13}$$

При $k = 0.051$ маємо $D_{min} = 0.105$.

РОЗДІЛ 3. ЕКСПЕРИМЕНТИ З ПОРІВНЯННЯ ФУНКЦІЙ АКТИВАЦІЇ

Було проведено ряд експериментів, що порівнюють роботу однакових нейронних мереж на декількох завданнях, де різниця у мережах для кожного окремо взятого завдання полягає у тому, які функції активації було використано у проміжних шарах. Перші дві групи експериментів порівнюють ReLU з запропонованою функцією активації SLU у двох конфігураціях: зі спільним параметром (1 параметр на шар, далі позначено *shared*) та з індивідуальними параметрами (1 параметр на нейрон, далі позначено *individual*). Далі порівнюється по 5 мереж: мережа з ReLU у якості контролю, більш "продвинені" ELU та GELU, а також зазначені вище SLU (shared) та SLU (individual).

Перші два розділи присвячені роботі з простими синтетичними даними та спрямовані на те, щоб проілюструвати на простих завданнях різницю у підходах двох функцій активації та дати інтуїтивне розуміння того, що відбувається у більш складних прикладах.

Третій розділ містить набір експериментів з класифікації рукописних цифр на датасеті MNIST (сірі зображення 28x28 в 10 класах, 60 тис. для навчання та 10 тис. для тесту). Було проведено 4 експерименти з навчання повнозв'язних нейронних мереж, що відрізняються лише кількістю нейронів та шарів. Розділ спрямований на порівняння результатів із тими, що були отримані у [4] та [5], а також на демонстрацію того, який вплив має потужність (*capacity*) мережі на динаміку навчання при кожній з використаних функцій активації.

Усі експерименти можна відтворити за допомогою побудованого командного інтерфейсу. Усі графіки в роботі побудовані цією ж програмою. Параметри, що використовувались при проведенні експериментів, встановлені за замовчуванням. Наприклад, щоб відтворити один з пунктів

першого експерименту, достатньо запустити наступну команду: `python commands.py regress_1d_root`.

Четвертий розділ містить два експерименти з класифікації кольорових зображень різних об'єктів (літаки, автомобілі, птахи, коти, олені, собаки, жаби, коні, кораблі та вантажівки) на датасеті CIFAR-10 (кольорові зображення 32x32 в 10 класах, 50 тис. для навчання та 10 тис. для тесту). Розділ спрямований на відтворення результатів, отриманих в експериментах з [5], а також на перевірку того, чи змінить вибір архітектури мережі (з повнозв'язної на згорткову) загальну картину, отриману в попередніх розділах. Ключові умови проведення оригінальних експериментів відтворені, архітектури мереж відповідають тим, що були використані у роботі.

При навчанні використовувався Adam (Adaptive Moment Estimation) – оптимізатор, який у своїй основі має градієнтний спуск. Adam є одним з найбільш ефективних алгоритмів оптимізації в навчанні нейронних мереж. Даний вибір також мотивовано тим, що автори роботи, де було запропоновано GELU [5], також використовували його в експериментах, а використання одного оптимізатора збільшує надійність результатів порівняння динамік навчання.

3.1. Апроксимація одномірних функцій

Для ілюстрації переваг нелінійної функції активації над ReLU було проведено експеримент із розв'язанням задачі одновимірної регресії — апроксимації одномірної функції на деякому проміжку. Було взято три повнозв'язні мережі з фіксованою кількістю шарів та нейронів (2 шари по 5 нейронів, а також вихідний шар з єдиним нейроном без активації). У першій мережі в якості функції активації бралась ReLU, у другій використовувалась SLU зі спільним параметром (1 параметр на шар, усього 2), у третій

використано SLU з індивідуальними параметрами (1 параметр на нейрон, тобто 10 параметрів).

У якості даних було обрано функції $y = x^2,\ y = \sqrt{x},\ y = 1/x$. При навчанні до y було додано шум по формулі:

$$y = y + 0.1 \cdot \varepsilon \cdot \text{noise}. \tag{14}$$

Значення noise було обрано рівним 0.3.

Нижче представлені результати апроксимації та динаміка навчання.

3.1.1. $y = x^2$

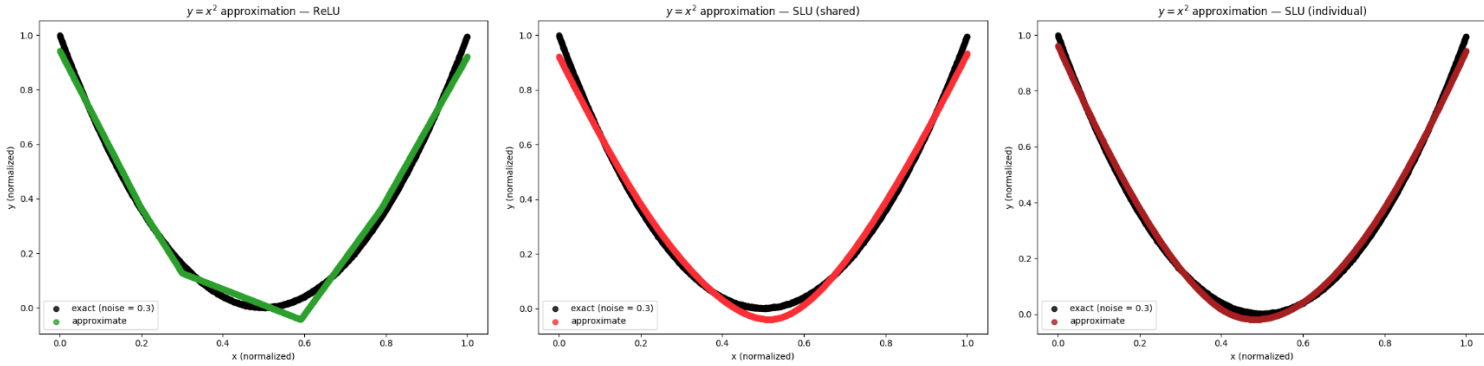

Рисунок 8. Візуальне порівняння апроксимацій.

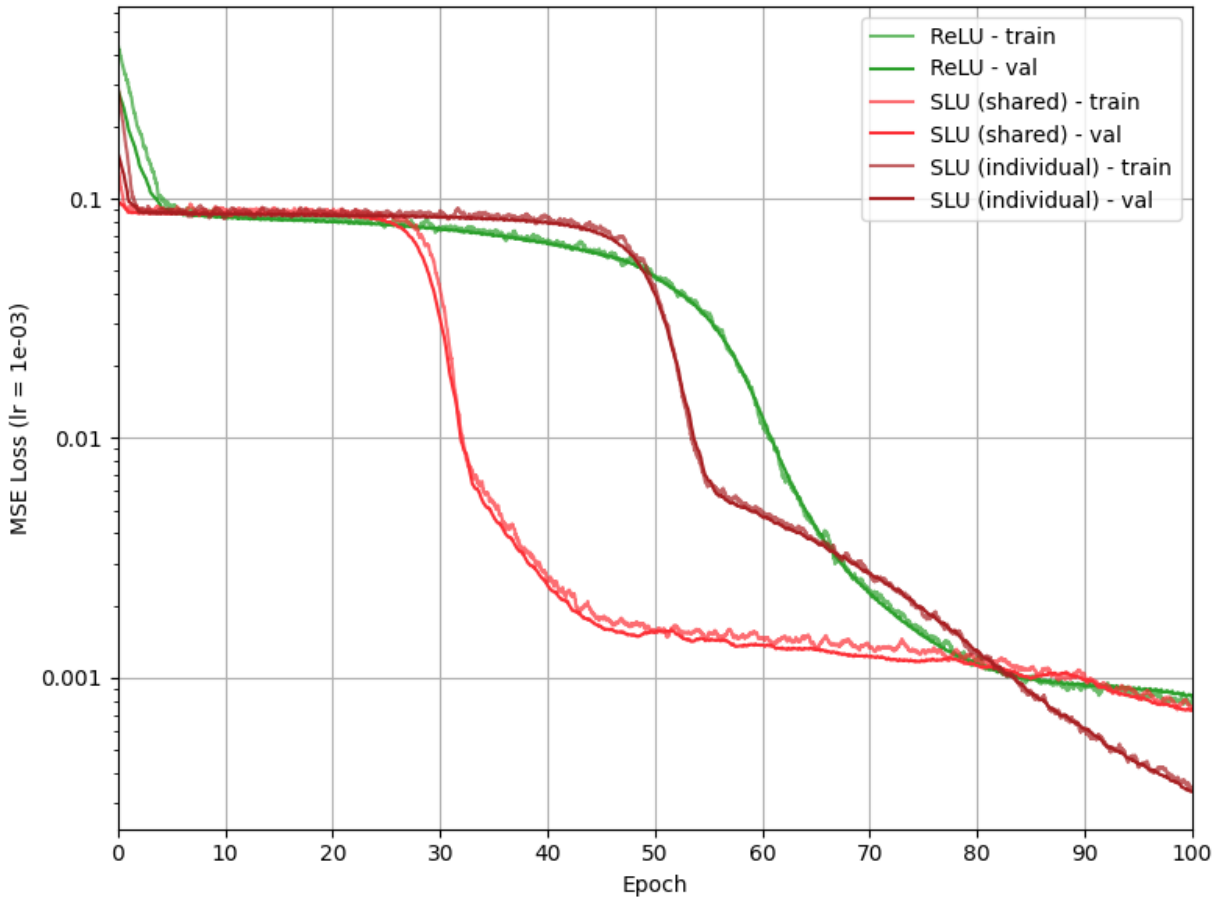

Рисунок 9. Порівняння динаміки навчання.

3.1.2. $y = \sqrt{x}$

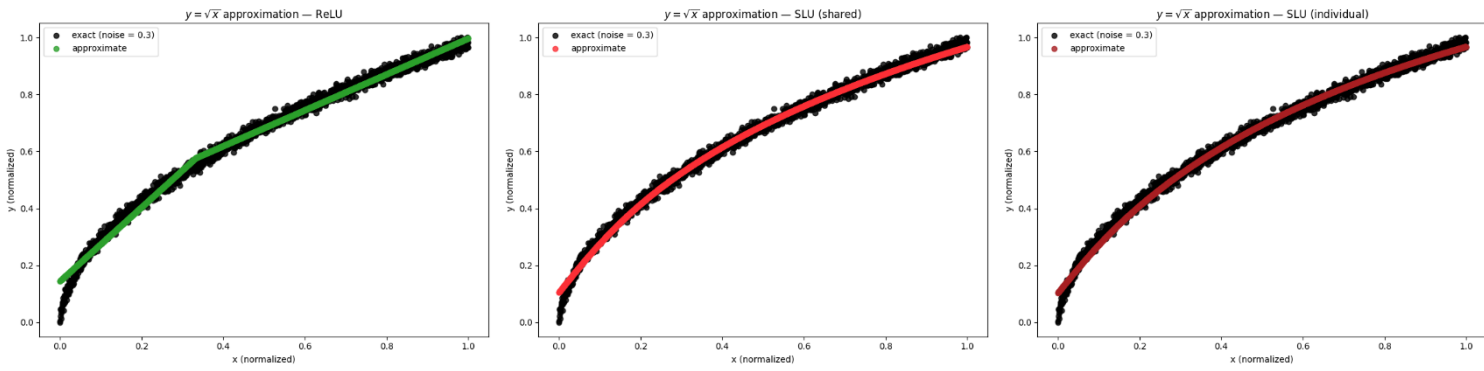

Рисунок 10. Візуальне порівняння апроксимацій.

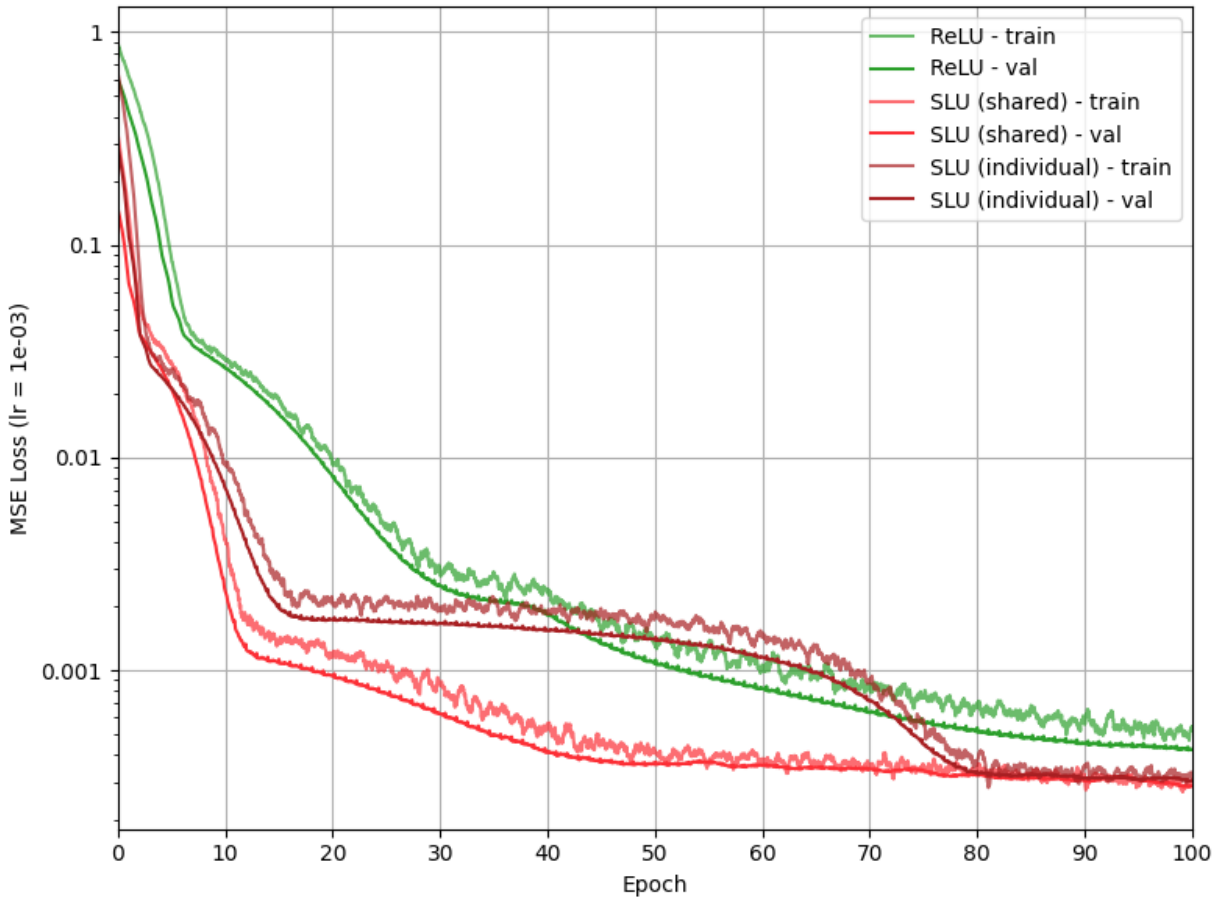

Рисунок 11. Порівняння динаміки навчання.

3.1.3. $y = 1/x$

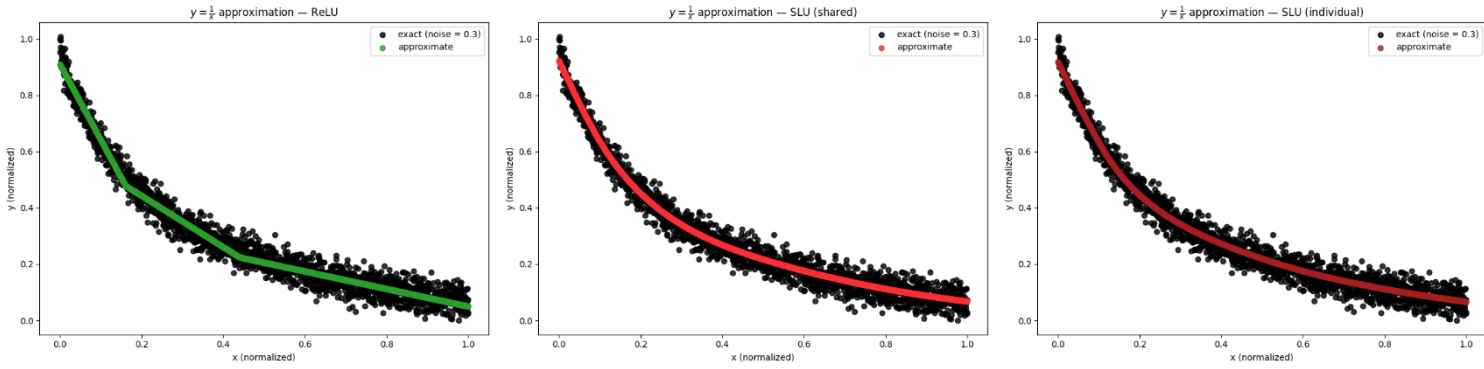

Рисунок 12. Візуальне порівняння апроксимацій.

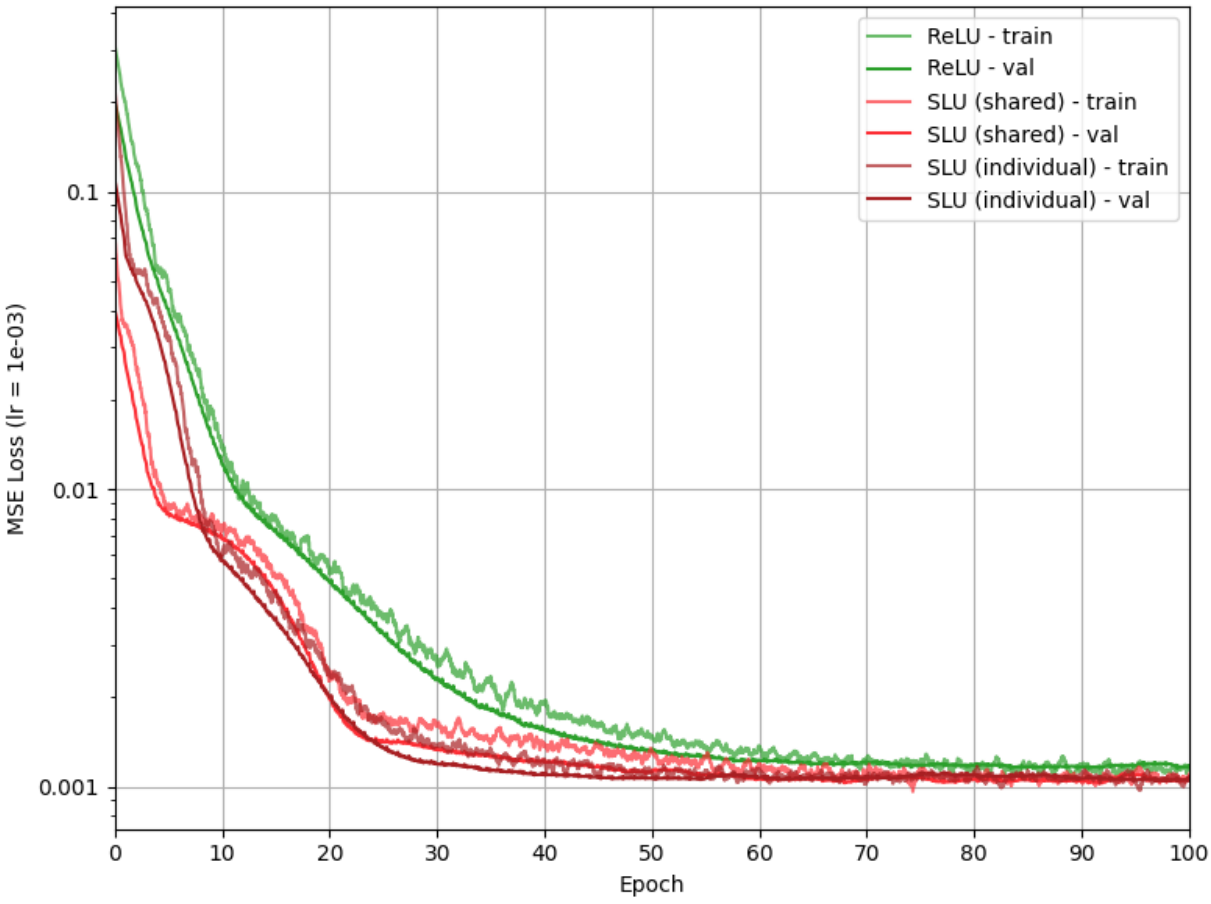

Рисунок 13. Порівняння динаміки навчання.

3.1.4. Інтерпретація результатів експериментів

Можемо зробити такі попередні висновки:

- SLU показало кращі результати для всіх трьох поставлених задач;
- Використання SLU навіть зі спільними параметрами значно збільшило здатність мережі відповідати даним (на графіках апроксимацій ламані при ReLU у випадках SLU замінені кривими, що повторюють розподіл даних);
- SLU з індивідуальними параметрами не показало значущого приросту якості в порівнянні з SLU зі спільним параметром. Це може бути пов'язано з простотою завдання (потужності другої мережі вистачило для отримання кращого результату при даному рівні шуму і третій нічого було покращувати);

- Як можна було очікувати, збільшена кількість параметрів третьої мережі іноді призводить до довшого періоду навчання в порівнянні з другою, що можна побачити на випадку $y = x^2$. Даний феномен можна пояснити тим, що з кількістю параметрів зростає вимірність простору, в якому градієнтні методи шукають оптимальний набір параметрів для роботи мережі.

3.2. Бінарна класифікація двомірних точок

Далі було проведено ряд експериментів із розв'язанням задачі бінарної класифікації — розбиття двох класів даних у двовимірному просторі. Архітектура мереж майже не змінилась з попереднього пункту, проте на вході стоїть по два нейрона замість одного (тепер дані двовимірні), а на виході, де раніше не було функції активації, додана сигмоїда (розв'язується задача бінарної класифікації, тому виход мережі повинен бути від 0 до 1, щоб результат класифікації можна було б розуміти як імовірність належності точки до одного з класів).

У якості даних було обрано два "місяці", що переплітаються (*moons*), а також дві спіралі (*spirals*). Датасет moons згенеровано за допомогою вбудованої функції `make_moons` бібліотеки scikit-learn. Датасет spirals було згенеровано наступною функцією:

```
def make_spirals(n_samples: int, noise: float) -> Tuple[np.ndarray,
np.ndarray]:
    theta = np.sqrt(np.random.rand(n_samples))*2*np.pi

    r_a = 2*theta + np.pi
    data_a = np.array([np.cos(theta)*r_a, np.sin(theta)*r_a]).T
    x_a = data_a + np.random.randn(n_samples, 2) * 3*noise

    r_b = -2*theta - np.pi
```

```
data_b = np.array([np.cos(theta)*r_b, np.sin(theta)*r_b]).T
x_b = data_b + np.random.randn(n_samples, 2) * 3*noise

res_a = np.append(x_a, np.zeros((n_samples, 1)), axis=1)
res_b = np.append(x_b, np.ones((n_samples, 1)), axis=1)

res = np.append(res_a, res_b, axis=0)
np.random.shuffle(res)

return res[:, :2], res[:, 2]
```

Значення noise було обрано 0.3 для обох датасетів.

Нижче представлені результати апроксимації та динаміка навчання.

3.2.1. Moons

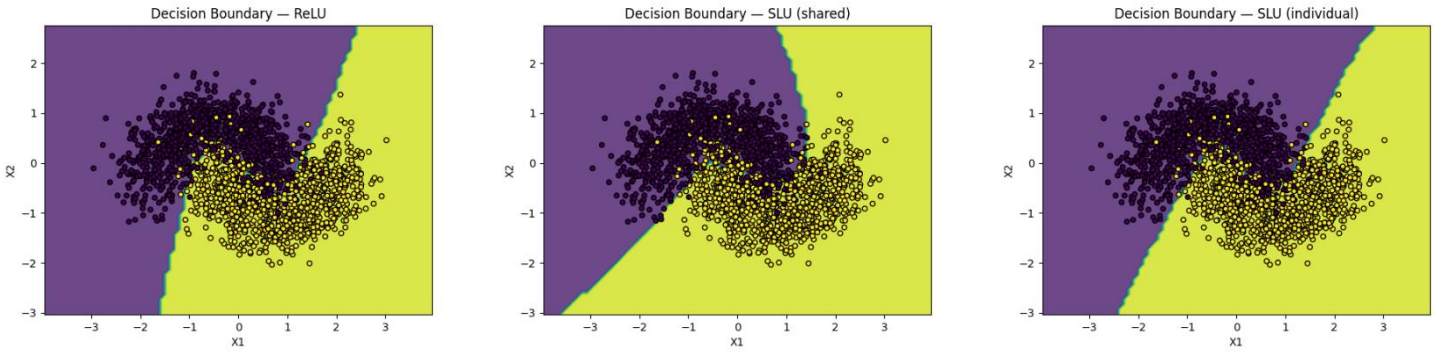

Рисунок 14. Візуальне порівняння класифікацій.

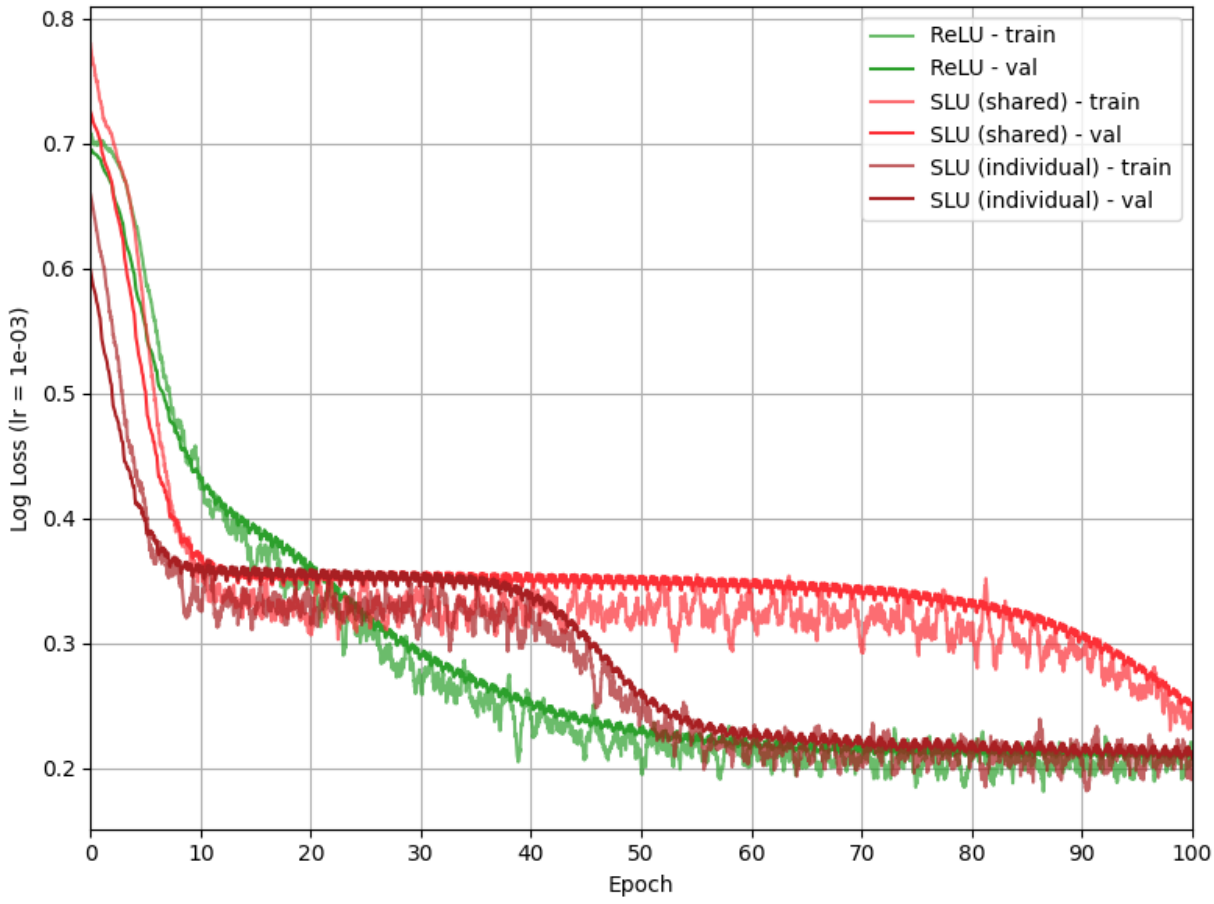

Рисунок 15. Порівняння динаміки навчання.

Усі 3 мережі доходять до одного рівня точності, проте мережі з SLU збігаються довше. Це можна пояснити тим, що класи без проблем розділяються ламаною (як можна побачити на візуалізації класифікації з ReLU, ламаної з трьох ланок насправді достатньо). Це призводить до переваги для ReLU: мережі з SLU мають додаткові параметри, підбір яких ускладнює пошук оптимального розділення на класи, проте наявність самих параметрів нічого не дає.

3.2.2. Spirals

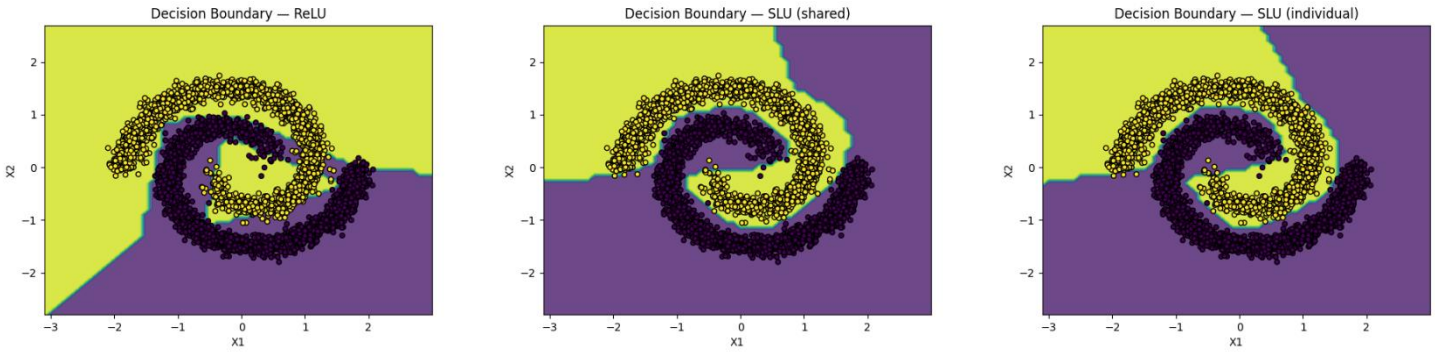

Рисунок 16. Візуальне порівняння класифікацій.

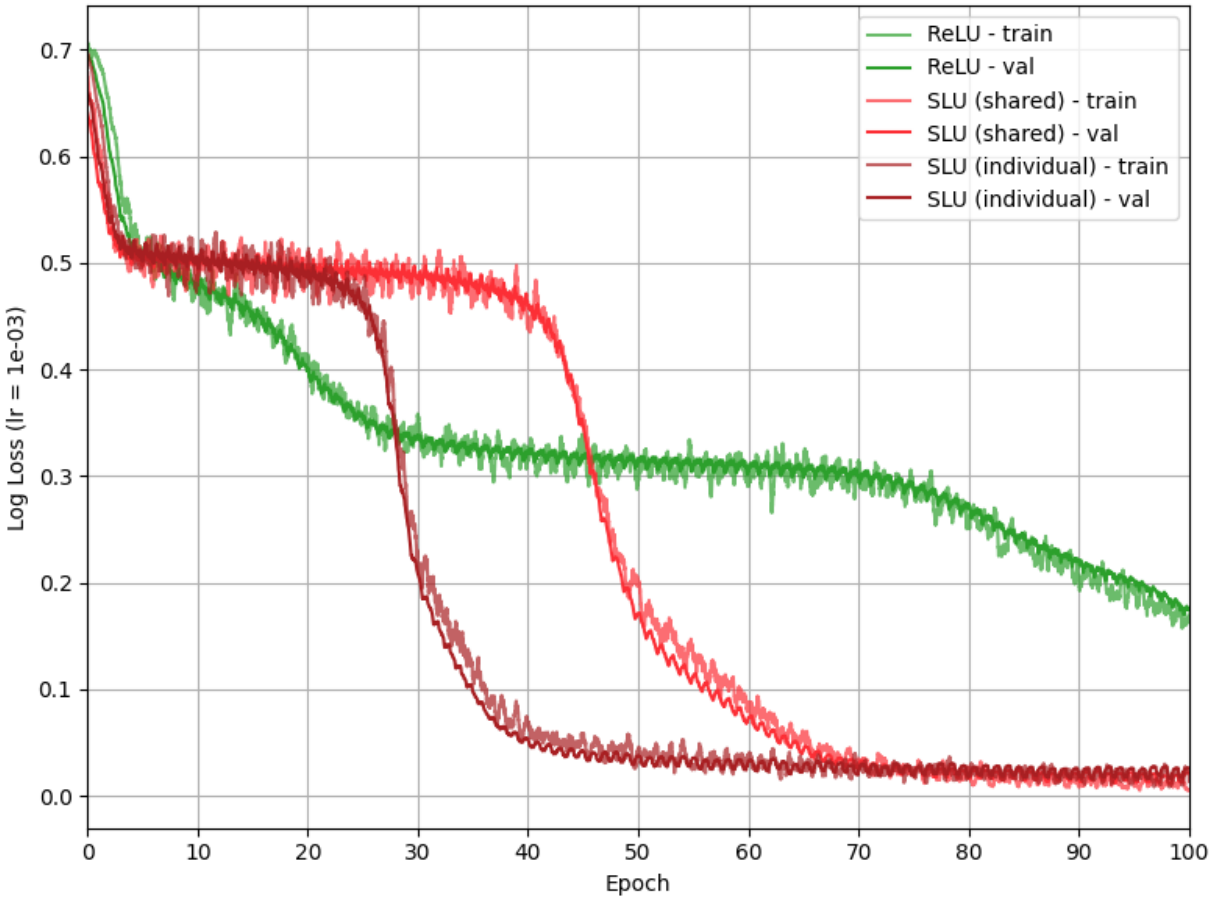

Рисунок 17. Порівняння динаміки навчання.

Тут ми бачимо зворотню ситуацію: спіралі складно розділити ламаними, а SLU показує значну перевагу в швидкості навчання. Що цікаво, в обох експериментах з класифікації модель SLU з індивідуальними

параметрами збіглася швидше, що контрастує з останнім висновком попереднього розділу. Варто пам'ятати, що кількість параметрів — це лише один з багатьох факторів, що визначають швидкість навчання, і даній темі можна було б присвятити окрему роботу.

3.3. Класифікація рукописних цифр MNIST

Наступний набір експериментів присвячений класифікації рукописних цифр на датасеті MNIST. Було проведено 4 експерименти з навчання повнозв'язних нейронних мереж з різною кількістю нейронів та шарів:

- 4 шари по 64 нейрона;
- 8 шарів по 64 нейрона;
- 4 шари по 128 нейронів;
- 8 шарів по 128 нейронів.

Ціль даних експериментів полягає в тому, щоб порівняти динаміку навчання та результуючу точність на більш складному завданні класифікації. Класів тепер 10, а "властивостей" об'єктів — 784 (по пікселю на властивість). Кількість експериментів у даному розділі пояснюється тим, що залежність результатів від глибини мережі та кількості нейронів представляє інтерес з декількох причин:

1. Збільшення кількості параметрів мережі ускладнює пошук в просторі параметрів точок, де похибка прогнозів мінімальна. Враховуючи, що запропонована функція активації має власні параметри (один параметр або на кожний шар, або на кожний нейрон), цікаво подивитись, чи буде даний факт негативно впливати на швидкість навчання.
2. Безкінечне поглиблення мережі не може призводити до безкінечного підвищення якості прогнозів, і даний датасет є чудовим прикладом

цього. MNIST традиційно використовується для перевірки доволі простих архітектур, і поставлений набір експериментів продемонструє вплив кожної з функцій активації на стійкість мереж до перенавчання.

3. Даний датасет уже використовувався у роботах [4] та [5], причому також із повнозв'язними мережами. Користуючись нагодою, ми можемо порівняти наші результати з тими, що були представлені у вказаних роботах, і подивитись, чи має різниця в імплементації значний на них вплив.

На графіках нижче можна побачити графіки похибки на навчальній та тестовій вибірках, а також мінімальні рівні похибок, досягнутих у процесі навчання, для кожної з мереж.

3.3.1. 4x64

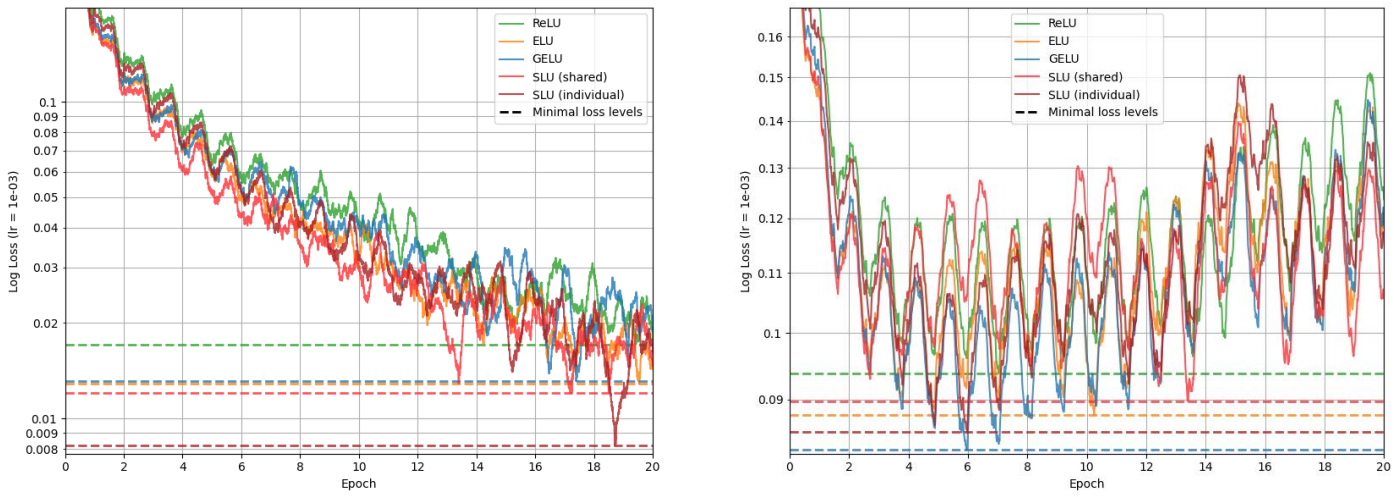

Рисунок 18. Порівняння динаміки навчання
(зліва результати на навчальній вибірці, справа на тестовій).

3.3.2. 8x64

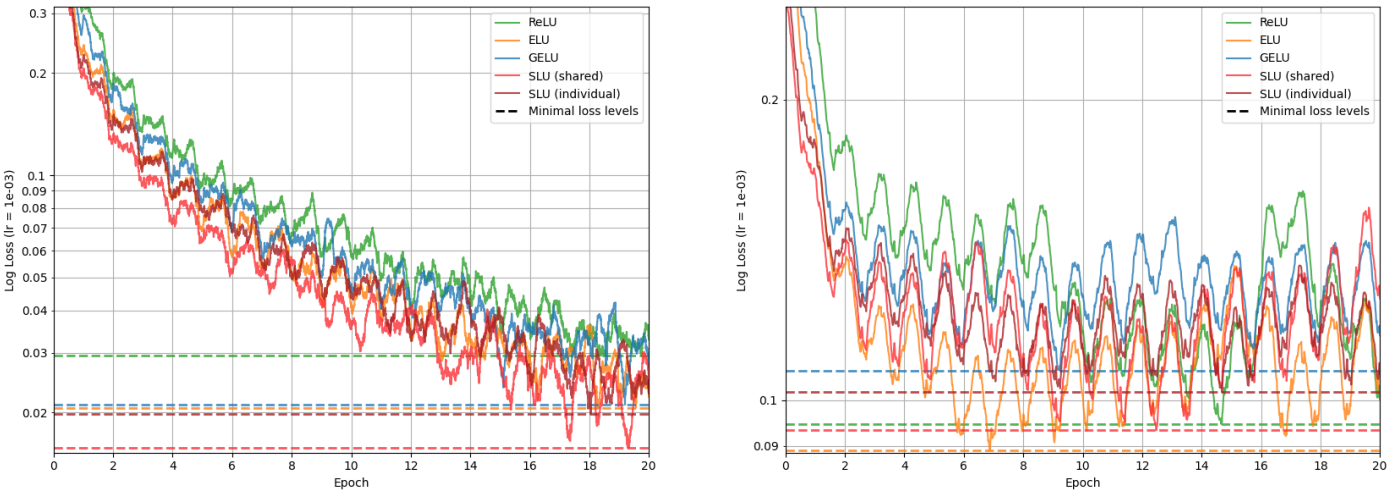

Рисунок 19. Порівняння динаміки навчання
(зліва результати на навчальній вибірці, справа на тестовій).

3.3.3. 4x128

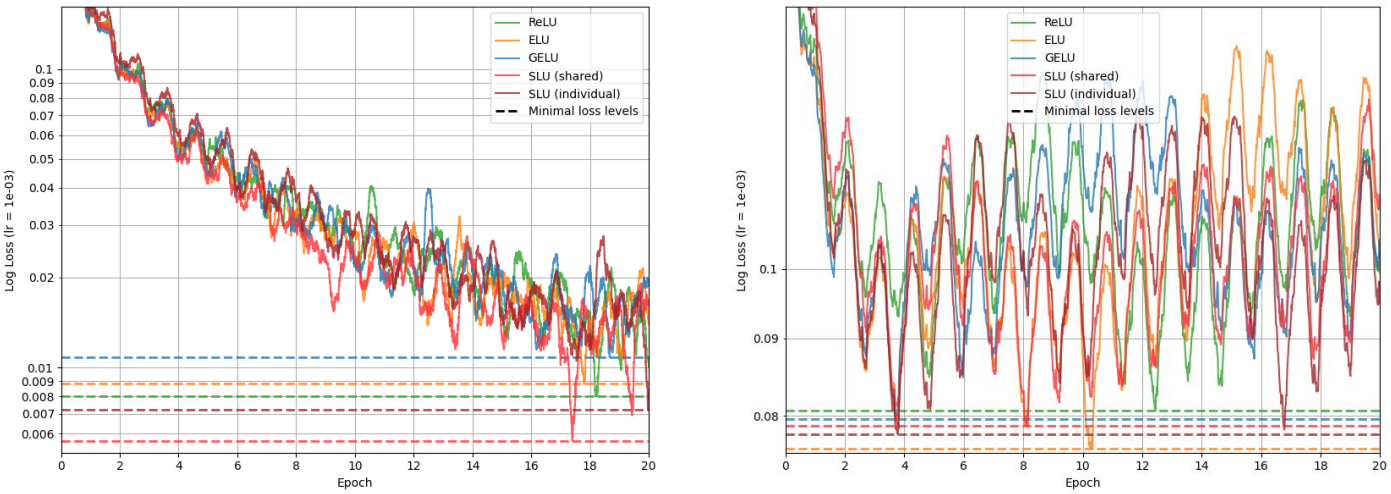

Рисунок 20. Порівняння динаміки навчання
(зліва результати на навчальній вибірці, справа на тестовій).

3.3.4. 8x128

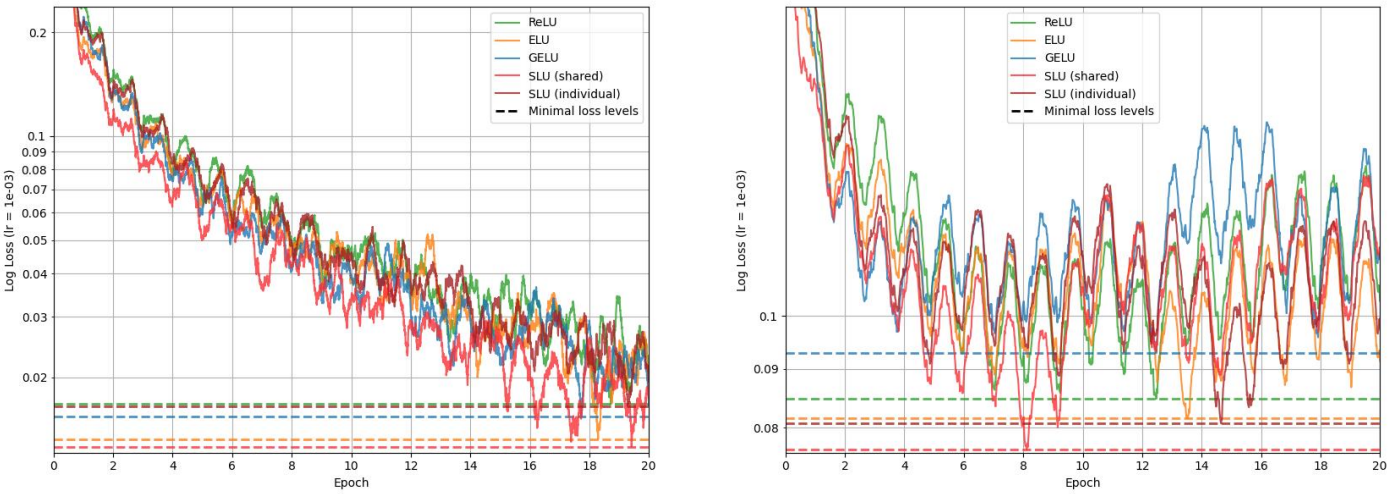

Рисунок 21. Порівняння динаміки навчання
(зліва результати на навчальній вибірці, справа на тестовій).

3.3.5. Інтерпретація результатів експериментів

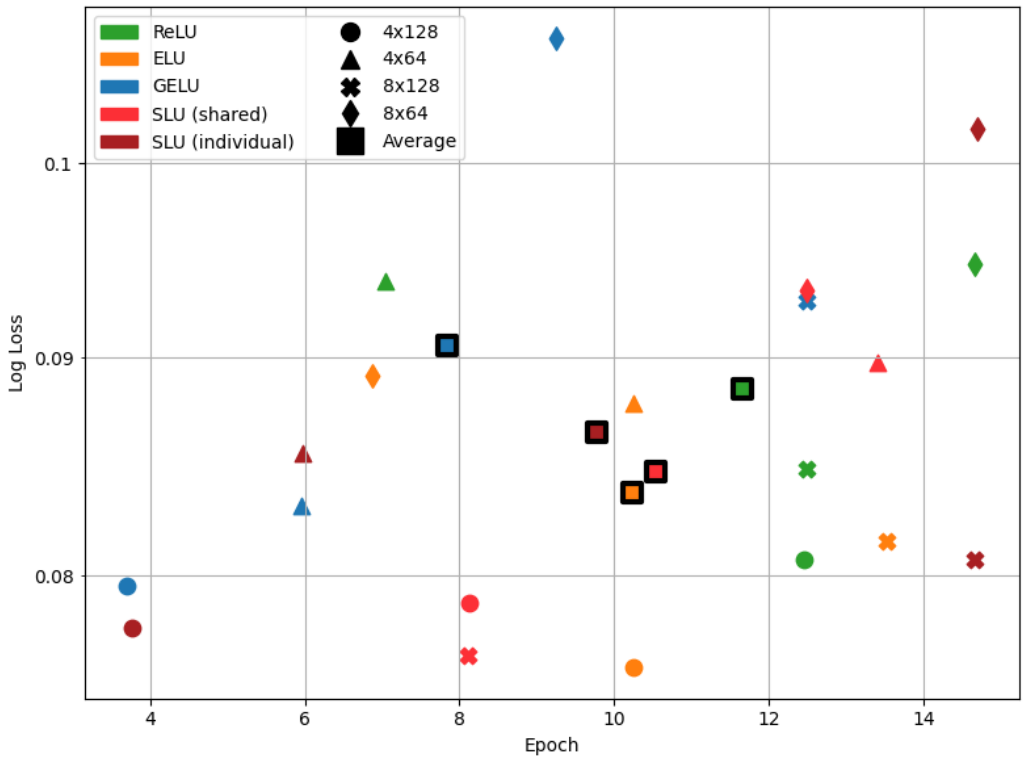

Рисунок 22. Візуалізація, на якій епосі навчання та з яким рівнем похибки збіглись мережі в кожному з експериментів.

Network (4x64)	Epoch	Best Loss	Network (8x64)	Epoch	Best Loss
ReLU	7.0385	0.0938	ReLU	14.6556	0.0947
ELU	10.2479	0.0878	ELU	6.8732	0.0891
GELU	5.9504	0.0831	GELU	9.2424	0.1070
SLU (shared)	13.4022	0.0897	SLU (shared)	12.4793	0.0933
SLU (individual)	5.9779	0.0855	SLU (individual)	14.6831	0.1018
Network (4x128)	Epoch	Best Loss	Network (8x128)	Epoch	Best Loss
ReLU	12.4517	0.0806	ReLU	12.4793	0.0847
ELU	10.2479	0.0761	ELU	13.5261	0.0815
GELU	3.7052	0.0795	GELU	12.4793	0.0928
SLU (shared)	8.1267	0.0788	SLU (shared)	8.1129	0.0766
SLU (individual)	3.7741	0.0777	SLU (individual)	14.6556	0.0806

Таблиця 1. Значення, показані на рисунку вище.

Network	Epoch	Best Loss	Network	Epoch	Best Loss
ELU 4x128	10.2479	0.0761	SLU (individual) 4x64	5.9779	0.0855
SLU (shared) 8x128	8.1129	0.0766	ELU 4x64	10.2479	0.0878
SLU (individual) 4x128	3.7741	0.0777	ELU 8x64	6.8732	0.0891
SLU (shared) 4x128	8.1267	0.0788	SLU (shared) 4x64	13.4022	0.0897
GELU 4x128	3.7052	0.0795	GELU 8x128	12.4793	0.0928
ReLU 4x128	12.4517	0.0806	SLU (shared) 8x64	12.4793	0.0933
SLU (individual) 8x128	14.6556	0.0806	ReLU 4x64	7.0385	0.0938
ELU 8x128	13.5261	0.0815	ReLU 8x64	14.6556	0.0947

GELU 4x64	5.9504	0.0831	SLU (individual) 8x64	14.6831	0.1018
ReLU 8x128	12.4793	0.0847	GELU 8x64	9.2424	0.1070

Таблиця 2. Результати навчання мереж, відсортовані за розміром похибки.

Activation function	Average Epoch	Average Loss
ELU	10.2238	0.0836
SLU (shared)	10.5303	0.0846
SLU (individual)	9.7727	0.0864
ReLU	11.6563	0.0884
GELU	7.8443	0.0906

Таблиця 3. Середні показники мереж з кожною з представлених функцій активації, відсортовані за розміром похибки.

Результати можемо підсумувати наступним чином:

- SLU показала результати, співставні з іншими функціями активації, що вказує на те, що функцію було підібрано успішно;
- Кращі результати на тестовій вибірці серед усіх мереж у даній групі експериментів показали: ELU (4x128), SLU shared (8x128), SLU individual (4x128), SLU shared (4x128), GELU (4x128). Бачимо, що за 3 кращі результати з 5 показала запропонована нами функція активації;
- Негативний ефект збільшеної кількості параметрів SLU на швидкість навчання було компенсовано: мережі з SLU навчались в середньому за 10.1515 епох, що на 0.7% швидше за ELU, на 12.9% швидше за ReLU, але на 29.4% повільніше за GELU;
- На графіках не виявлено залежності схильності мереж до перенавчання від вибору функції активації;
- GELU отримала результати, кращі за ELU, лише в одному випадку з чотирьох, що не відповідає висновкам роботи [5]. Це може свідчити про чутливість отриманих у роботі "Gaussian Error Linear Units (GELUs)" показників до імплементації та вимагає додаткового дослідження. ELU ж показала себе краще за ReLU (мережі з ELU навчались на 12.2% швидше та на 5.4% точніше) і для [4] результати зійшлись.

3.4. Класифікація кольорових об’єктів CIFAR-10

Наступний набір експериментів присвячений класифікації кольорових зображень об'єктів у датасеті CIFAR-10. Було проведено 2 експерименти з використанням різних архітектур:

- повнозв'язна мережа з 4 шарів по 64 нейрона;
- згорткова мережа (архітектуру взято з [5], додаток А).

Дана задача складніше за класифікацію MNIST, і також часто використовується при бенчмаркінгу. Експерименти спрямовані на порівняння впливів функцій активації на поведінки мереж з одним датасетом, але різними архітектурами.

На графіках нижче можна побачити графіки похибки на навчальній та тестовій вибірках, а також мінімальні рівні похибок, досягнутих у процесі навчання, для кожної з мереж.

3.4.1. FC 4x64

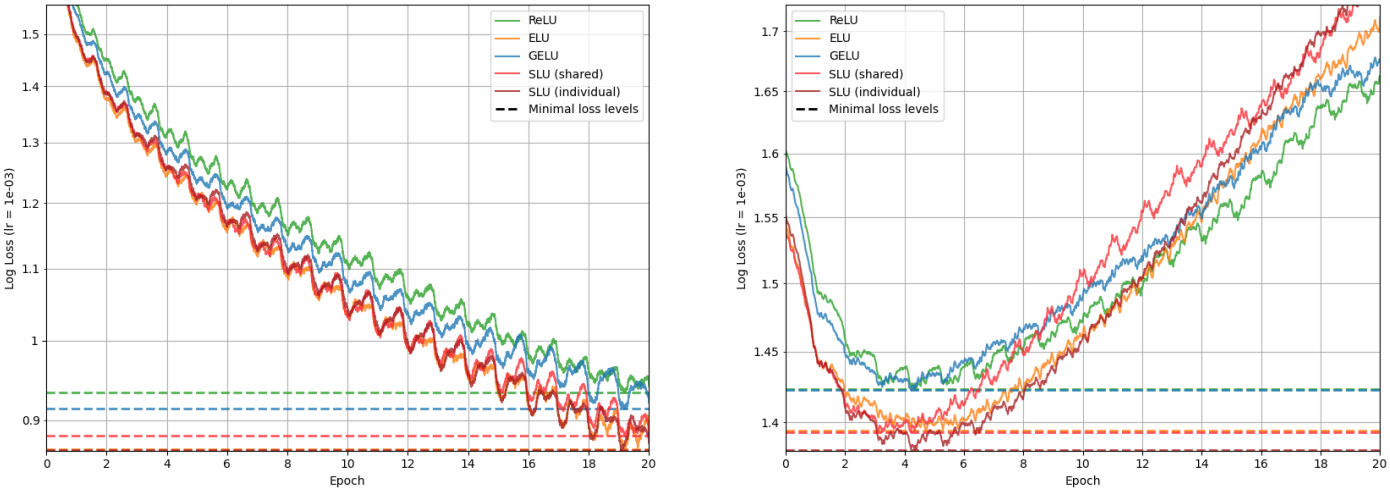

Рисунок 23. Порівняння динаміки навчання
(зліва результати на навчальній вибірці, справа на тестовій).

3.4.2. CNN

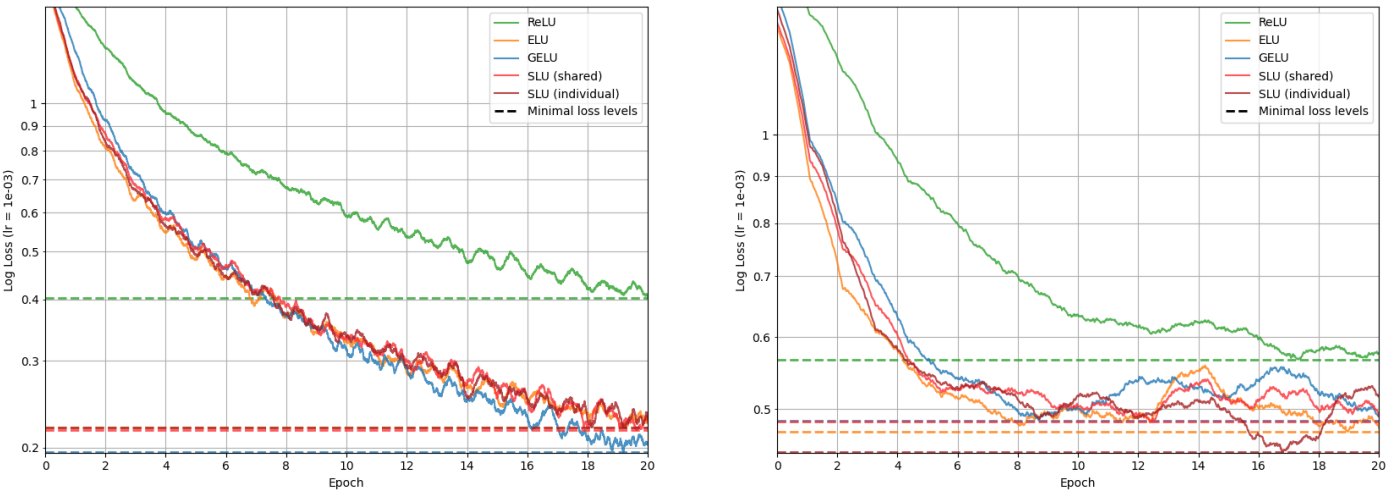

Рисунок 24. Порівняння динаміки навчання
(зліва результати на навчальній вибірці, справа на тестовій).

3.4.3. Інтерпретація результатів експериментів

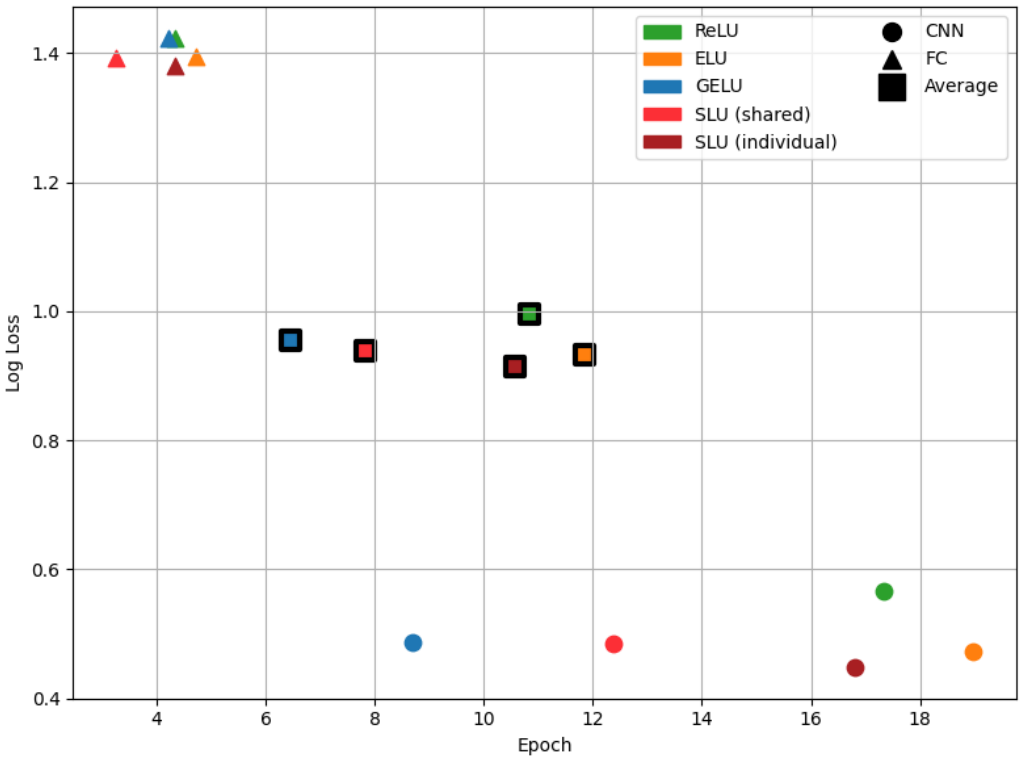

Рисунок 25. Візуалізація, на якій епосі навчання та з яким рівнем похибки збіглись мережі в кожному з експериментів.

Network (FC)	Epoch	Best Loss	Network (CNN)	Epoch	Best Loss
ReLU	4.3388	1.4231	ReLU	17.3278	0.5669
ELU	4.7245	1.3942	ELU	18.9807	0.4724
GELU	4.2148	1.4226	GELU	8.6914	0.4861
SLU (shared)	3.2506	1.3931	SLU (shared)	12.3829	0.4842
SLU (individual)	4.3388	1.3810	SLU (individual)	16.8044	0.4486

Таблиця 4. Значення, показані на рисунку вище.

Network	Epoch	Best Loss	Network	Epoch	Best Loss
SLU (individual) CNN	16.8044	0.4486	SLU (individual) FC	4.3388	1.3810
ELU CNN	18.9807	0.4724	SLU (shared) FC	3.2506	1.3931
SLU (shared) CNN	12.3829	0.4842	ELU FC	4.7245	1.3942
GELU CNN	8.6914	0.4861	GELU FC	4.2148	1.4226
ReLU CNN	17.3278	0.5669	ReLU FC	4.3388	1.4231

Таблиця 5. Результати навчання мереж, відсортовані за розміром похибки.

Activation function	Average Epoch	Average Loss
SLU (individual)	10.5716	0.9148
ELU	11.8526	0.9333
SLU (shared)	7.8168	0.9386
GELU	6.4531	0.9544
ReLU	10.8333	0.9950

Таблиця 6. Середні показники мереж з кожною з представлених функцій активації, відсортовані за розміром похибки.

Результати можемо підсумувати наступним чином:

- SLU в середньому виявилась на 17.8% швидше та на 7.4% точніше за ReLU;
- В обох експериментах GELU вийшла на трохи гірші результати за ELU, проте зробила це значно швидше (в середньому на 45.5% швидше при результатах на 2.2% гірше);
- Shared конфігурація SLU має менше параметрів, що потребують навчання, за Individual, чим пояснюється різниця у швидкості та точності (Shared виявилась на 38.6% швидше та 6% точніше за ReLU, а Individual покращила її результати на пару відсотків ціною швидкості: на 8.7% точніше за ReLU, але лише на 2.5% швидше);
- За середніми результатами точності SLU зайняла перше і третє місце, причому різниця між третім місцем і другим (ELU) становить 0.5% точності на користь ELU, проте 34% швидкості збіжності на користь SLU Shared.

ВИСНОВОК

1. Було розглянуто сучасні функції активації та порівняно їх характеристики. Написано код для відтворюваного проведення чотирьох груп експериментів. Було проаналізовано властивості кожної з функцій та співставлено з результатами експериментів.

2. Було надано огляд популярних на даний момент функцій активації: ReLU, ELU та GELU. Це включає наведення та візуалізацію самих функцій, а також перелік їх властивостей, що призвели до їх широкого використання.

3. Було зроблено узагальнення властивостей оглянутих функцій. Було підібрано власну функцію активації Smooth Logarithmic Unit (SLU), що має всі виділені властивості. У подальшому дана функція приймала участь в експериментах з порівняння наряду з іншими.

4. Було проведено дві групи експериментів з простими синтетичними даними для ілюстрації відмінностей у роботі функцій активації, групу експериментів на датасеті зображень MNIST, а також два експерименти на датасеті зображень CIFAR-10.

5. Результати експериментів загалом відповідають міркуванням з перших двох розділів. При оцінюванні ELU вдалося відтворити результати роботи [4]. GELU не показала результатів, на які вказували висновки роботи [5], що може свідчити про чутливість отриманих у роботі показників до імплементації. SLU показала результати строго краще за ReLU (на 12.9% швидше та на 3.3% точніше на MNIST, на 17.8% швидше та на 7.4% точніше на CIFAR-10) та співставні з іншими функціями активації, що вказує на те, що функцію було підібрано успішно.

СПИСОК ЛІТЕРАТУРИ

1. Nair, V. and Hinton, G. E. Rectified linear units improve restricted Boltzmann machines. In F¨urnkranz, J. and Joachims, T. (eds.), Proceedings of the 27th International Conference on Machine Learning (ICML10), pp. 807–814, 2010.
2. Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudk, M. (eds.), JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011), volume 15, pp. 315–323, 2011.
3. Lu L. Dying ReLU and Initialization: Theory and Numerical Examples. Communications in Computational Physics. 2020. Vol. 28, no. 5. P. 1683. URL: https://doi.org/10.4208/cicp.oa-2020-0165 (date of access: 04.05.2024).
4. Clevert D.-A., Unterthiner T., Hochreiter S. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). International Conference on Learning Representations. 2015. URL: https://doi.org/10.48550/arXiv.1511.07289 (date of access: 07.02.2024).
5. Hendrycks D., Gimpel K. Gaussian Error Linear Units (GELUs). 2016. URL: https://doi.org/10.48550/arXiv.1606.08415 (date of access: 07.02.2024).

ДОДАТОК А. КОД ПРОГРАМИ

commands.py

```
from pathlib import Path

import fire

from src.utils import set_global_seed
from src.visualizations import (
    PlotActivations,
    PlotSLU
)
from src.experiments import (
    Classify2D,
    Regress1D,
    ClassifyMNIST,
    SummarizeMNIST,
    ClassifyCIFAR10,
    SummarizeCIFAR10
)


set_global_seed(0)

log_dir = Path('logs')
log_dir.mkdir(exist_ok=True)

data_dir = Path('data')
data_dir.mkdir(exist_ok=True)
```

```
def plot_activations() -> None:
    PlotActivations(fig_path=log_dir/'activations.png').evaluate()
    PlotActivations(fig_path=log_dir/'sigmoid.png',
function='Sigmoid').evaluate()
    PlotActivations(fig_path=log_dir/'relu.png',
function='ReLU').evaluate()
    PlotActivations(fig_path=log_dir/'elu.png',
function='ELU').evaluate()
    PlotActivations(fig_path=log_dir/'gelu.png',
function='GELU').evaluate()


def plot_SLU() -> None:
    PlotSLU(x_min=-20, x_max=20,
fig_path=log_dir/'slu_20.png').evaluate()
    PlotSLU(x_min=-3, x_max=3,
fig_path=log_dir/'slu_3.png').evaluate()


def classify_spirals(train: bool = True) -> None:
    experiment = Classify2D(artifacts_dir=log_dir/'spirals',
dataset='spirals')
    if train: experiment.train()
    experiment.plot()


def classify_moons(train: bool = True) -> None:
    experiment = Classify2D(artifacts_dir=log_dir/'moons',
dataset='moons')
    if train: experiment.train()
    experiment.plot()
```

```
def regress_1d_square(train: bool = True) -> None:
    experiment = Regress1D(artifacts_dir=log_dir/'square',
dataset='square')
    if train: experiment.train()
    experiment.plot()


def regress_1d_root(train: bool = True) -> None:
    experiment = Regress1D(artifacts_dir=log_dir/'root',
dataset='root')
    if train: experiment.train()
    experiment.plot()


def regress_1d_reciprocal(train: bool = True) -> None:
    experiment = Regress1D(artifacts_dir=log_dir/'reciprocal',
dataset='reciprocal')
    if train: experiment.train()
    experiment.plot()


def classify_mnist_4x64(train: bool = True, force_retrain = True) ->
None:
    experiment = ClassifyMNIST(artifacts_dir=log_dir/'mnist'/'4x64',
force_retrain=force_retrain,
                               n_layers=4, n_neurons=64)
    if train: experiment.train()
    experiment.plot()


def classify_mnist_4x128(train: bool = True, force_retrain = True) ->
None:
```

```
    experiment = ClassifyMNIST(artifacts_dir=log_dir/'mnist'/'4x128',
force_retrain=force_retrain,
                               n_layers=4, n_neurons=128)
    if train: experiment.train()
    experiment.plot()


def classify_mnist_8x64(train: bool = True, force_retrain = True) ->
None:
    experiment = ClassifyMNIST(artifacts_dir=log_dir/'mnist'/'8x64',
force_retrain=force_retrain,
                               n_layers=8, n_neurons=64)
    if train: experiment.train()
    experiment.plot()


def classify_mnist_8x128(train: bool = True, force_retrain = True) ->
None:
    experiment = ClassifyMNIST(artifacts_dir=log_dir/'mnist'/'8x128',
force_retrain=force_retrain,
                               n_layers=8, n_neurons=128)
    if train: experiment.train()
    experiment.plot()


def summarize_mnist() -> None:
    SummarizeMNIST(artifacts_dir=log_dir/'mnist').plot()


def classify_cifar10_fc(train: bool = True, force_retrain = True) ->
None:
    experiment = ClassifyCIFAR10(artifacts_dir=log_dir/'cifar-
10'/'FC', force_retrain=force_retrain,
```

```
                                    network_type='FC')
    if train: experiment.train()
    experiment.plot()


def classify_cifar10_cnn(train: bool = True, force_retrain = True) ->
None:
    experiment = ClassifyCIFAR10(artifacts_dir=log_dir/'cifar-
10'/'CNN', force_retrain=force_retrain,
                                    network_type='CNN')
    if train: experiment.train()
    experiment.plot()


def summarize_cifar10() -> None:
    SummarizeCIFAR10(artifacts_dir=log_dir/'cifar-10').plot()


if __name__ == '__main__':
    fire.Fire()
```

pyproject.toml

```
[tool.poetry]
name = "activation functions comparison"
version = "0.1.0"
description = "A set of experiments aimed to evaluate modern
activation functions in various tasks"
authors = ["Ivan Lytvynenko <lytvynenko.i.d@gmail.com>"]
license = "GPL-3.0"
readme = "README.md"
```

```
[tool.poetry.dependencies]
python = "^3.9"
torch = "^2.2.1"
numpy = "^1.26.4"
matplotlib = "^3.8.3"
scikit-learn = "^1.4.1.post1"
torchvision = "^0.17.1"
fire = "^0.5.0"

[tool.black]
line-length = 90
target-version = ["py311"]

[tool.isort]
src_paths = ["src"]
profile = "black"
line_length = 90
lines_after_imports = 2

[tool.pyright]
venvPath = "."
venv = ".venv"

[tool.flake8]
max-line-length = 90
ignore = ['E203', 'E501', 'W503', 'B950']
max-complexity = 12
select = ['B', 'C', 'E', 'F', 'W', 'B9']
per-file-ignores = [
    # for easier imports to __init__ without __all__
    '**/__init__.py: F401',
    # file to define custom types
    'types.py: F401',
```

```
]
count = true

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

.gitignore

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# Poetry venv folder
.venv

data/*
logs/*
```

src\colors.py

```
COLORS = {
    # general
    'ReLU': '#2CA02C',
    'ELU': '#FF7F0E',
    'GELU': '#1F77B4',
    'SLU': '#D62728',

    # performance comparison
    'SLU (shared)': '#FC2F36',
```

```
    'SLU (individual)': '#A81F22',

    # activation plots
    'SLU (k = -0.2)': '#FC2F36',
    'SLU (k = 0)': '#D62728',
    'SLU (k = 0.2)': '#A81F22',

    # other
    'Sigmoid': '#17BECF',
}
```

src\layers.py

```
import torch
import torch.nn as nn


class SLU(nn.Module):
    def __init__(self, num_parameters=1):
        super(SLU, self).__init__()
        self.k = nn.Parameter(torch.full((num_parameters,), 0.0))

    def forward(self, x):
        A = torch.log(1 + torch.abs(x))
        B = self.k * A.pow(2)
        return torch.where(x > 0, x + B, B - A)
```

src\types.py

```
from pathlib import Path
```

```
from typing import Callable, ClassVar, Iterable, List, Literal,
Optional, Tuple, Type, Union

from torch import device


File = Path
Directory = Path
Device = Union[str, device]


__all__ = (
    'File',
    'Directory',
    'Device',
    'Callable',
    'ClassVar',
    'Iterable',
    'List',
    'Literal',
    'Optional',
    'Tuple',
    'Type',
    'Union',
)
```

src\utils.py

```
import time
import random

import numpy as np
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data

from .types import Union, Optional, Device, Tuple, List


def smooth_data(data: np.ndarray, window_size: int) -> np.ndarray:
    return np.convolve(data, np.ones(window_size)/window_size,
mode='valid')


def to_scientific_notation(num: Union[int, float]) -> str:
    str_num = str(num)
    if 'e' in str_num:
        return str_num
    str_num = f'{num:f}'
    int_part, fractional_part = str_num.split('.')
    precision = len(fractional_part.strip('0')) + len(int_part) - 1
    if int_part == '0':
        precision -= 1
    return f'{num:.{precision}e}'


def set_global_seed(seed: int) -> None:
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)


def train(
    model: nn.Module,
```

```
    train_loader: data.DataLoader,
    val_loader: data.DataLoader,
    criterion: nn.modules.loss._Loss,
    lr: float = 1e-3,
    n_epochs: int = 50,
    is_verbose: bool = False,
    device: Optional[Device] = None
) -> Tuple[List, List]:
    device = torch.device(device or torch.device('cuda' if
torch.cuda.is_available() else 'cpu'))
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    train_loss_log = []
    val_loss_log = []

    for epoch in range(1, n_epochs + 1):
        start_time = time.time()
        ep_train_loss_log = []
        ep_val_loss_log = []
        for train_batch_i, (train_inputs, train_targets) in
enumerate(train_loader):
            train_inputs = train_inputs.to(device)
            train_targets = train_targets.to(device)
            model.train(True)
            optimizer.zero_grad()
            train_outputs = model(train_inputs)

            if isinstance(criterion, (nn.BCELoss,
nn.BCEWithLogitsLoss)):
                train_loss = criterion(train_outputs,
train_targets.unsqueeze(1))
            else:
                train_loss = criterion(train_outputs, train_targets)
```

```
            train_loss.backward()
            optimizer.step()

            if train_batch_i == int(len(train_loader) / 2):
                model.train(False)
                for val_inputs, val_targets in val_loader:
                    val_inputs = val_inputs.to(device)
                    val_targets = val_targets.to(device)
                    val_outputs = model(val_inputs)
                    if isinstance(criterion, (nn.BCELoss,
nn.BCEWithLogitsLoss)):
                        val_loss = criterion(val_outputs,
val_targets.unsqueeze(1))
                    else:
                        val_loss = criterion(val_outputs, val_targets)

                    ep_val_loss_log.append(val_loss.item())
                    val_loss_log.append(val_loss.item())

            ep_train_loss_log.append(train_loss.item())
            train_loss_log.append(train_loss.item())

        if is_verbose:
            print(f"Epoch {epoch} of {n_epochs} took {time.time() -
start_time:.3f}s")
            print(f"\t  learning rate:
{optimizer.param_groups[0]['lr']:.2e}")
            print(f"\t  training loss:
{np.mean(ep_train_loss_log):.4f}")
            print(f"\tvalidation loss:
{np.mean(ep_val_loss_log):.4f}")
```

```
    return train_loss_log, val_loss_log
```

src__init__.py

```
__version__ = '0.1.0'
```

src\experiments\base.py

```
from abc import ABC, abstractmethod

import torch
import torch.utils.data as data
from sklearn.model_selection import train_test_split

from ..types import Directory, Tuple, Type


class BasePreprocessor(ABC):
    def __init__(self):
        self.is_trained = False

    @abstractmethod
    def fit(self, X: torch.Tensor, y: torch.Tensor) -> None:
        ...

    @abstractmethod
    def transform(self, X: torch.Tensor, y: torch.Tensor) ->
Tuple[torch.Tensor, torch.Tensor]:
        ...
```

```
class BaseExperiment(ABC):
    def __init__(self, artifacts_dir: Directory, Preprocessor:
Type[BasePreprocessor]):
        self.artifacts_dir = artifacts_dir
        self.preprocessor = Preprocessor()
        self.artifacts_dir.mkdir(parents=True, exist_ok=True)

    @abstractmethod
    def train(self) -> None:
        ...

    @abstractmethod
    def plot(self) -> None:
        ...

    def get_data_loaders(self, X: torch.Tensor, y: torch.Tensor,
batch_size: int = 32)\
        -> Tuple[data.DataLoader, data.DataLoader]:
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

        self.preprocessor.fit(X_train, y_train)
        X_train, y_train = self.preprocessor.transform(X_train,
y_train)
        X_test, y_test = self.preprocessor.transform(X_test, y_test)
        self.artifacts_dir.mkdir(exist_ok=True)
        torch.save(self.preprocessor,
self.artifacts_dir/'preprocessor.pth')

        train_dataset = data.TensorDataset(X_train, y_train)
        test_dataset = data.TensorDataset(X_test, y_test)
        train_loader = data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
```

```
        val_loader = data.DataLoader(test_dataset,
batch_size=batch_size)

        return train_loader, val_loader
```

src\experiments\classification_2d.py

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from matplotlib.ticker import MaxNLocator
from sklearn.datasets import make_moons

from ..colors import COLORS
from ..layers import SLU
from ..types import Callable, Directory, Tuple
from ..utils import smooth_data, to_scientific_notation, train
from .base import BaseExperiment, BasePreprocessor


class Preprocessor(BasePreprocessor):
    def __init__(self):
        super().__init__()

    def fit(self, X: torch.Tensor, y: torch.Tensor) -> None:
        self.mean = torch.mean(X)
        self.std = torch.std(X)
        self.is_trained = True

    def transform(self, X: torch.Tensor, y: torch.Tensor) ->
Tuple[torch.Tensor, torch.Tensor]:
```

```
        return (X - self.mean) / self.std, y


class Classify2D(BaseExperiment):
    datasets = {'spirals', 'moons'}

    def __init__(self, artifacts_dir: Directory, dataset: str,
                 n_samples: int = 2000, noise: float = .3,
                 n_epochs: int = 100, lr: float = 1e-3,
                 is_verbose: bool = True, window_size: int = 40):
        super().__init__(artifacts_dir, Preprocessor)
        assert dataset in Classify2D.datasets
        self.dataset = dataset
        self.n_samples = n_samples
        self.noise = noise
        self.n_epochs = n_epochs
        self.lr = lr
        self.is_verbose = is_verbose
        self.window_size = window_size

    @staticmethod
    def make_spirals(n_samples: int, noise: float) ->
Tuple[np.ndarray, np.ndarray]:
        theta = np.sqrt(np.random.rand(n_samples))*2*np.pi

        r_a = 2*theta + np.pi
        data_a = np.array([np.cos(theta)*r_a, np.sin(theta)*r_a]).T
        x_a = data_a + np.random.randn(n_samples, 2) * 3*noise

        r_b = -2*theta - np.pi
        data_b = np.array([np.cos(theta)*r_b, np.sin(theta)*r_b]).T
        x_b = data_b + np.random.randn(n_samples, 2) * 3*noise
```

```
        res_a = np.append(x_a, np.zeros((n_samples, 1)), axis=1)
        res_b = np.append(x_b, np.ones((n_samples, 1)), axis=1)

        res = np.append(res_a, res_b, axis=0)
        np.random.shuffle(res)

        return res[:, :2], res[:, 2]

    @staticmethod
    def init_network(activation_fn: Callable[[int], nn.Module]) ->
nn.Sequential:
        return nn.Sequential(
            nn.Linear(2, 5),
            activation_fn(5),
            nn.Linear(5, 5),
            activation_fn(5),
            nn.Linear(5, 1),
            nn.Sigmoid()
        )

    def train(self) -> None:
        if self.dataset == 'spirals':
            X, y = Classify2D.make_spirals(
                n_samples=self.n_samples,
                noise=self.noise
            )
        else:
            X, y = make_moons(
                n_samples=self.n_samples,
                noise=self.noise
            )
        X = torch.tensor(X, dtype=torch.float32)
        y = torch.tensor(y, dtype=torch.float32)
```

```
        train_loader, val_loader = self.get_data_loaders(X, y)

        relu_net = Classify2D.init_network(lambda _: nn.ReLU())
        slu_shared_net = Classify2D.init_network(lambda _: SLU())
        slu_ind_net = Classify2D.init_network(lambda size: SLU(size))

        self.nets = {
            'ReLU': relu_net,
            'SLU (shared)': slu_shared_net,
            'SLU (individual)': slu_ind_net,
        }

        for net_name, net in self.nets.items():
            train_loss_log, val_loss_log = train(
                model=net,
                train_loader=train_loader,
                val_loader=val_loader,
                criterion=nn.BCELoss(),
                lr=self.lr,
                n_epochs=self.n_epochs,
                is_verbose=self.is_verbose
            )
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            subfolder_name.mkdir(exist_ok=True)
            torch.save(train_loss_log,
subfolder_name/'train_loss.pth')
            torch.save(val_loss_log, subfolder_name/'val_loss.pth')

        torch.save(self.nets, self.artifacts_dir/'nets.pth')

    def plot_classification(self) -> None:
        if self.dataset == 'spirals':
```

```
    X, y = Classify2D.make_spirals(
        n_samples=self.n_samples,
        noise=self.noise
    )
else:
    X, y = make_moons(
        n_samples=self.n_samples,
        noise=self.noise
    )
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)
if not self.preprocessor.is_trained:
    self.preprocessor = torch.load(
        self.artifacts_dir/'preprocessor.pth')
X, y = self.preprocessor.transform(X, y)
X = X.numpy()
y = y.numpy()
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
grid_tensor = torch.tensor(
    np.c_[xx.ravel(), yy.ravel()], dtype=torch.float32)

if not hasattr(self, 'nets'):
    self.nets = torch.load(self.artifacts_dir/'nets.pth')

for net_name, net in self.nets.items():
    with torch.no_grad():
        Z = net(grid_tensor)
        Z = torch.round(Z).numpy().reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8)
```

```
            plt.scatter(X[:, 0], X[:, 1], c=y, s=20, edgecolors='k')
            plt.xlabel('X1')
            plt.ylabel('X2')
            plt.title(f'Decision Boundary — {net_name}')
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            subfolder_name.mkdir(exist_ok=True)
            plt.savefig(subfolder_name/'classification.png')
            plt.clf()

    def plot_loss(self) -> None:
        _, ax = plt.subplots(figsize=(8, 6))
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        for net_name in ['ReLU', 'SLU (shared)', 'SLU (individual)']:
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            train_loss = torch.load(subfolder_name/'train_loss.pth')
            train_loss = smooth_data(train_loss, self.window_size)
            x_train = np.linspace(0, self.n_epochs, len(train_loss))
            plt.plot(x_train, train_loss,
                     c=COLORS[net_name], label=f'{net_name} - train',
alpha=0.7)
            val_loss = torch.load(subfolder_name/'val_loss.pth')
            val_loss = smooth_data(val_loss, self.window_size)
            x_val = np.linspace(0, self.n_epochs, len(val_loss))
            plt.plot(x_val, val_loss,
                     c=COLORS[net_name], label=f'{net_name} - val')
        plt.xlim([0, self.n_epochs])
        plt.xlabel('Epoch')
        plt.ylabel(f'Log Loss (lr =
{to_scientific_notation(self.lr)})')
        plt.grid()
        plt.legend()
```

```
        plt.tight_layout()
        plt.savefig(self.artifacts_dir/'train_comparison.png')
        plt.clf()

    def plot(self) -> None:
        self.plot_classification()
        self.plot_loss()
```

src\experiments\classification_cifar10.py

```
from pathlib import Path

import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.lines as mlines
from matplotlib.ticker import FormatStrFormatter, MaxNLocator
import torch.utils.data as data
from torchvision import datasets, transforms

from ..colors import COLORS
from ..layers import SLU
from ..types import Callable, Directory, Tuple, Literal
from ..utils import smooth_data, to_scientific_notation, train
from .base import BaseExperiment, BasePreprocessor


class Preprocessor(BasePreprocessor):
    '''NOTE:
    No preprocessor is needed
```

```
    as pytorch has a prebuilt CIFAR10 integration
    with torchvision transforms support
    '''
    def __init__(self):
        super().__init__()

    def fit(self, X: torch.Tensor, y: torch.Tensor) -> None:
        pass

    def transform(self, X: torch.Tensor, y: torch.Tensor) ->
Tuple[torch.Tensor, torch.Tensor]:
        return X, y


class ClassifyCIFAR10(BaseExperiment):
    def __init__(self, artifacts_dir: Directory,
                 force_retrain: bool,
                 network_type: Literal['FC', 'CNN'],
                 n_layers: int = 4, n_neurons: int = 64,
                 noise: float = 0,
                 n_epochs: int = 20, lr: float = 1e-3,
                 is_verbose: bool = True, window_size: int = 128):
        super().__init__(artifacts_dir, Preprocessor)
        self.n_layers = n_layers
        self.n_neurons = n_neurons
        self.force_retrain = force_retrain
        self.network_type = network_type
        self.noise = noise
        self.n_epochs = n_epochs
        self.lr = lr
        self.is_verbose = is_verbose
        self.window_size = window_size
```

```
    def init_network(self, activation_fn: Callable[[int], nn.Module])
-> nn.Sequential:
        if self.network_type == 'FC':
            layers = [
                nn.Linear(3*32*32, self.n_neurons),
                activation_fn(self.n_neurons)
            ]
            for _ in range(self.n_layers - 1):
                layers.append(nn.Linear(self.n_neurons,
self.n_neurons))
                layers.append(activation_fn(self.n_neurons))
            layers.append(nn.Linear(self.n_neurons, 10))
            return nn.Sequential(*layers)
        else:
            return nn.Sequential(
                nn.Conv2d(3, 96, kernel_size=3, padding=1),
                activation_fn(32),
                nn.Conv2d(96, 96, kernel_size=3, padding=1),
                activation_fn(32),
                nn.Conv2d(96, 96, kernel_size=3, padding=1),
                activation_fn(32),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Dropout(p=0.5),

                nn.Conv2d(96, 192, kernel_size=3, padding=1),
                activation_fn(16),
                nn.Conv2d(192, 192, kernel_size=3, padding=1),
                activation_fn(16),
                nn.Conv2d(192, 192, kernel_size=3, padding=1),
                activation_fn(16),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Dropout(p=0.5),
```

```
                nn.Conv2d(192, 192, kernel_size=3),
                activation_fn(6),
                nn.Conv2d(192, 192, kernel_size=1),
                activation_fn(6),
                nn.Conv2d(192, 192, kernel_size=1),
                activation_fn(6),
                nn.AvgPool2d(kernel_size=6),
                nn.Flatten(),
                nn.Linear(192, 10),
            )

    def train(self) -> None:
        transformations = [
            transforms.ToTensor(),
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023,
0.1994, 0.2010)),
            transforms.Lambda(lambda x: x + torch.randn(x.shape) *
self.noise)
        ]
        if self.network_type == 'FC':
            transformations.append(transforms.Lambda(torch.flatten))
        transform = transforms.Compose(transformations)

        train_dataset = datasets.CIFAR10(root='data', train=True,
download=True, transform=transform)
        val_dataset = datasets.CIFAR10(root='data', train=False,
download=True, transform=transform)
        train_loader = data.DataLoader(train_dataset, batch_size=128)
        val_loader = data.DataLoader(val_dataset, batch_size=128)

        relu_net = self.init_network(lambda _: nn.ReLU())
        elu_net = self.init_network(lambda _: nn.ELU())
        gelu_net = self.init_network(lambda _: nn.GELU())
```

```
        slu_shared_net = self.init_network(lambda _: SLU())
        slu_ind_net = self.init_network(lambda size: SLU(size))

        self.nets = {
            'ReLU': relu_net,
            'ELU': elu_net,
            'GELU': gelu_net,
            'SLU (shared)': slu_shared_net,
            'SLU (individual)': slu_ind_net,
        }

        for net_name, net in self.nets.items():
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            if not self.force_retrain and subfolder_name.exists():
                if self.is_verbose:
                    print(f'{net_name} is pretrained, moving on')
                continue

            if self.is_verbose:
                print(f'{net_name} training started')

            train_loss_log, val_loss_log = train(
                model=net,
                train_loader=train_loader,
                val_loader=val_loader,
                criterion=nn.CrossEntropyLoss(),
                lr=self.lr,
                n_epochs=self.n_epochs,
                is_verbose=self.is_verbose
            )

            subfolder_name.mkdir(exist_ok=True)
```

```
            torch.save(train_loss_log,
subfolder_name/'train_loss.pth')
            torch.save(val_loss_log, subfolder_name/'val_loss.pth')
            torch.save(net, subfolder_name/'net.pth')

            if self.is_verbose:
                print(f'{net_name} training finished')

    def plot_loss(self) -> None:
        for group in ['train', 'val']:
            _, ax = plt.subplots(figsize=(8, 6))
            y_min, y_max = np.inf, -np.inf
            for net_name in ['ReLU', 'ELU', 'GELU', 'SLU (shared)',
'SLU (individual)']:
                subfolder_name =
self.artifacts_dir/(net_name.replace(' ', '_'))
                loss = torch.load(subfolder_name/f'{group}_loss.pth')
                loss = smooth_data(loss, self.window_size)

                y_min = min(y_min, min(loss))
                y_max = max(y_max, np.percentile(loss, 95))

                x = np.linspace(0, self.n_epochs, len(loss))
                plt.plot(x, loss, c=COLORS[net_name],
label=f'{net_name}', alpha=0.8)
                plt.axhline(min(loss), linewidth=2,
linestyle='dashed', c=COLORS[net_name], alpha=0.8)

            plt.yscale('log')
            ax.xaxis.set_major_locator(MaxNLocator(integer=True))
            ax.yaxis.set_major_formatter(FormatStrFormatter('%.3g'))
            ax.yaxis.set_minor_formatter(FormatStrFormatter('%.3g'))
```

```
            plt.xlim([0, self.n_epochs])
            plt.ylim([y_min - 5e-4, y_max])

            plt.xlabel('Epoch')
            plt.ylabel(f'Log Loss (lr =
{to_scientific_notation(self.lr)})')

            plt.grid(which='both')
            plt.axhline(y_min - 10, linewidth=2, linestyle='dashed',
c='k', label='Minimal loss levels')
            plt.legend()
            plt.tight_layout()
            plt.savefig(self.artifacts_dir/f'{group}_comparison.png')
            plt.clf()

    def plot(self) -> None:
        self.plot_loss()

class SummarizeCIFAR10(BaseExperiment):
    def __init__(self, artifacts_dir: Directory, n_epochs: int = 20,
window_size: int = 128):
        super().__init__(artifacts_dir, Preprocessor)
        self.window_size = window_size
        self.n_epochs = n_epochs

    def plot_best_scores(self) -> None:
        _, ax = plt.subplots(figsize=(8, 6))
        markers = ['o', '^', 'X', 'd', 'p', 'P', 'v', '*']
        handles = []

        for net_name in ['ReLU', 'ELU', 'GELU', 'SLU (shared)', 'SLU
(individual)']:
```

```
            handles.append(mpatches.Patch(label=net_name,
color=COLORS[net_name]))

        marker_i = 0
        for item in Path(self.artifacts_dir).iterdir():
            if not item.is_dir(): continue
            architecture_name = item.name
            handles.append(mlines.Line2D([], [], color='black',
marker=markers[marker_i], linestyle='None',
                                markersize=10, label=architecture_name))
            marker_i += 1

        handles.append(mlines.Line2D([], [], color='black',
marker='s', linestyle='None',
                           markersize=15, label='Average'))
        handles.append(mlines.Line2D([], [], color='black', marker='',
label='', alpha=0))

        with open(self.artifacts_dir/f'mean_best_scores.txt', 'wt') as
f:
            for net_name in ['ReLU', 'ELU', 'GELU', 'SLU (shared)',
'SLU (individual)']:
                min_iter_list = []
                min_loss_list = []
                marker_i = 0
                for item in Path(self.artifacts_dir).iterdir():
                    if not item.is_dir(): continue
                    architecture_name = item.name

                    subfolder_name =
self.artifacts_dir/architecture_name/(net_name.replace(' ', '_'))
                    loss = torch.load(subfolder_name/'val_loss.pth')
                    loss = smooth_data(loss, self.window_size)
```

```
                    x = np.linspace(0, self.n_epochs, len(loss))
                    min_iter = x[np.argmin(loss)]
                    min_loss = np.min(loss)

                    f.write(f'{net_name}; {architecture_name};
{min_iter}; {min_loss}\n')
                    min_iter_list.append(min_iter)
                    min_loss_list.append(min_loss)
                    plt.scatter(min_iter, min_loss,
c=COLORS[net_name], s=80, marker=markers[marker_i])
                    marker_i += 1

                mean_iter, mean_loss = np.mean(min_iter_list),
np.mean(min_loss_list)
                plt.scatter(mean_iter, mean_loss, c=COLORS[net_name],
s=100,
                            marker='s', edgecolors='k', linewidth=3)

                f.write(f'\t{net_name}; {mean_iter}; {mean_loss}\n')

        ax.xaxis.set_major_locator(MaxNLocator(integer=True))

        plt.xlabel('Epoch')
        plt.ylabel(f'Log Loss')

        plt.grid(which='both')

        plt.legend(handles=handles, ncol=2)
        plt.tight_layout()
        plt.savefig(self.artifacts_dir/f'best_scores.png')
        plt.clf()
```

```
    def train(self) -> None:
        pass

    def plot(self) -> None:
        self.plot_best_scores()
```

src\experiments\classification_mnist.py

```
from pathlib import Path

import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.lines as mlines
from matplotlib.ticker import FormatStrFormatter, MaxNLocator
import torch.utils.data as data
from torchvision import datasets, transforms

from ..colors import COLORS
from ..layers import SLU
from ..types import Callable, Directory, Tuple
from ..utils import smooth_data, to_scientific_notation, train
from .base import BaseExperiment, BasePreprocessor


class Preprocessor(BasePreprocessor):
    '''NOTE:
    No preprocessor is needed
    as pytorch has a prebuilt MNIST integration
    with torchvision transforms support
```

```
    '''
    def __init__(self):
        super().__init__()

    def fit(self, X: torch.Tensor, y: torch.Tensor) -> None:
        pass

    def transform(self, X: torch.Tensor, y: torch.Tensor) ->
Tuple[torch.Tensor, torch.Tensor]:
        return X, y


class ClassifyMNIST(BaseExperiment):
    def __init__(self, artifacts_dir: Directory,
                 n_layers: int, n_neurons: int, force_retrain: bool,
                 noise: float = 0,
                 n_epochs: int = 20, lr: float = 1e-3,
                 is_verbose: bool = True, window_size: int = 128):
        super().__init__(artifacts_dir, Preprocessor)
        self.n_layers = n_layers
        self.n_neurons = n_neurons
        self.force_retrain = force_retrain
        self.noise = noise
        self.n_epochs = n_epochs
        self.lr = lr
        self.is_verbose = is_verbose
        self.window_size = window_size

    def init_network(self, activation_fn: Callable[[int], nn.Module])
-> nn.Sequential:
        layers = [
            nn.Linear(784, self.n_neurons),
            activation_fn(self.n_neurons)
```

```
        ]
        for _ in range(self.n_layers - 1):
            layers.append(nn.Linear(self.n_neurons, self.n_neurons))
            layers.append(activation_fn(self.n_neurons))
        layers.append(nn.Linear(self.n_neurons, 10))
        return nn.Sequential(*layers)

    def train(self) -> None:
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,)),
            transforms.Lambda(torch.flatten),
            transforms.Lambda(lambda x: x + torch.randn(x.shape) *
self.noise)
        ])

        train_dataset = datasets.MNIST('data', train=True,
download=True, transform=transform)
        val_dataset = datasets.MNIST('data', train=False,
transform=transform)
        train_loader = data.DataLoader(train_dataset, batch_size=128)
        val_loader = data.DataLoader(val_dataset, batch_size=128)

        relu_net = self.init_network(lambda _: nn.ReLU())
        elu_net = self.init_network(lambda _: nn.ELU())
        gelu_net = self.init_network(lambda _: nn.GELU())
        slu_shared_net = self.init_network(lambda _: SLU())
        slu_ind_net = self.init_network(lambda size: SLU(size))

        self.nets = {
            'ReLU': relu_net,
            'ELU': elu_net,
            'GELU': gelu_net,
```

```
            'SLU (shared)': slu_shared_net,
            'SLU (individual)': slu_ind_net,
        }

        for net_name, net in self.nets.items():
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            if not self.force_retrain and subfolder_name.exists():
                if self.is_verbose:
                    print(f'{net_name} is pretrained, moving on')
                continue

            if self.is_verbose:
                print(f'{net_name} training started')

            train_loss_log, val_loss_log = train(
                model=net,
                train_loader=train_loader,
                val_loader=val_loader,
                criterion=nn.CrossEntropyLoss(),
                lr=self.lr,
                n_epochs=self.n_epochs,
                is_verbose=self.is_verbose
            )

            subfolder_name.mkdir(exist_ok=True)
            torch.save(train_loss_log,
subfolder_name/'train_loss.pth')
            torch.save(val_loss_log, subfolder_name/'val_loss.pth')
            torch.save(net, subfolder_name/'net.pth')

            if self.is_verbose:
                print(f'{net_name} training finished')
```

```
    def plot_loss(self) -> None:
        for group in ['train', 'val']:
            _, ax = plt.subplots(figsize=(8, 6))
            y_min, y_max = np.inf, -np.inf
            for net_name in ['ReLU', 'ELU', 'GELU', 'SLU (shared)',
'SLU (individual)']:
                subfolder_name =
self.artifacts_dir/(net_name.replace(' ', '_'))
                loss = torch.load(subfolder_name/f'{group}_loss.pth')
                loss = smooth_data(loss, self.window_size)

                y_min = min(y_min, min(loss))
                y_max = max(y_max, np.percentile(loss, 95))

                x = np.linspace(0, self.n_epochs, len(loss))
                plt.plot(x, loss, c=COLORS[net_name],
label=f'{net_name}', alpha=0.8)
                plt.axhline(min(loss), linewidth=2,
linestyle='dashed', c=COLORS[net_name], alpha=0.8)

            plt.yscale('log')
            ax.xaxis.set_major_locator(MaxNLocator(integer=True))
            ax.yaxis.set_major_formatter(FormatStrFormatter('%.3g'))
            ax.yaxis.set_minor_formatter(FormatStrFormatter('%.3g'))

            plt.xlim([0, self.n_epochs])
            plt.ylim([y_min - 5e-4, y_max])

            plt.xlabel('Epoch')
            plt.ylabel(f'Log Loss (lr =
{to_scientific_notation(self.lr)})')
```

```
            plt.grid(which='both')
            plt.axhline(y_min - 10, linewidth=2, linestyle='dashed',
c='k', label='Minimal loss levels')
            plt.legend()
            plt.tight_layout()
            plt.savefig(self.artifacts_dir/f'{group}_comparison.png')
            plt.clf()

    def plot(self) -> None:
        self.plot_loss()


class SummarizeMNIST(BaseExperiment):
    def __init__(self, artifacts_dir: Directory, n_epochs: int = 20,
window_size: int = 128):
        super().__init__(artifacts_dir, Preprocessor)
        self.window_size = window_size
        self.n_epochs = n_epochs

    def plot_best_scores(self) -> None:
        _, ax = plt.subplots(figsize=(8, 6))
        markers = ['o', '^', 'X', 'd', 'p', 'P', 'v', '*']
        handles = []

        for net_name in ['ReLU', 'ELU', 'GELU', 'SLU (shared)', 'SLU
(individual)']:
            handles.append(mpatches.Patch(label=net_name,
color=COLORS[net_name]))

        marker_i = 0
        for item in Path(self.artifacts_dir).iterdir():
            if not item.is_dir(): continue
            architecture_name = item.name
```

```
            handles.append(mlines.Line2D([], [], color='black',
marker=markers[marker_i], linestyle='None',
                                markersize=10, label=architecture_name))
            marker_i += 1

        handles.append(mlines.Line2D([], [], color='black',
marker='s', linestyle='None',
                           markersize=15, label='Average'))

        with open(self.artifacts_dir/f'mean_best_scores.txt', 'wt') as
f:
            for net_name in ['ReLU', 'ELU', 'GELU', 'SLU (shared)',
'SLU (individual)']:
                min_iter_list = []
                min_loss_list = []
                marker_i = 0
                for item in Path(self.artifacts_dir).iterdir():
                    if not item.is_dir(): continue
                    architecture_name = item.name

                    subfolder_name =
self.artifacts_dir/architecture_name/(net_name.replace(' ', '_'))
                    loss = torch.load(subfolder_name/'val_loss.pth')
                    loss = smooth_data(loss, self.window_size)

                    x = np.linspace(0, self.n_epochs, len(loss))
                    min_iter = x[np.argmin(loss)]
                    min_loss = np.min(loss)

                    f.write(f'{net_name}; {architecture_name};
{min_iter}; {min_loss}\n')
                    min_iter_list.append(min_iter)
                    min_loss_list.append(min_loss)
```

```
                    plt.scatter(min_iter, min_loss,
c=COLORS[net_name], s=80, marker=markers[marker_i])
                    marker_i += 1

                mean_iter, mean_loss = np.mean(min_iter_list),
np.mean(min_loss_list)
                plt.scatter(mean_iter, mean_loss, c=COLORS[net_name],
s=100,
                            marker='s', edgecolors='k', linewidth=3)

                f.write(f'\t{net_name}; {mean_iter}; {mean_loss}\n')

        plt.yscale('log')
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        ax.yaxis.set_major_formatter(FormatStrFormatter('%.3g'))
        ax.yaxis.set_minor_formatter(FormatStrFormatter('%.3g'))

        plt.xlabel('Epoch')
        plt.ylabel(f'Log Loss')

        plt.grid(which='both')

        plt.legend(handles=handles, ncol=2)
        plt.tight_layout()
        plt.savefig(self.artifacts_dir/f'best_scores.png')
        plt.clf()

    def train(self) -> None:
        pass

    def plot(self) -> None:
        self.plot_best_scores()
```

src\experiments\regression_1d.py

```
from dataclasses import dataclass

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from matplotlib.ticker import FormatStrFormatter, MaxNLocator

from ..colors import COLORS
from ..layers import SLU
from ..types import Callable, Directory, Tuple
from ..utils import train, smooth_data, to_scientific_notation
from .base import BaseExperiment, BasePreprocessor


@dataclass
class Dataset1D:
    name: str
    latex: str
    range: Tuple[float, float]
    function: Callable


class Preprocessor(BasePreprocessor):
    def __init__(self):
        super().__init__()

    def fit(self, X: torch.Tensor, y: torch.Tensor) -> None:
        self.x_max = torch.max(X)
        self.x_min = torch.min(X)
        self.y_max = torch.max(y)
```

```
        self.y_min = torch.min(y)

    def transform(self, X: torch.Tensor, y: torch.Tensor) ->
Tuple[torch.Tensor, torch.Tensor]:
        X_norm = (X - self.x_min) / (self.x_max - self.x_min)
        y_norm = (y - self.y_min) / (self.y_max - self.y_min)
        return X_norm, y_norm


class Regress1D(BaseExperiment):
    datasets = [
        Dataset1D('square', 'x^2', (-5, 5), lambda x: x**2),
        Dataset1D('root', r'\sqrt{x}', (0, 5), lambda x: x**0.5),
        Dataset1D('reciprocal', r'\frac{1}{x}', (1, 5), lambda x:
x**(-1)),
    ]

    def __init__(self, artifacts_dir: Directory, dataset: str,
                 n_samples: int = 2000, noise: float = .3,
                 n_epochs: int = 100, lr: float = 1e-3,
                 is_verbose: bool = True, window_size: int = 40):
        super().__init__(artifacts_dir, Preprocessor)
        self.dataset: Dataset1D = next((ds for ds in
Regress1D.datasets if ds.name == dataset), None)
        if self.dataset is None:
            raise ModuleNotFoundError('No dataset with such name
exists')
        self.n_samples = n_samples
        self.noise = noise
        self.n_epochs = n_epochs
        self.lr = lr
        self.is_verbose = is_verbose
        self.window_size = window_size
```

```
    @staticmethod
    def make_data(dataset: Dataset1D, n_samples: int, noise: float) ->
Tuple[np.ndarray, np.ndarray]:
        X = np.linspace(*dataset.range, n_samples)[:, None]
        y = dataset.function(X)
        y += np.random.randn(n_samples, 1) * 0.1*noise
        return X, y

    @staticmethod
    def init_network(activation_fn: Callable[[int], nn.Module]) ->
nn.Sequential:
        return nn.Sequential(
            nn.Linear(1, 5),
            activation_fn(5),
            nn.Linear(5, 5),
            activation_fn(5),
            nn.Linear(5, 1),
        )

    def train(self) -> None:
        X, y = Regress1D.make_data(
            dataset=self.dataset,
            n_samples=self.n_samples,
            noise=self.noise
        )
        X = torch.tensor(X, dtype=torch.float32)
        y = torch.tensor(y, dtype=torch.float32)
        train_loader, val_loader = self.get_data_loaders(X, y)

        relu_net = Regress1D.init_network(lambda _: nn.ReLU())
        slu_shared_net = Regress1D.init_network(lambda _: SLU())
        slu_ind_net = Regress1D.init_network(lambda size: SLU(size))
```

```
        self.nets = {
            'ReLU': relu_net,
            'SLU (shared)': slu_shared_net,
            'SLU (individual)': slu_ind_net,
        }

        for net_name, net in self.nets.items():
            train_loss_log, val_loss_log = train(
                model=net,
                train_loader=train_loader,
                val_loader=val_loader,
                criterion=nn.MSELoss(),
                lr=self.lr,
                n_epochs=self.n_epochs,
                is_verbose=self.is_verbose
            )
            subfolder_name = self.artifacts_dir/(net_name.replace(" ",
"_"))
            subfolder_name.mkdir(exist_ok=True)
            torch.save(train_loss_log,
subfolder_name/'train_loss.pth')
            torch.save(val_loss_log, subfolder_name/'val_loss.pth')

        torch.save(self.nets, self.artifacts_dir/f'nets.pth')

    def plot_regression(self) -> None:
        X, y = Regress1D.make_data(
            dataset=self.dataset,
            n_samples=self.n_samples,
            noise=self.noise
        )
        X_tensor = torch.tensor(X, dtype=torch.float32)
```

```
        y_tensor = torch.tensor(y, dtype=torch.float32)
        if not self.preprocessor.is_trained:
            self.preprocessor =
torch.load(self.artifacts_dir/'preprocessor.pth')
        X_tensor, y_tensor = self.preprocessor.transform(X_tensor,
y_tensor)
        X = X_tensor.numpy()
        y = y_tensor.numpy()

        if not hasattr(self, 'nets'):
            self.nets = torch.load(self.artifacts_dir/'nets.pth')

        for net_name, net in self.nets.items():
            with torch.no_grad(): y_ = net(X_tensor)
            plt.figure(figsize=(8, 6))
            plt.scatter(X, y, alpha=0.8, label=f'exact (noise =
{self.noise})', c='k')
            plt.scatter(X, y_, alpha=0.8, label='approximate',
c=COLORS[net_name])
            plt.xlabel('x (normalized)')
            plt.ylabel('y (normalized)')
            plt.legend()
            plt.title(f'$y={self.dataset.latex}$ approximation —
{net_name}')
            plt.tight_layout()
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            subfolder_name.mkdir(exist_ok=True)
            plt.savefig(subfolder_name/'approximation.png')
            plt.clf()

    def plot_loss(self) -> None:
        _, ax = plt.subplots(figsize=(8, 6))
```

```
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        for net_name in ['ReLU', 'SLU (shared)', 'SLU (individual)']:
            subfolder_name = self.artifacts_dir/(net_name.replace(' ',
'_'))
            train_loss = torch.load(subfolder_name/'train_loss.pth')
            train_loss = smooth_data(train_loss, self.window_size)
            x_train = np.linspace(0, self.n_epochs, len(train_loss))
            plt.plot(x_train, train_loss,
                     c=COLORS[net_name], label=f'{net_name} - train',
alpha=0.7)
            val_loss = torch.load(subfolder_name/'val_loss.pth')
            val_loss = smooth_data(val_loss, self.window_size)
            plt.tight_layout()
            x_val = np.linspace(0, self.n_epochs, len(val_loss))
            plt.plot(x_val, val_loss,
                     c=COLORS[net_name], label=f'{net_name} - val')

        plt.yscale('log')
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        ax.yaxis.set_major_formatter(FormatStrFormatter('%.3g'))

        plt.xlim([0, self.n_epochs])
        plt.xlabel('Epoch')
        plt.ylabel(f'MSE Loss (lr =
{to_scientific_notation(self.lr)})')
        plt.grid()
        plt.legend()
        plt.tight_layout()
        plt.savefig(self.artifacts_dir/'train_comparison.png')
        plt.clf()

    def plot(self) -> None:
        self.plot_regression()
```

```
        self.plot_loss()
```

src\experiments__init__.py

```
from .classification_2d import Classify2D
from .regression_1d import Regress1D
from .classification_mnist import ClassifyMNIST, SummarizeMNIST
from .classification_cifar10 import ClassifyCIFAR10, SummarizeCIFAR10
```

src\visualizations\plot_activations.py

```
import matplotlib.pyplot as plt
import torch

from ..colors import COLORS
from ..types import File, Union


def sigmoid(x: torch.Tensor) -> torch.Tensor:
    return (1 + torch.exp(-x))**(-1)


def relu(x: torch.Tensor) -> torch.Tensor:
    return torch.max(torch.tensor(0), x)


def elu(x: torch.Tensor, alpha: float = 1.0) -> torch.Tensor:
    return torch.where(x > 0, x, alpha*(torch.exp(x) - 1))


def gelu(x: torch.Tensor) -> torch.Tensor:
```

```
    return 0.5 * x * (1 +
torch.tanh(torch.sqrt(torch.tensor(2/torch.pi)) * (x + 0.044715 *
x**3)))


def slu(x: torch.Tensor, k: float = 0.0) -> torch.Tensor:
    A = torch.log(1 + torch.abs(x))
    B = k * A.pow(2)
    return torch.where(x > 0, x + B, B - A)


class PlotActivations:
    functions_info = {
        'Sigmoid': (sigmoid, ()),
        'ReLU': (relu, ()),
        'ELU': (elu, ()),
        'GELU': (gelu, ()),
        'SLU (k = -0.2)': (slu, (-0.2,)),
        'SLU (k = 0)': (slu, (0,)),
        'SLU (k = 0.2)': (slu, (0.2,)),
    }

    to_exclude = (
        'Sigmoid',
    )

    def __init__(self, fig_path: File, x_min: float = -5., x_max:
float = 5., function: Union[str, None] = None):
        assert function is None or function in
PlotActivations.functions_info
        self.fig_path = fig_path
        self.x_min = x_min
        self.x_max = x_max
```

```
        self.function = function

    def evaluate(self):
        X = torch.linspace(self.x_min, self.x_max, 500,
requires_grad=True)
        X_arr = X.detach().numpy()

        _, axs = plt.subplots(1, 2, figsize=(10, 5))

        if self.function is None:
            axs[0].set_title('Functions')
            axs[1].set_title('Derivatives')
            y_lim = relu(torch.tensor(self.x_max)).numpy()
            axs[0].set_ylim([-y_lim, y_lim])
        else:
            axs[0].set_title('Function')
            axs[1].set_title('Derivative')

        axs[0].set_xlim([self.x_min, self.x_max])
        axs[0].axhline(y=0, color='k', alpha=0.8)
        axs[0].axvline(x=0, color='k', alpha=0.8)

        axs[1].set_xlim([self.x_min, self.x_max])
        axs[1].plot(X_arr, X_arr*0, color='k', alpha=0.8)
        axs[1].plot(X_arr, X_arr*0 + 1, color='g', linewidth=10,
alpha=0.2)
        axs[1].plot(X_arr, X_arr*0, color='r', linewidth=10,
alpha=0.2)
        axs[1].axvline(x=0, color='k', alpha=0.8)

        if self.function is None:
            for i, (label, (f, params)) in
enumerate(PlotActivations.functions_info.items()):
```

```
                if label in PlotActivations.to_exclude: continue
                Y = f(X, *params)
                Y.backward(torch.ones_like(X), retain_graph=True)
                Y_prime = X.grad
                ls = ['-', '--', '-.', ':'][i % 4]
                lw = [3, 4, 5, 6][i % 4]
                axs[0].plot(X_arr, Y.detach().numpy(), alpha=0.8,
                            c=COLORS[label], label=label, ls=ls,
lw=lw)
                axs[1].plot(X_arr, Y_prime.detach().numpy(),
alpha=0.8,
                            c=COLORS[label], label=label, ls=ls,
lw=lw)
                X.grad.zero_()
        else:
            f, params = PlotActivations.functions_info[self.function]
            Y = f(X, *params)
            Y.backward(torch.ones_like(X), retain_graph=True)
            Y_prime = X.grad
            axs[0].plot(X_arr, Y.detach().numpy(),
                        c=COLORS[self.function], lw=6)
            axs[1].plot(X_arr, Y_prime.detach().numpy(),
                        c=COLORS[self.function], lw=6)

        axs[0].grid(alpha=0.4)
        axs[1].grid(alpha=0.4)

        if self.function is None:
            axs[0].legend()
            axs[1].legend()

        plt.tight_layout()
```

```
        plt.savefig(self.fig_path)
        plt.clf()
```

src\visualizations\plot_slu.py

```
import matplotlib.pyplot as plt
import numpy as np
import torch
from matplotlib import cm
from matplotlib.colorbar import ColorbarBase
from matplotlib.ticker import FormatStrFormatter

from ..types import File


def relu(x: torch.Tensor) -> torch.Tensor:
    return torch.max(torch.tensor(0), x)


def slu(x: torch.Tensor, k: float = 0.0) -> torch.Tensor:
    A = torch.log(1 + torch.abs(x))
    B = k * A.pow(2)
    return torch.where(x > 0, x + B, B - A)


class PlotSLU:
    def __init__(self, fig_path: File,
                 k_min: float = -0.5, k_max: float = 0.5,
                 x_min: float = -20., x_max: float = 20.):
        self.fig_path = fig_path
        self.k_min = k_min
        self.k_max = k_max
```

```
        self.x_min = x_min
        self.x_max = x_max

    def evaluate(self):
        X = torch.linspace(self.x_min, self.x_max, 500,
requires_grad=True)
        X_arr = X.detach().numpy()
        k_values = np.linspace(self.k_min, self.k_max, 120)

        _, axs = plt.subplots(1, 3, figsize=(10, 5), gridspec_kw={
                              'width_ratios': [48, 4, 48]})
        axs[0].set_title('SLU for different values of k')
        axs[1].set_title('k')
        axs[2].set_title('Derivative of SLU for different values of
k')

        axs[0].set_xlim([self.x_min, self.x_max])
        axs[0].axhline(y=0, color='k', alpha=0.8)
        axs[0].axvline(x=0, color='k', alpha=0.8)

        axs[2].set_xlim([self.x_min, self.x_max])
        axs[2].plot(X_arr, X_arr*0, color='k', alpha=0.8)
        axs[2].plot(X_arr, X_arr*0 + 1, color='g', linewidth=10,
alpha=0.2)
        axs[2].plot(X_arr, X_arr*0, color='r', linewidth=10,
alpha=0.2)
        axs[2].axvline(x=0, color='k', alpha=0.8)

        for k in k_values:
            Y = slu(X, k)
            Y.backward(torch.ones_like(X), retain_graph=True)
            Y_prime = X.grad
            k_norm = (k - self.k_min) / (self.k_max - self.k_min)
```

```
            axs[0].plot(X_arr, Y.detach().numpy(),
                        alpha=0.4, color=cm.plasma(k_norm))
            axs[2].plot(X_arr, Y_prime.detach().numpy(),
                        alpha=0.4, color=cm.plasma(k_norm))
            X.grad.zero_()

        axs[0].grid(alpha=0.4)
        axs[2].grid(alpha=0.4)
        ColorbarBase(axs[1], cmap='plasma',
                     ticks=np.linspace(self.k_min, self.k_max, 5),
                     values=np.linspace(self.k_min, self.k_max, 100),
                     format=FormatStrFormatter('%.2f'),
ticklocation='left')
        plt.tight_layout()

        plt.savefig(self.fig_path)
        plt.clf()
```

src\visualizations__init__.py

```
from .plot_activations import PlotActivations
from .plot_slu import PlotSLU
```