

Харківський національний університет імені В.Н. Каразіна
Факультет математики і інформатики
Кафедра прикладної математики

Курсова науково-дослідницька робота
на тему «*Математичні основи штучних
нейронних мереж*»

Виконав: студент групи МП41 IV курсу
(перший бакалаврський рівень),
спеціальності 113
“Прикладна математика”
освітньої програми
“Прикладна математика”
Захаров Д.О.

Керівник: доктор фіз.-мат. наук,
професор кафедри
прикладної математики
Ігнатович С.Ю.

Харків — 2024 рік

Анотації

Захаров Д.О. Математичні основи штучних нейронних мереж.

В сучасному світі важко уявити сферу, де б не використовувались штучні нейронні мережі. Проте, незважаючи на таку кількість різноманітних досліджень, більшість з них не обґрунтують чому саме нейронні мережі так добре апроксимують данні та видають гарну точність. Ця робота призначена опису фундаменту нейронних мереж та, в певній мірі, формалізації процесу навчання: що саме розв'язують нейронні мережі, чому вони (теоретично) здатні апроксимувати важливі для нас залежності і як на практиці реалізувати процес підбору параметрів моделі.

Zakharov D.O. Mathematical foundations of artificial neural networks. In the modern world, it is almost impossible to imagine a technical field where artificial neural networks are not used. However, despite the abundance of research of their applications, most of them do not explain why neural networks are so good at approximating data and achieving high accuracy. This work is dedicated to describing the fundamentals of neural networks and, to some extent, formalizing the learning process: what neural networks solve, why they are (theoretically) capable of approximating important dependencies, and how to practically implement the process of selecting model parameters.

Зміст

Анотації	2
Вступ	4
1. Задачі машинного навчання	6
1.1. Що таке модель?	6
1.2. Параметризація моделей	8
1.3. Основні задачі машинного навчання	11
2. Теорія Апроксимації	13
2.1. Апроксимація сігмоїdalьними функціями: Теорема Цибенко	13
2.1.1. Постановка задачі	13
2.1.2. Узгодження з сучасною термінологією	15
2.1.3. Теореми Цибенко	16
2.1.4. Практичний Приклад	18
2.1.5. Подальший розвиток архітектури Цибенко	20
2.2. Теорема Колмогорова-Арнольда	25
2.2.1. Історія та Мотивація	25
2.2.2. Мережі Колмогорова-Арнольда	27
2.2.3. Порівняння з MLP архітектурою	31
Висновки	33
А. Програмний код для тренування мережі Цибенко	38

Вступ

Без всяких сумнівів, нейронні мережі є одними з найбільш популярних інструментів машинного навчання для пошуку складних залежностей. Вони використовуються в безлічі різних областей, таких як комп’ютерний зір [Wan+24], обробка природних мов [Qin+24] та біометричних даних [Min+23], розробка рекомендаційних систем [Li+24], генерації зображень [ITD23] тощо. Наші нещодавні дослідження [ZKF24; Kuz+24; KZF24] додатково підтвердили високу ефективність нейронних мереж у задачах кібербезпеки та систем захисту біометричних даних. Що уж там, мабуть кожна людина чула або використовувала новітні розробки OpenAI — архітектуру *GPT-3* (Generative Pre-trained Transformer) [Bro20] або *Open AI o1*, що вже навіть здатна розв’язувати задачі з міжнародної олімпіади з математики або аналізувати складні наукові тексти.

Проте, незважаючи на таку кількість різноманітних досліджень, більшість з них зводиться до доволі типічного алгоритму (звичайно, з варіаціями в залежності від конкретної задачі):

1. Визначення типу задачі (класифікація, регресія, сегментація, тощо).
2. Підбір набору даних (далі, скорочено — датасет).
3. Вибір архітектури моделі, функції втрати та метрик якості.
4. Тренування та корегування параметрів моделі для максимізації метрики якості.
5. Аналіз результатів.

Проте, протягом цього процесу, ми зазвичай пропускаємо одне доволі

фундаментальне питання: а чому, взагалі кажучи, обрані архітектури нейронних мереж здатні вирішувати такі задачі? Звичайно, що для практичних задач це питання часто не принципове: якщо воно працює і працює добре, то цього більш, ніж достатньо¹.

Саме тому ми присвятили цю роботу опису фундаменту нейронних мереж та, в певній мірі, формалізації процесу навчання: що саме розв'язують нейронні мережі, чому вони (теоретично) здатні апроксимувати важливі для нас залежності і як на практиці реалізувати процес підбору параметрів моделі.

¹Тим не менш, в сучасних роботах іноді трапляється спроба пояснити, чому описана методологія може теоретично дати, скажімо, мінімум для певної метрики, як це було зроблено в оригінальному описі генеративних адверсальних мереж (Generative Adversarial Networks) [Goo+14]

Розділ 1

Задачі машинного навчання

1.1. Що таке модель?

Насправді, чітко поставити задачу сучасної теорії машинного навчання одним визначенням дуже складно. Це пов'язано з тим, що підхід до розв'язку задачі дуже залежить від того, що ми очікуємо від так званої *моделі машинного навчання*. Що ж ми розуміємо під терміном “модель”?

Найчастіше, на вхід подається певний набір даних \mathcal{D} . Це можуть бути картинки разом з маркуванням, що зображено. Можуть бути текстові данні, чисельні данні, аудіо- або відео-записи, результати вимірювань на сенсорних пристроях, тощо. Маючи цей набір даних, ми часто хочемо зрозуміти певні закономірності в цих даних. Функцію, що бере певний вхід, що містить інформацію про об'єкт, і повертає вихід, що містить закономірність, часто називаємий *передбаченням*, як раз і називають **моделлю**. Далі наведемо кілька нетривільних прикладів з задач машинного навчання.

Приклад 1.1 (Класифікація цифр). Будь-яке сіро-біле зображення \mathbf{X} розміру $W \times H$ пікселів можна розглядати як матрицю $\mathbf{X} \in \mathbb{R}^{W \times H}$, де кожен елемент матриці $X_{i,j}$ — це значення яскравості відповідного пікселя на позиції (i, j) (наприклад, значення 0 може позначати чорний колір, 1 — білий, а значення проміж — степінь сірості)¹.

Нехай нам наданий набір $\mathcal{D} = \{(\mathbf{X}_n, y_n)\}_{1 \leq n \leq N}$ — пари “зображення-

¹Іноді таку множину явно записують як $[0, 1]^{W \times H}$, щоб підкреслити, що значення нормалізовані на відрізок $[0, 1]$. Тим не менш, використовуємо позначення $\mathbb{R}^{W \times H}$, оскільки оптимальна нормалізація даних залежить від обраної методології

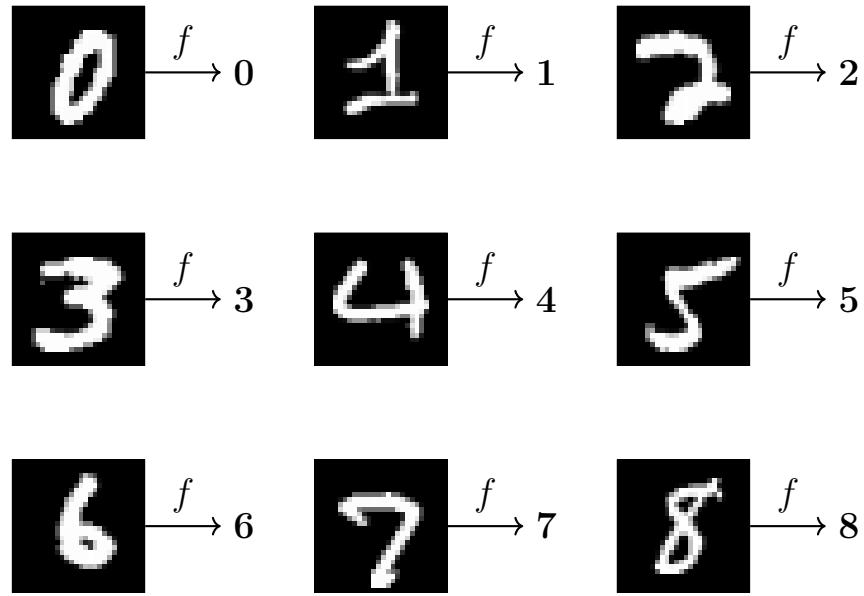


Рис. 1.1: Приклад класифікації цифр. Маючи зображення $X \in \mathbb{R}^{W \times H}$, наша функція дає дискретне передбачення $f(X) \in \{0, \dots, 9\}$ — цифра, яка зображена на зображенні X .

цифра”, де $y_n \in \{0, \dots, 9\}$. Наша ціль — побудувати так звану *класифікаційну модель* $f : \mathbb{R}^{W \times H} \rightarrow \{0, \dots, 9\}$, яка буде приймати на вхід зображення \mathbf{X} та видавати цифру $f(\mathbf{X})$, що зображена. Приклад зображенено на Рисунку 1.1 на базі набору даних MNIST [Den12].

Приклад 1.2 (Розпізнавання дрону). Нехай наша задача: це розпізнати розташування дронів на кадрі відео. Нехай в нас кольорове зображення розміру $W \times H$. Тоді зображення \mathbf{X} береться з множини $\mathbb{R}^{W \times H \times 3}$, де замість яскравості пікселя, маємо тривимірний вектор $(r, g, b) \in \mathbb{R}^3$ — колір пікселя (інтенсивність червоного, зеленого та синіх каналів, відповідно).

Поділимо наше зображення на сітку розміру $n_W \times n_H$. Тоді в якості моделі можна взяти функцію $f : \mathbb{R}^{W \times H \times 3} \rightarrow \mathbb{R}^{n_W \times n_H}$, що видає матрицю $\{p(S_{i,j} | \mathbf{X})\}_{1 \leq i \leq n_W, 1 \leq j \leq n_H}$, де $S_{i,j}$ — подія “в клітинці (i, j) сітки знаходиться дрон”. Ця модель візуально проілюстрована на Рис. 1.2. Відмітимо, що ідея описаної конструкції частково використовується в архітектурі YOLO (You Look Only Once) [Red+16] — одній з найпопулярніших архітектур для розпізнавання об'єктів на зображеннях.

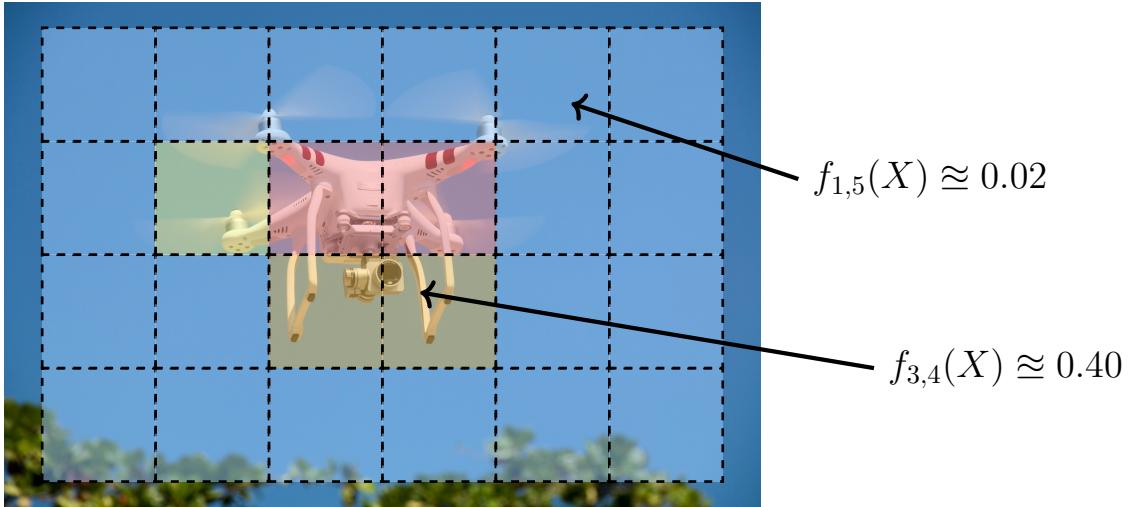


Рис. 1.2: Приклад розпізнавання дронів. Маючи зображення $\mathbf{X} \in \mathbb{R}^{W \times H \times 3}$, наша функція дає ймовірність того, що на кожному сегменті зображення знаходиться дрон. Чим тепліше кольори, тим вища ймовірність.

Зауваження 1.3. Зверніть увагу, що в обох прикладах вище, модель f видає дискретне або неперервне передбачення. Це може бути класифікація, регресія, сегментація, тощо. Це залежить від задачі, що ми розв'язуємо.

1.2. Параметризація моделей

Проте, як саме ми будуємо моделі? Іншими словами, як обрати функцію f ? Зазвичай, ми маємо задати певне сімейство функцій \mathcal{F} , поміж яких ми шукаємо “найкращу”² функцію. Наприклад, це може бути сімейство лінійних/квадратичних функцій або функцій вигляду $f(x) = (\theta_1 x^2, \theta_2 x)$. Звичайно, оскільки алгоритм пошуку f має бути заданий програмно, то ми не можемо покласти в якості \mathcal{F} , скажімо, просто $L^2(\mathbb{R})$, бо тоді не зрозуміло, як саме задати алгоритм пошуку f . Саме тому, для практичних застосувань, ми *параметризуємо* функції набором параметрів $\boldsymbol{\theta} \in \Theta \subset \mathbb{R}^m$. Таким чином, наша модель має вигляд $f(\mathbf{x}|\boldsymbol{\theta})$, де параметри $\boldsymbol{\theta}$ можна змінювати, щоб зробити модель точною.

²Що таке “найкраща” функція, ми обговоримо пізніше.

Приклад 1.4 (Лінійна Регресія). Одна з класичних та найбільш відомих моделей — це лінійна регресія. Нехай маємо набір даних $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{1 \leq n \leq N} \subset \mathbb{R}^m \times \mathbb{R}$ і ми вважаємо, що залежність між \mathbf{x}_n та y_n лінійна. Іншими словами, ми введемо модель $f(\mathbf{x}|\boldsymbol{\theta}) = \mathbf{w}^\top \mathbf{x} + \beta$, де $\boldsymbol{\theta} = (\mathbf{w}, \beta) \in \mathbb{R}^{m+1}$ — вектор параметрів.

В цілому, саме так дуже часто вводиться модель лінійної регресії. Проте, часто в літературі можна зустріти у якості моделі *роздiл* величини y :

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^\top \mathbf{x} + \beta, \sigma^2), \quad (1.1)$$

де $\boldsymbol{\theta} = (\mathbf{w}, \beta, \sigma^2)$ — вектор параметрів, а $\mathcal{N}(y|\mu, \sigma^2)$ — щільність нормального розподiлу. Така альтернатива дозволяє ввести бiльш гнучку модель, яка може давати степiнь впевненостi у своїх передбаченнях.

Зauważення 1.5. Приклад вище легко узагальнити для випадку, коли вихiд $\mathbf{y} \in \mathbb{R}^r$ — вектор. В такому випадку, моделлю є передбачення наступного розподiлу:

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \prod_{i=1}^r \mathcal{N}(y_i|\mathbf{w}_i^\top \mathbf{x} + \beta_i, \sigma_i^2), \quad \mathbf{w}_i \in \mathbb{R}^m, \beta_i, \sigma_i \in \mathbb{R} \quad (1.2)$$

Отже, нехай ми вибрали параметризацiю моделi. Як тепер обрати найкращi параметри? Тут, наскiльки б це не звучало банально, але знову все залежить вiд того, що ми очiкуємо вiд моделi:

- Якщо наша модель має апроксимувати певну функцiю $\varphi(\mathbf{x})$ на обмеженiй множинi $\mathcal{X} \subset \mathbb{R}^r$, то ми можливо хочемо мiнiмiзувати $L^2(\mathcal{X}, \mu)$ норму рiзницi:

$$\hat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta} \in \Theta} \int_{\mathcal{X}} d\mu(\mathbf{x}) \|f(\mathbf{x}|\boldsymbol{\theta}) - \varphi(\mathbf{x})\|_2^2$$

- Можливо, ми хочемо максимізувати функцію правдоподібності:

$$\hat{\boldsymbol{\theta}} = \arg \max_{\boldsymbol{\theta} \in \Theta} \prod_{n=1}^N p(y_n | \mathbf{x}_n, \boldsymbol{\theta})$$

- Якщо модель видає ймовірністний розподіл над простором $\Omega \subset \mathbb{R}^r$, то можливо ми хочемо мінімізувати відстань Кульбака-Лейблера до заданого розподілу $\pi(\mathbf{x})$:

$$\hat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta} \in \Theta} D_{\text{KL}}(f(\mathbf{x} | \boldsymbol{\theta}) || \pi(\mathbf{x})) = \arg \min_{\boldsymbol{\theta} \in \Theta} \int_{\Omega} f(\mathbf{x} | \boldsymbol{\theta}) \log \frac{f(\mathbf{x} | \boldsymbol{\theta})}{\pi(\mathbf{x})} d\mathbf{x}$$

Приклад 1.6 (Розв'язок лінійної регресії). Наприклад, нехай ми вирішуємо задачу лінійної регресії для набору даних $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{1 \leq n \leq N}$. Нехай ми хочемо мінімізовувати функцію правдоподібності:

$$\hat{\boldsymbol{\theta}} = \arg \max_{\boldsymbol{\theta} \in \Theta} \prod_{n=1}^N p(y_n | \mathbf{x}_n, \boldsymbol{\theta}) = \arg \max_{(\mathbf{w}, \beta, \sigma^2)} \prod_{n=1}^N \mathcal{N}(y_n | \mathbf{w}^\top \mathbf{x}_n + \beta, \sigma^2) \quad (1.3)$$

Можна довести, що якщо позначити $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{m \times N}$ — матриця даних, а $\mathbf{y} = [y_1, \dots, y_N] \in \mathbb{R}^N$ — вектор маркерів, то розв'язок має вигляд:

$$\hat{\boldsymbol{\theta}} = (\mathbf{X} \mathbf{X}^\top)^{-1} \mathbf{X} \mathbf{y} \quad (1.4)$$

Проте, яку б ми теорію не використовували і які б параметризації моделей не використовували, в кінці кінців, перед нами постає наступна задача оптимізації, що розв'язується чисельно:

$$\hat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\mathcal{D} | \boldsymbol{\theta}), \quad (1.5)$$

де $\mathcal{L}(\mathcal{D} | \boldsymbol{\theta})$ — функція втрат, яка відображає як добре модель з параметрами $\boldsymbol{\theta}$ апроксимує дані \mathcal{D} . Зауважимо, що хоч, в ідеалі, ми хочемо отримати вихід як найближчий до “істинного”, але в більшості випадків,

ми не знаємо “істинного” вихідного розподілу. Тому, функція втрати, на практиці, залежить від набору даних \mathcal{D} , що подається на вхід, та від параметризації $\boldsymbol{\theta}$.

Отже, стає питання: а як обрати параметризацію? Насправді, саме в цьому питанні лежить більшість сучасних досліджень у глибокому навчанні: наприклад, у 1995 році, конволюційні нейронні мережі (Convolutional Neural Networks — CNN) прийшли на заміну повнозв’язним нейронним мережам [LB+95], а механізм уваги (Attention) у 2017 став основою багатьох NLP нейронних мереж [Vas17]. Щоб підкреслити важливість цього питання, наведемо приклад.

Приклад 1.7. Нехай ми хочемо апроксимувати залежність $y(x)$ для набору $\mathcal{D} = \{(x_n, y_n)\}_{1 \leq n \leq N}$, причому навіть знаючи вибірку, ми не маємо уяви, яка має бути залежність $y(x)$. За невідомими причинами, нехай ми вирішили використати модель $f(x|\boldsymbol{\theta}) = \left(\sum_{i=1}^{1000} \theta_i\right)x$ для вектору параметрів $\boldsymbol{\theta} \in \mathbb{R}^{1000}$. Хоча ця модель містить досить велику кількість параметрів, вона не може апроксимувати жодні залежності окрім лінійних. Отже навіть якщо залежність y від x квадратична, що є відносно простою залежністю, модель не зможе її апроксимувати, незважаючи на велику кількість параметрів.

1.3. Основні задачі машинного навчання

Отже, на практиці, ми хочемо мати як можна менше параметрів, але при цьому модель повинна бути достатньо гнучкою, щоб апроксимувати будь-яку залежність. Інакшими словами, модель має апроксимувати як можна більш широкий клас функцій.

Таким чином, підсумуємо, перед якими проблемами стоїть дослідник у

глибокому навчанні. Ми виділили три основні проблеми, які можна сформулювати наступним чином:

1. **Проблема статистики/ймовірності/узагальнення:** маючи лише набір даних \mathcal{D} , не знаючи істинного розподілу чи функції, чи достатньо добре функція втрати \mathcal{L} відображає степінь наближення моделі до істинної функції або розподілу?
2. **Проблема оптимізації:** маючи функцію втрати \mathcal{L} , наскільки точно і чи взагалі можливо знайти оптимальні параметри θ для мінімізації функції втрати?
3. **Проблема апроксимації:** яка найкраща і чи взагалі існує така параметризація моделі, щоб вона була достатньо гнучкою, але при цьому мала якнайменшу кількість параметрів?³

В наступному підрозділі, ми розглянемо декілька теорем, що дозволяють відповісти на третє запитання. Після цього, ми перейдемо до методології другої проблеми, а саме, до методів оптимізації.

³Мала кількість параметрів сприяє як і, очевидно, швидкості навчання та діставання передбачень, так і робить модель менш склонною до перенавчання та проблем градієнтів (стосується проблеми оптимізації).

Розділ 2

Теорія Аproxимації

Приблизно на цьому етапі, більшість літератури з машинного та, зокрема, глибокого навчання починається з фрази:

“Задамо багатошарову нейронну мережу з \star шарами, де зв’язок активацій $\mathbf{x}^{(j)}$ та $\mathbf{x}^{(j+1)}$ задається рівнянням
$$\mathbf{x}^{(j+1)} = \phi^{(j)}(\mathbf{W}^{(j)}\mathbf{x}^{(j)} + \boldsymbol{\beta}^{(j)})”$$

При цьому, зазвичай, не обґрунттовується (окрім як базової інтуїції) вибір саме такої формули для зв’язку між шарами. Ще рідше, чому така архітектура може апроксимувати широкий клас функцій. Саме тому в цьому підрозділі ми підійдемо до цього питання більш системно.

2.1. Аproxимація сігмоїдальними функціями: Теорема Цибенка

2.1.1. Постановка задачі

Один з перших результатів, що дозволяє відповісти на питання про апроксимацію функцій, був отриманий Цибенком в 1989 році у роботі [Cyb89]. Результати саме цієї роботи лежать в основі побудови перших щільних шарів у багатошарових нейронних мережах (Dense Layers): в певному вигляді, ця робота містить одну з перших архітектур, що дозволяють побудувати модель класифікації. Тому, в багатьох джерелах, теорема Цибенка отримала назву універсальної апроксимаційної теореми (Universal Approximation

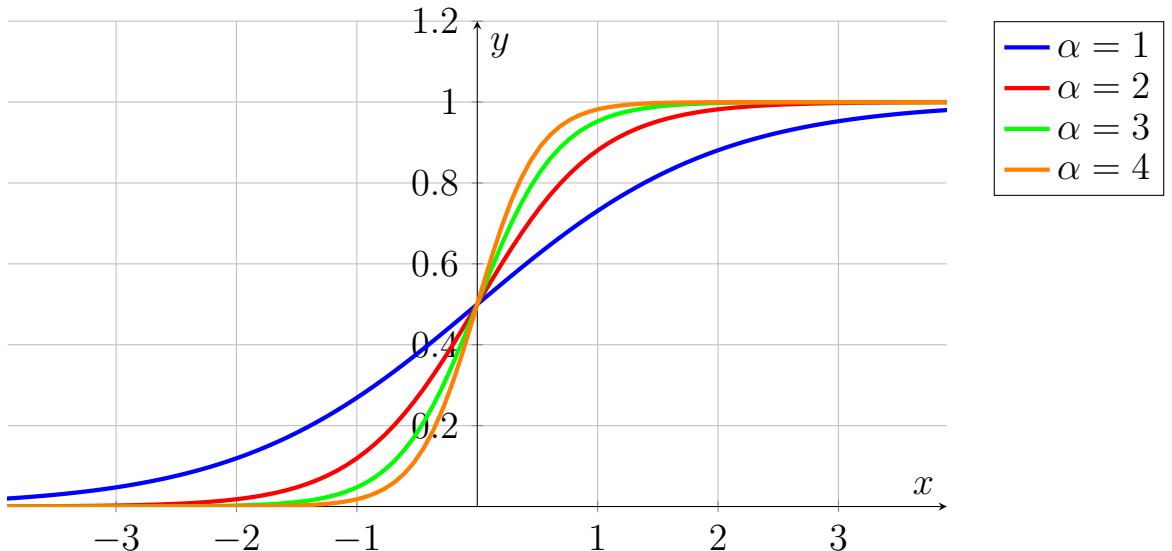


Рис. 2.1: Графіки сігмоїдальних функцій $\sigma(x|\alpha) = 1/(1 + e^{-\alpha x})$ для різних параметрів α .

Theorem). Спочатку, введемо основний клас функцій, що буде в серці нашої теореми: сігмоїдальні функції.

Означення 2.1. Сігмоїдальною функцією $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ називається функція, що задовольняє двом умовам:

$$\lim_{x \rightarrow +\infty} \sigma(x) = 1, \quad \lim_{x \rightarrow -\infty} \sigma(x) = 0. \quad (2.1)$$

Приклад 2.2. Найбільш відомою сігмоїдальною функцією є функція Логістичної регресії:

$$\sigma(x|\alpha) = \frac{1}{1 + e^{-\alpha x}}, \quad \alpha > 0. \quad (2.2)$$

Її зручність полягає у неперервності, диференційовності та зручності обчислення похідної, оскільки $\sigma' = \alpha \sigma(1 - \sigma)$. Графіки цієї функції для різних параметрів α наведені на Рис. 2.1.

Робота Цибенко присвячена на той час широкозастосованій апроксимації

функції $f : \mathbb{R}^m \rightarrow \mathbb{R}$ за допомогою наступної суми (дивись [Lip87]):

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^n \alpha_j \sigma(\mathbf{w}_j^\top \mathbf{x} + \beta_j), \quad \mathbf{w}_j \in \mathbb{R}^m, \quad \alpha_j, \beta_j \in \mathbb{R}. \quad (2.3)$$

Таким чином, ми маємо відносно просту параметризацію, що складається з $\mathcal{O}(mn)$ параметрів.

2.1.2. Узгодження з сучасною термінологією

Більш того, цю архітектуру достатньо легко описати на сучасній термінології нейронних мереж: розглянемо модель з Рис. 2.2. Діаграму читаємо наступним чином: кожен нейрон (коло) відповідає певному дійсному значенню з \mathbb{R} . Вхідний шар має m нейронів, що відповідають вхідному вектору $\mathbf{x} \in \mathbb{R}^m$. Наступний крок — це обрахунок n виразів $\Sigma_j \leftarrow \mathbf{w}_j^\top \mathbf{x} + \beta_j$ для $1 \leq j \leq n$. Ці вирази подаються на вхід сігмоїdalній функції σ , що називають *активаційною функцією*, що видає значення скритого шару $z_j = \sigma(\Sigma_j)$. Нарешті, вихідний шар це просто лінійна комбінація значень скритого шару з вагами α_j ¹:

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^n \alpha_j z_j = \sum_{j=1}^n \alpha_j \sigma(\mathbf{w}_j^\top \mathbf{x} + \beta_j).$$

Альтернативно, “на сучасний лад” зараз цю формулу більшість дослідників записали б в наступному вигляді:

$$\hat{f}(\mathbf{x}) = \boldsymbol{\alpha}^\top \sigma(\mathbf{W}\mathbf{x} + \boldsymbol{\beta}),$$

де $\boldsymbol{\alpha} \in \mathbb{R}^n$ — вектор ваг скритого шару, $\mathbf{W} \in \mathbb{R}^{m \times n}$ — матриця ваг, а $\boldsymbol{\beta} \in \mathbb{R}^n$ — вектор зсувів (biases).

Зauważення 2.3. Тут і далі запис $\sigma(\mathbf{z})$ для вектору $\mathbf{z} \in \mathbb{R}^n$ розуміємо як

¹Зараз такий б перехід на вихідний шар би назвали звичайним шаром без активаційної функції

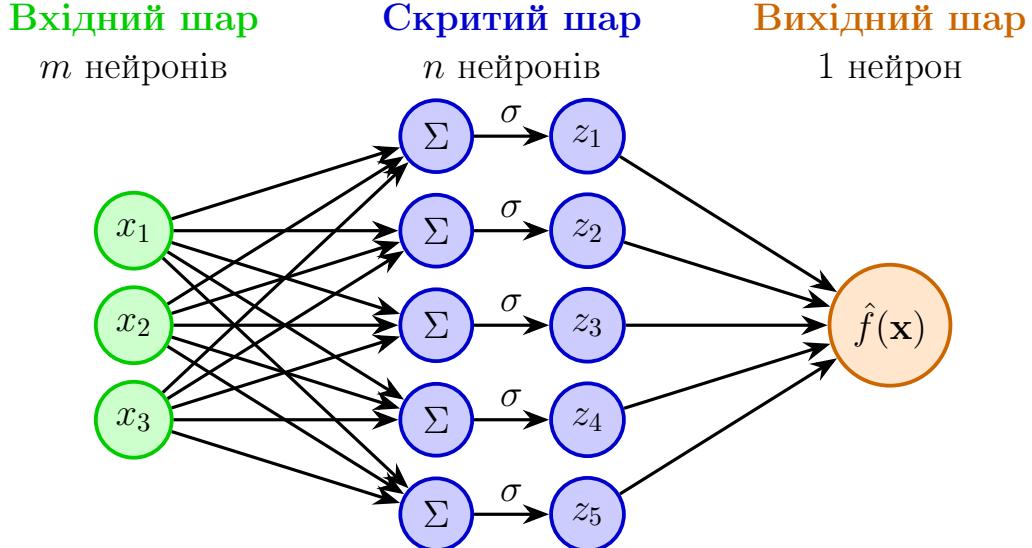


Рис. 2.2: Архітектура нейронної мережі з оригінальної роботи Цибенко [Cyb89] для випадку $m = 3$, $n = 5$. Стрілочки позначають передачу значення з відповідною вагою.

вектор $(\sigma(z_1), \dots, \sigma(z_n)) \in \mathbb{R}^n$.

2.1.3. Теореми Цибенко

Нехай $\mathcal{Q}_m = [0, 1]^m$ є m -вимірним одиничним гіперкубом. Простір неперевних функцій $f : \mathcal{Q}_m \rightarrow \mathbb{R}$ на \mathcal{Q}_m позначимо як $\mathcal{C}(\mathcal{Q}_m)$ і введемо норму функції $f \in \mathcal{C}(\mathcal{Q}_m)$ як:

$$\|f\|_{\mathcal{Q}_m} = \sup_{\mathbf{x} \in \mathcal{Q}_m} |f(\mathbf{x})|.$$

Один з головних результатів, отриманих Цибенко, наступний:

Теорема 2.4. Нехай σ будь-яка неперевна сігмоїдальна функція. Суми вигляду $\hat{f}(\mathbf{x}) = \sum_{j=1}^n \alpha_j \sigma(\mathbf{w}_j^\top \mathbf{x} + \beta_j)$ є щільними у $\mathcal{C}(\mathcal{Q}_m)$ та $L^1(\mathcal{Q}_m)$.

Інакшими словами, для будь-якої функції $f \in \mathcal{C}(\mathcal{Q}_m)$ та $\varepsilon > 0$, існує сума $\hat{f}(\mathbf{x})$ така, що:

$$(A) \quad |\hat{f}(\mathbf{x}) - f(\mathbf{x})| < \varepsilon \text{ для всіх } \mathbf{x} \in \mathcal{Q}_m.$$

$$(B) \quad \int_{\mathcal{Q}_m} |\hat{f}(\mathbf{x}) - f(\mathbf{x})| d\mathbf{x} < \varepsilon.$$

Дуже просто цю теорему можна пояснити наступним чином: для будь-якої неперервної на \mathcal{Q}_m функції f знайдеться параметризації нейронної мережі, що дозволить апроксимувати за допомогою \hat{f} цю функцію з довільною точністю. Зауважимо, що це *теорема про існування* і вона не є конструктивною: вона не дає алгоритму, який знаходить параметри $\{\alpha_j, \mathbf{w}_j, \beta_j\}_{1 \leq j \leq n}$ для довільної функції f і навіть не показує, чи можна їх знайти за допомогою алгоритмів оптимізації.

Окрім доведення теореми про апроксимацію, Цибенко також показав, що задана сума \hat{f} може апроксимувати класифікатор на \mathcal{Q}_m з довільною точністю. Більш конкретно, нехай $\mathcal{P}_0, \dots, \mathcal{P}_{C-1}$ — розбиття \mathcal{Q}_m на C підмножин (що називають *класами*). Нехай маємо функцію $f : \mathcal{Q}_m \rightarrow \{0, \dots, C-1\}$, що задана за наступним правилом:

$$f(\mathbf{x}) = j \iff \mathbf{x} \in \mathcal{P}_j.$$

Ця функція, вочевидь, не є неперервною на \mathcal{Q}_m , тому Теорему 2.4 застосувати не можна. Проте, можна показати, що і цю функцію ми можемо апроксимувати за допомогою суми \hat{f} з довільною точністю. Це дозволяє використовувати нейронні мережі для класифікації даних. Розглянемо наступну теорему.

Теорема 2.5. *Нехай σ будь-яка неперервна сігмоїдальна функція і функція f задана як вище. Тоді для будь-якої такої функції існує сума*

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^n \alpha_j \sigma(\mathbf{w}_j^\top \mathbf{x} + \beta_j)$$

та множина $\mathcal{D} \subseteq \mathcal{Q}_m$ така, що міра $\mu(\mathcal{D}) \geq 1 - \varepsilon$ та $|\hat{f}(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$ для всіх $\mathbf{x} \in \mathcal{D}$.

На відміну від Теореми 2.4, Теорема 2.5 не гарантує апроксимацію на

усьому гіперкубі \mathcal{Q}_m . Проте, з і збільшенням точності (тобто, зменшенням ε) ми можемо збільшувати міру тої області \mathcal{D} , на якій апроксимація “гарна” (себто в тій області, на якій відхилення $\hat{f}(\mathbf{x})$ від $f(\mathbf{x})$ менше за ε).

2.1.4. Практичний Приклад

Приклад 2.6. Розглянемо більш конкретний приклад. Нехай нам потрібно побудувати класифікатор для двох класів на квадраті \mathcal{Q}_2 (класифікацію з двох класів називають *бінарною*). Задамо дві області:

$$\mathcal{P}_1 := \left\{ (x_1, x_2) \in \mathcal{Q}_2 : a^2 (x_2 - 0.5)^2 - b^2 (x_1 - 0.5)^2 < 1 \right\}, \quad \mathcal{P}_0 := \mathcal{Q}_2 \setminus \mathcal{P}_1,$$

де обрано $a := 5, b := 2\sqrt{5}$. Іншими словами, наша задача — це апроксимувати індикатор функцію $f(\mathbf{x}) = \mathbb{1}[\mathbf{x} \in \mathcal{P}_1]$. Для наглядності, обидві області зображені на Рис. 2.3. В якості сігмоїдалної функції оберемо функцію логістичної регресії: $\sigma(x) := 1/(1 + e^{-x})$ та візьмемо $n = 6$ нейронів у скритому шарі. Таким чином, функція \hat{f} матиме вигляд:

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^6 \frac{\alpha_j}{1 + e^{-\mathbf{w}_j^\top \mathbf{x} + \beta_j}}, \quad \mathbf{w}_j \in \mathbb{R}^2, \quad \alpha_j, \beta_j \in \mathbb{R}.$$

Питання: якими мають бути параметри для того, щоб функція \hat{f} апроксимувала функцію f з гарною точністю? Виявляється, що достатньо непоганий результат можна отримати використовуючи наступні параметри:

$$\boldsymbol{\alpha} \approx (-8.18, 3.81, 3.91, -3.41, 5.07, 1.16),$$

$$\boldsymbol{\beta} \approx (1.13, -2.20, 1.72, 12.47, 8.46, 5.02),$$

$$\mathbf{W} \approx \begin{bmatrix} 0.06 & 4.85 & -4.39 & -11.81 & -7.59 & 14.19 \\ -6.44 & -7.67 & -6.76 & -15.67 & -10.26 & -19.17 \end{bmatrix}^\top.$$

Помітимо, що ми трошки скоротили запис, сформувавши з параметрів ве-

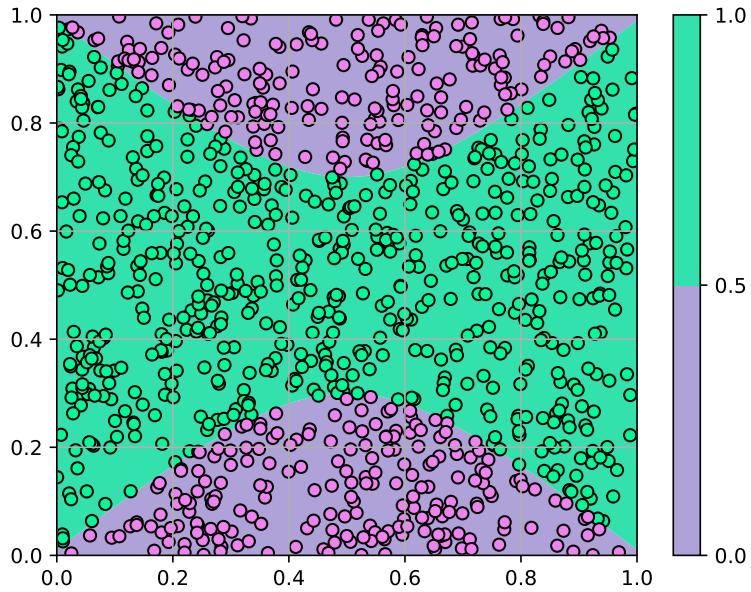


Рис. 2.3: Класи \mathcal{P}_0 та \mathcal{P}_1 на квадраті \mathcal{Q}_2 . Разом з класами, зображене набір даних $\mathcal{D} = \{(\mathbf{x}_n, \mathbb{1}(\mathbf{x}_n \in \mathcal{P}_1))\}_{1 \leq n \leq N} \subset \mathcal{Q}_2 \times \{0, 1\}$.

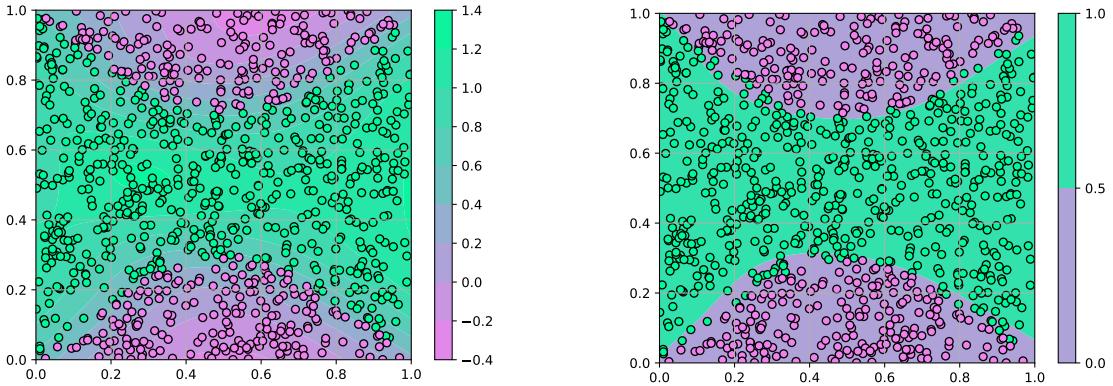
ктори та матриці, як це було зроблено у Формулі 2.1.2.

Зверніть, що на Рис. 2.3 ми також зображуємо набір даних \mathcal{D} , що складається з $N = 1000$ точок з відповідним маркуванням (бітом), що відповідає класу, до якого належить точка. Головна причина цього — мати спосіб знайти параметри моделі \hat{f} : ми можемо використати, наприклад, алгоритм градієнтного спуску для мінімізації функції втрат.

Зауваження 2.7 (Про тренування моделі). Забігаючи вперед, для підбору оптимальних параметрів ми використовували середньоквадратичну функцію втрати:

$$\mathcal{L}(\mathcal{D}|\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \left(\hat{f}(\mathbf{x}_n|\boldsymbol{\theta}) - y_n \right)^2,$$

і далі використовували алгоритм градієнтного спуску (Adam Optimizer [Kin14]) для мінімізації цього виразу відносно параметрів $\boldsymbol{\theta}$. Більше деталей наведено у Додатку А.



(а) Результат класифікації $\hat{f}(\mathbf{x})$ на (б) Бінарний результат класифікації $\mathbb{1}(\hat{f}(\mathbf{x}) > \tau)$ з порогом $\tau \approx 0.62$.

Рис. 2.4: Результати класифікації для класів \mathcal{P}_0 та \mathcal{P}_1 на квадраті \mathcal{Q}_2 .

Після тренування, результати зображені на Рис. 2.4(а). Помітимо, що вихід $\hat{f}(\mathbf{x})$ не є бінарним, але можна ввести поріг $\tau \in \mathbb{R}$ такий, що передбачення $\hat{y} := \mathbb{1}(\hat{f}(\mathbf{x}) > \tau)$ відповідає класу 1 за умови $\hat{f}(\mathbf{x}) > \tau$, а інакше — класу 0. На Рис. 2.4(б) зображені результати класифікації для $\tau = 0.62$. Як бачимо, класифікатор працює досить добре.

Також для цікавості, можна побудувати подібне зображення передбачень, але для кожного нейрону. На Рис. 2.5 зображені результати передбачень для кожного нейрона у скритому шарі.

2.1.5. Подальший розвиток архітектури Цибенко

Звичайно, що науковці не зупинились на результатах Цибенко. В подальших архітектурах, дослідники ставили багато питань, таких як:

- Що, якщо зробити кілька скритих шарів у мережі?
- Чи можна використовувати інші активаційні функції окрім сігмоїдів?
- А чи можна поєднувати два скритих шара іншим способом?

Саме тому, була введена багатошарова модель персепtronів (Multi-Layer

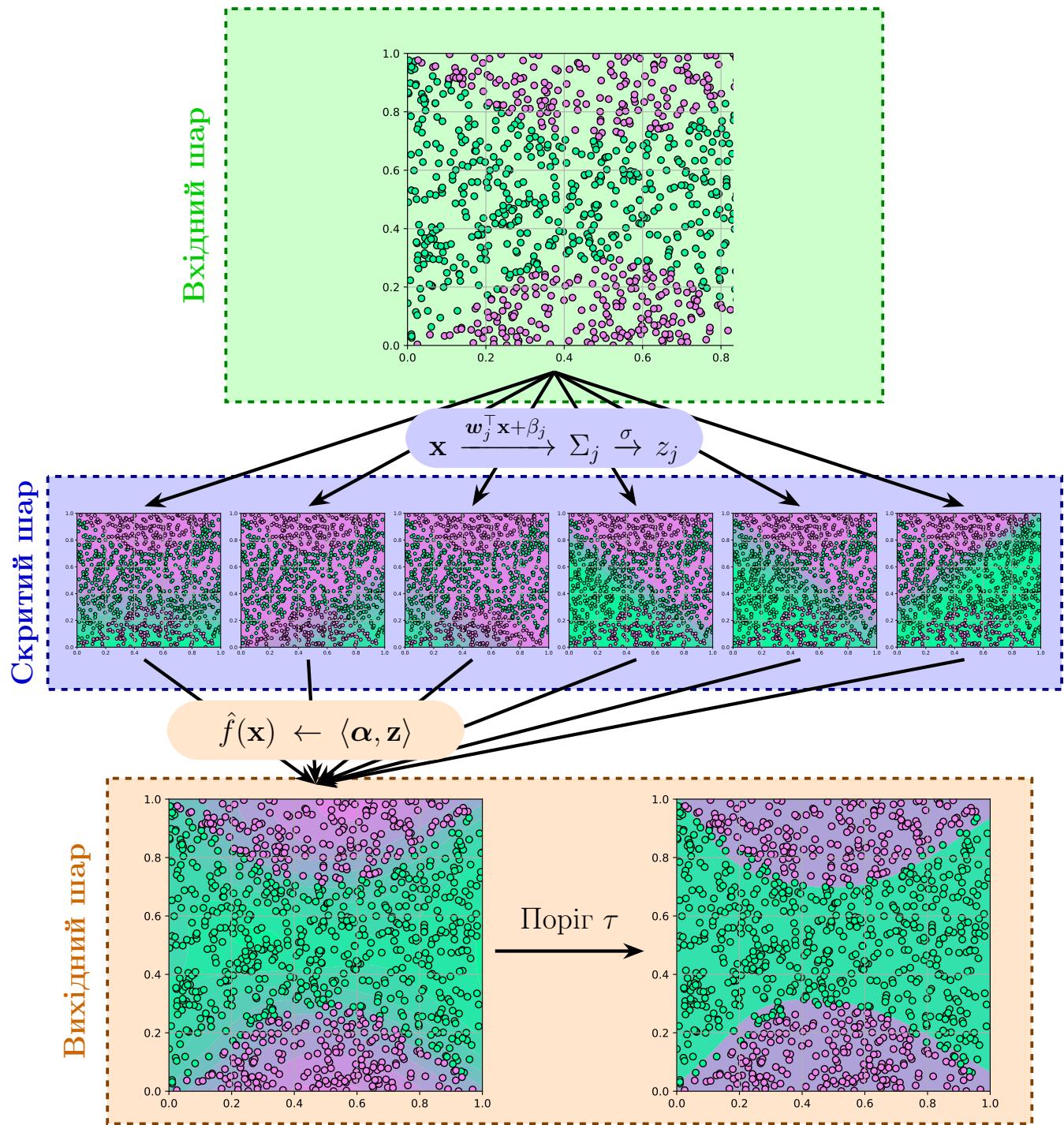


Рис. 2.5: Результати передбачень для кожного нейрону у скритому шарі.

Perceptrons — MLP), яку ми сформулюємо нижче.

Означення 2.8 (Багатошарова модель персепtronів). Нехай $\ell \in \mathbb{N}$ — кількість шарів у мережі, а $n_0, \dots, n_\ell \in \mathbb{N}$ — кількість нейронів у кожному шарі, де $\mathbf{x}^{(0)} \in \mathbb{R}^{n_0}$ відповідає вхідному шару. Тоді, для знаходження виходу $\mathbf{x}^{(\ell)} \in \mathbb{R}^{n_\ell}$, багатошарова модель персепtronів використовує наступне рекурентне правило:

$$\mathbf{x}^{(j+1)} = \phi^{(j)}(\mathbf{z}^{(j)}), \quad \mathbf{z}^{(j)} = \mathbf{W}^{(j)} \mathbf{x}^{(j)} + \boldsymbol{\beta}^{(j)}, \quad j = 0, \dots, \ell - 1,$$

де $\phi^{(j)}$ — активаційна функція у шарі j , а $\mathbf{W}^{(j)} \in \mathbb{R}^{n_{j+1} \times n_j}$ та $\boldsymbol{\beta}^{(j)} \in \mathbb{R}^{n_{j+1}}$ — матриця ваг та вектор зсуву у шарі j відповідно. Таким чином, параметризація моделі є $\boldsymbol{\theta} = \left\{ \mathbf{W}^{(j)}, \boldsymbol{\beta}^{(j)} \right\}_{0 \leq j \leq \ell - 1}$.

Навіщо більше шарів? Здавалося б, якщо ми можемо апроксимувати будь-яку функцію за допомогою одного скритого шару, то навіщо нам багатошарові моделі? Виявляється, що більше шарів дозволяють нам апроксимувати складніші функції за допомогою меншої кількості нейронів і чи-сельно знаходити їх стає простіше. Емпірично, набір правил, що описують ефективність моделі від кількості тренувальних параметрів, розміру набору даних та інших факторів, називають законами масштабування (*Scaling Laws*). Зокрема, емпірично, середнє значення функції втрати L залежить від кількості параметрів N як $L \propto N^{-\alpha}$ для певної константи $\alpha > 0$. Більш детальне дослідження від інших параметрів таких як гіперпараметри архітектури, розміру набору даних можна подивитися у джерелі [Кар+20]. Зокрема, джерело [ВН89] строго показує асимптотичну залежність між кількістю параметрів та загальнізацією моделі для задачі класифікації, а джерело [Бар94] доводить, що якщо розмір вибірки D і вхід складається з t нейронів, то при кількості параметрів $N = (D/(m \log D))^{1/2}$, то, асимптоти-

чно, L_2 норма різниці між апроксимацією та реальною функцією обмежена як $\mathcal{O}((m/D) \log D)^{1/2}$.

Навіщо інші активаційні функції? Окрім того, на практиці, логістична функція виявляється дуже незручною. Зокрема, вона відноситься до класу функцій, що називаються *ненасиченими* (або *non-saturating activation function*). Основна проблема полягає у тому, що під час навчання моделі, ми використовуємо градієнтні методи, які полагаються на значення матриці Якобіана функції втрати відносно параметрів моделі. Спрощено, моделі змінюються на мале значення $\delta\theta \approx \eta(\partial\mathcal{L}/\partial\theta)\Big|_{\theta=\theta_{\text{current}}}$. У ході обчислення цього градієнту, ми використовуємо правило ланцюга, що призводить до того, що кожен доданок у виразі містить вираз $\sigma'(z)$ у добутку. Для логістичної функції, похідна $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ і легко бачити, що як для малих, так і для великих значень z , ця похідна дуже стрімко наближається до нуля, що призводить до проблеми *вицідження градієнту* (*vanishing gradient problem*). Це означає, що градієнт функції втрати може бути дуже малим, що призводить до того, що модель не навчається. Зокрема, нижче ми наводимо популярні функції, що використовують для розв'язку цієї проблеми:

1. **ReLU** (Rectified Linear Unit): $\phi(z) = \max\{0, z\}$. Ця функція має похідну $\phi'(z) = 1(z > 0)$, тобто за додатніх значень z градієнт не виціджується. Проте, для від'ємних значень z , градієнт нульовий, через що модель може “тормозити”. Для виріження цієї проблеми, було запропоновано модифікації ReLU, такі як Leaky ReLU.
2. **Leaky ReLU**: $\phi(z) = \max\{\alpha z, z\}$ де $\alpha \in [0, 1)$ — мале значення (на практиці, порядку 10^{-3}). Ця функція має похідну $\phi'(z)\Big|_{z<0} = \alpha$, що дозволяє градієнту не виціджуватись для від'ємних значень z .

3. **ELU** (Exponential Linear Unit): $\phi(z) = \max\{0, z\} + \min\{\alpha(e^z - 1), 0\}$, де α — додатне значення. Ця функція має похідну $\phi'(z)\Big|_{z<0} = \alpha e^z$. За великих від'ємних значень z , ця функція збігається з ReLU, але має неперервну та нестрого нульову похідну для від'ємних значень z .

Інша архітектура Ще одним питанням, яке виникає, є як поєднувати шари у мережі. Поки що, логіка така: кожен нейрон i у шарі ℓ з'єднаний з кожним нейроном j у шарі $\ell + 1$ з певною вагою $w_{i,j}^{(\ell)}$ (що і утворюють матрицю ваг $\mathbf{W}^{(\ell)}$). Проте, чи дійсно нам потрібно стільки зв'язків? І чи дійсно така репрезентація здатна відобразити достатньо складні стосунки за малу кількість параметрів? Виявляється, що для певних задач можна використовувати інші архітектури. Нижче наведено два приклади, що показують проблематику зв'язків у мережі.

Приклад 2.9. Уявіть, що вам потрібно побудувати бінарну класифікацію для кольорового зображення відносно невеликого розміру — скажімо, 200×200 пікселів. Якщо у якості входу взяти кожен окремий піксель як нейрон, то кількість параметрів у першому шарі буде $200 \times 200 \times 3 = 120000$. Уявімо, що у скритий шар ми поставимо буквально 10 нейронів (на практиці, така кількість мала для досягнення хорошої точності). Таким чином, кількість параметрів у моделі буде як мінімум $120000 \times 10 = 1.2$ млн.

Приклад 2.10. Що робити, якщо розмір входу та виходу, взагалі кажучи, не фіксовані (наприклад, обробка тексту)? Звичайно, можна задати “зазделегідь” максимальний розмір входу та виходу, але окрім аномальної кількості параметрів, точність такої моделі може бути дуже низькою.

У рамках цієї курсової роботи, ми не беремось за повний і строгий опис всіх сучасних архітектур, проте навели основний фундамент для подальшого вивчення глибокого навчання. Для більш детального огляду, реко-

мендуємо звернутися до джерел [Mur22; Zha+21].

2.2. Теорема Колмогорова-Арнольда

2.2.1. Історія та Мотивація

Ще один дуже цікавий спосіб підходу до апроксимації функцій — це використання теореми Колмогорова-Арнольда. Історично, ця теорема походить з 13 задачі Гільберта, яка була запропонована на Паризькому конгресі математиків у 1900 році. Вона задає доволі провокативне питання: а чи існують справді неперервні дійснозначні багатовимірні функції? Здавалося б, доволі дивне питання, але воно по суті і описує 13 задачу і, відповідно, її розв'язок — теорему Колмогорова-Арнольда.

Більш конкретно, питання полягає у тому, чи можна будь-яку, скажімо, неперервну функцію $f : \mathcal{Q}_m \rightarrow \mathbb{R}$ апроксимувати за допомогою суми та композицій певного набору одновимірних неперервних функцій ϕ_1, \dots, ϕ_N ? Розглянемо декілька прикладів на основі [Mor20], щоб показати суть цієї задачі.

Приклад 2.11. Нехай $f : \mathcal{Q}_2 \rightarrow \mathbb{R}$ задана як $f(x, y) = 3x + 5y$. Якщо позначити $\phi_1(x) = 3x$, $\phi_2 = 5y$, то $f(x, y) = \phi_1(x) + \phi_2(y)$. Отже, маємо функцію двох змінних, проте вона може бути записана як сума двох функцій однієї змінної.

Приклад 2.12. Попередній приклад здається зовсім тривіальним. А що, якщо $f(x, y) = xy$? Помітимо наступне²:

$$f(x, y) = xy = e^{\log(x+1)+\log(y+1)} + (-x - 0.5) + (-y - 0.5).$$

²Тут і далі під записом \log розуміємо натуральний логарифм

Нехай $\phi_1(x) := e^x$, $\phi_2(x) := \log(x + 1)$, $\phi_3 := -x - 0.5$. Тоді:

$$f(x, y) = \phi_1(\phi_2(x) + \phi_2(y)) + \phi_3(x) + \phi_3(y).$$

Отже, функція $f(x, y)$ може бути записана як сума та композиція функцій однієї змінної $\phi_1(x), \phi_2(x), \phi_3(x)$.

Приклад 2.13. Нехай $f(x, y) = \sin(10e^x + y^{100})$, а також $\phi_1(x) := 10e^x$, $\phi_2(x) := y^{100}$ та $\phi_3(x) := \sin x$. Тоді $f(x, y) = \phi_3(\phi_1(x) + \phi_2(y))$, тобто так само маємо $f(x, y)$ як композицію та суму одновимірних функцій.

Отже, на основі цих прикладів можна зрозуміти суть 13 проблеми Гільберта.

Твердження 2.14 (Основна гіпотеза 13 проблеми Гільберта). Існує неперервна функція $f : \mathcal{Q}_3 \rightarrow \mathbb{R}$, що не може бути виражена як композиція та сума неперервних функцій $\phi_1, \dots, \phi_N \in \mathcal{C}(\mathbb{R}^2)$.

Знадобилося більше 50 років для того, щоб довести, що це твердження *хiбне*. У 1956 році Колмогоров довів, що функція будь-якої кількості змінних (себто, \mathcal{Q}_m може бути довільним гіперкубом) може бути записана як сума та композиція трьохвимірних функцій. У 1957, у 19 років, Арнольд показав, що три змінні можна замінити на дві, що власне і розв'язує в більш загальному вигляді 13 проблему Гільберта. Нарешті, згодом Колмогоров показав, що дві змінні можна замінити на одну, що врешті-решт і дає відому теорему Колмогорова-Арнольда.

Теорема 2.15 (Згідно джерелу [Mor20]). Для будь-якого натурального $m \geq 2$, існують неперервні функції $\phi_1, \dots, \phi_{2m+1} \in \mathcal{C}([0, 1])$ та дійсні числа $\lambda_1, \dots, \lambda_m \in \mathbb{R}$ з такою властивістю, що для будь-якої функції $f \in \mathcal{C}(\mathcal{Q}_m)$ знайдеться неперервна функція $\Phi : \mathbb{R} \rightarrow \mathbb{R}$ така, що для будь-якого $\mathbf{x} =$

$(x_1, \dots, x_m) \in \mathcal{Q}_m$ справедливо:

$$f(\mathbf{x}) = \sum_{q=1}^{2m+1} \Phi \left(\sum_{p=1}^m \lambda_p \phi_q(x_p) \right)$$

Зауваження 2.16. В цій формулі дуже багато чого цікавого! По-перше, дивує сам факт не наближеної апроксимації, а точної рівності. По-друге, важливо помітити, що функції $\phi_1, \dots, \phi_{2m+1}$ не залежать від f , але залежать від m . Це означає, що ми можемо знайти ці функції один раз, і вони будуть працювати для будь-якої функції f ! Після чого залишиться лише знайти Φ .

Здавалося б, враховуючи Зауваження 2.16, чому ми не можемо спочатку знайти ці функції, а потім використовувати їх у прикладних задачах? Річ у тому, що ніхто не гарантує, що ці функції взагалі мають бути диференційованими і тим паче неперервно диференційованими. Саме тому, подальший пошук функції Φ автоматично стає майже неможливим завданням. Тому, як ми можемо використовувати теорему Колмогорова-Арнольда для побудови нейронних мереж?

2.2.2. Мережі Колмогорова-Арнольда

Довгий час ідею теореми Колмогорова-Арнольда не пробували застосовувати до глибокого навчання через “поганість” функцій $\Phi, \phi_1, \dots, \phi_{2m+1}$. Проте, буквально у цьому році найпоширенішою темою дискусії у суспільстві розробників глибокого навчання стала робота “KAN: Kolmogorov-Arnold Networks” [Liu+24]. По суті, до моменту публікації, єдина парадигма апроксимації функцій полягала у побудові MLP мереж (та їх подальших варіацій у вигляді конволюційних, рекурентних мереж тощо), що ґрунтуються на вище описаній універсальній теоремі апроксимації 2.4. Однак, ав-

тори роботи [Liu+24] показали, що і на основі репрезентації Колмогорова-Арнольда можна побудувати нейронні мережі. Яким чином?

Робота [Liu+24] використовує оригінальну теорему Колмогорова-Арнольда [N57].

Означення 2.17 (Оригінальна теорема Колмогорова [N57]). Для будь-якого натурального $m \geq 2$, існують неперервні функції $\phi_{p,q} \in \mathcal{C}([0, 1])$ такі, що для будь-якої функції $f \in \mathcal{C}(\mathcal{Q}_m)$ знайдуться неперервні функції $\Phi_1, \dots, \Phi_{2m+1} \in \mathcal{C}(\mathbb{R})$ такі, що

$$f(x_1, \dots, x_m) = \sum_{q=1}^{2m+1} \Phi_q \left(\sum_{p=1}^n \phi_{p,q}(x_p) \right)$$

Проте, як і для Означення 2.8 MLP мереж, нам потрібно вміти узагальнювати означення для довільної кількості шарів та нейронів в кожному шарі. Робота [Liu+24] стала першою, що запропонувала таку узагальнену модель. Спочатку, наведемо що є з'єднанням в KAN мережі.

Означення 2.18. З'єднання КАН Мережі між шаром з $n_{\text{in}} \in \mathbb{N}$ нейронами (активаціями) та шаром з $n_{\text{out}} \in \mathbb{N}$ нейронами складається з матриці функцій $\Phi = \{\phi_{q,p}\}_{1 \leq p \leq n_{\text{in}}, 1 \leq q \leq n_{\text{out}}}$, де кожна функція $\phi_{q,p} : \mathbb{R} \rightarrow \mathbb{R}$ параметризується параметрами $\theta_{q,p}$. Значення нейронів (активацій) $\mathbf{x}_{\text{out}} \in \mathbb{R}^{n_{\text{out}}}$ через попередні нейрони $\mathbf{x}_{\text{in}} \in \mathbb{R}^{n_{\text{in}}}$ визначається згідно рівнянню $\mathbf{x}_{\text{out}} = \Phi \circ \mathbf{x}_{\text{in}}$, де під виразом $\Phi \circ \mathbf{x}_{\text{in}} \in \mathbb{R}^{n_{\text{out}}}$, по аналогії з матричним добутком, мається на увазі:

$$(\Phi \circ \mathbf{x}_{\text{in}})_j = \sum_{i=1}^{n_{\text{in}}} \phi_{j,i}(x_{\text{in},i}), \quad j \in \{1, \dots, n_{\text{out}}\}.$$

Отже, ми можемо дати означення безпосередньо мережі.

Означення 2.19 (Архітектура KAN). Нехай $\ell \in \mathbb{N}$ — кількість шарів у мережі, а $n_0, \dots, n_\ell \in \mathbb{N}$ — кількість нейронів у кожному шарі, де $\mathbf{x}^{(0)} \in$

\mathbb{R}^{n_0} відповідає вхідному шару. Тоді, для знаходження виходу $\mathbf{x}^{\langle \ell \rangle} \in \mathbb{R}^{n_\ell}$, архітектура КАН використовує рекурентне правило $\mathbf{x}^{\langle j+1 \rangle} = \Phi^{\langle j \rangle} \circ \mathbf{x}^{\langle j \rangle}$, $j \in \{0, \dots, \ell - 1\}$, де $\Phi^{\langle j \rangle} = \{\phi_{q,p}^{\langle j \rangle}\}_{1 \leq p \leq n_j, 1 \leq q \leq n_{j+1}}$ — матриця функцій-ваг j . В розгорнутому вигляді, нейронна мережа \hat{f}_{CAN} записується як:

$$\hat{f}_{\text{CAN}}(\mathbf{x}) = \left(\bigcirc_{j=1}^{\ell} \Phi^{\langle \ell-j \rangle} \right) \circ \mathbf{x}$$

Таке визначення може здатися доволі заплутаним, тому давайте розглянемо приклад.

Приклад 2.20 (Теорема Арнольда як частковий випадок КАН). Нагадаємо, що теорема Колмогорова-Арнольда стверджує, що будь-яка функція $f \in \mathcal{C}(\mathcal{Q}_m)$ може бути записана як

$$f(x_1, \dots, x_m) = \sum_{q=1}^{2m+1} \Phi_q \left(\sum_{p=1}^m \phi_{p,q}(x_p) \right).$$

Тоді, якщо ми визначимо дві матриці-ваги $\Phi^{\langle 0 \rangle} = \{\phi_{p,q}\}_{1 \leq p \leq m, 1 \leq q \leq 2m+1}$ та $\Phi^{\langle 1 \rangle} = \{\Phi_q\}_{1 \leq q \leq 2m+1}$ (матриця-рядок), то ми можемо записати

$$f(x_1, \dots, x_m) = \hat{f}_{\text{CAN}}(x_1, \dots, x_m) = \Phi^{\langle 1 \rangle} \circ \Phi^{\langle 0 \rangle} \circ \mathbf{x}$$

Отже, залишається лише обрати параметризацію для кожної функції $\phi_{p,q}^{\langle j \rangle}$. Оригінальна робота [Liu+24] пропонує використовувати лінійну комбінацію певної фіксованої базисної функції $\beta(x)$ та B -сплайн порядку K :

$$\phi_{p,q}^{\langle j \rangle}(x) = \omega_\beta \beta(x) + \omega_S \sum_{k=1}^K \theta_{p,q,k}^{\langle j \rangle} B_{k,K}(x),$$

де $B_{k,K}(x)$ — k -тий B -сплайн порядку K , $\theta_{p,q,k}^{\langle j \rangle}$ — параметри моделі і ω_β, ω_S — фіксовані ваги для базисної функції та B -сплайнів відповідно (що є гіперпараметрами). В роботі пропонується обрати $\beta(x) := x\sigma(x)$.

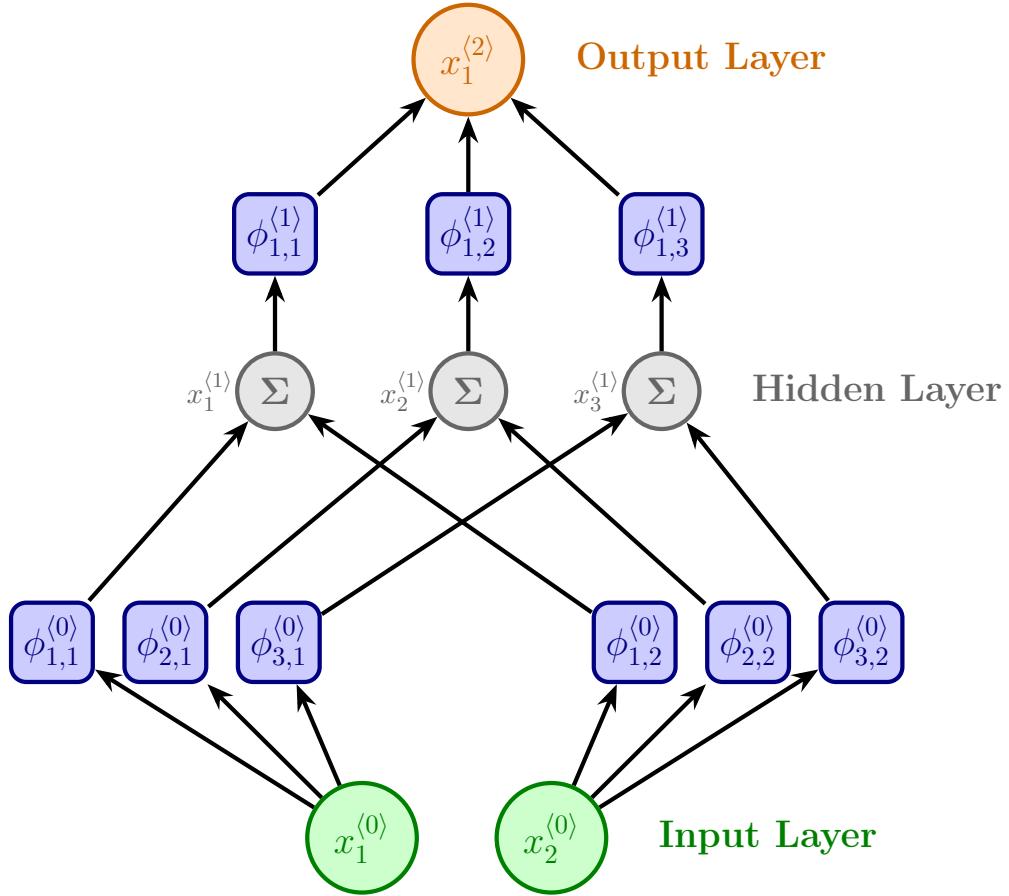


Рис. 2.6: Приклад КАН архітектури з трьома шарами з кількістю нейронів $n_0 = 2$, $n_1 = 3$, $n_2 = 1$.

Архітектура КАН для двох шарів зображена на Рис. 2.6.

Єдине, що ми ще не врахували — а чому така репрезентація взагалі дає універсальну апроксимацію на компакті \mathcal{Q}_m ? Дійсно, хоч теорема Колмогорова-Арнольда дає точну рівність, але ми поки ніяк не гарантуємо, що якщо замінити функції $\{\phi_{p,q}\}$ на B -сплайнами, то ми все одно можемо апроксимувати будь-яку функцію. Це питання досліджується в роботі [Liu+24], де показана наступна теорема.

Теорема 2.21 (Теорема апроксимації КАН). *Нехай функція f має вигляд $f = \Phi^{(\ell)} \circ \dots \circ \Phi^{(0)} \circ \mathbf{x}$, де усі функції $\phi_{p,q}^{(j)}$ є $k + 1$ разів неперервно диференційовані. Тоді, існує така константа γ , що залежить від f і функцій $\{\phi_{p,q}\}$, та існують матриці $\tilde{\Phi}^{(\ell)}, \dots, \tilde{\Phi}^{(0)}$, що складаються з B -сплайнів*

порядку k і розміром сітки n_G , що для всіх $0 \leq r \leq k$ маємо

$$\left\| f - \tilde{\Phi}^{\langle \ell \rangle} \circ \cdots \circ \tilde{\Phi}^{\langle 0 \rangle} \circ \mathbf{x} \right\|_{C^r} \leq \gamma n_G^{r-(k+1)},$$

де $\|g\|_{C^r} = \sup_{|\beta| \leq r} \sup_{\mathbf{x} \in \mathcal{Q}_m} |g^{(\beta)}(\mathbf{x})|$.

2.2.3. Порівняння з MLP архітектурою

Отже, що краще: MLP чи KAN? Наведемо деякі переваги та недоліки кожної з архітектур.

1. **Кількість інструментів.** Оскільки останні 10 років було присвячено розвитку MLP архітектур, то для них існує набагато більше інструментів, бібліотек та фреймворків. Навіть якщо KAN має потенціал, то простий налаштування та тренування може стати викликом.
2. **Інтерпретованість.** Оскільки KAN використовує B -сплайні, то вони є більш інтерпретованими, ніж MLP. Це може бути корисним для задач, де важливо зрозуміти, як саме мережа приймає рішення.
3. **Швидкість тренування.** Згідно з роботою [Liu+24], KAN мережі поки приблизно в $10\times$ повільніші за MLP під час тренування. Проте, на практиці, цей показник часто не є критичним: значно більш важливою є точність та швидкість обчислення під час обрахунку передбачення.
4. **Швидкість передбачення.** Для простоти аналізу, нехай маємо дві мережі, написані як на MLP, так і на KAN, у якої ℓ шарів, в кожному з яких n нейронів і активаційна функція має степінь d . Тоді, кількість операцій для MLP мережі буде $\mathcal{O}(\ell n^3 d)$: на кожному переході між шарами, маємо n^2 операцій для обрахунку добутку матриця-вектор, потім nd операцій для обрахунку активаційної функції над отриманим

вектором. Для КАН, кількість операцій буде $\mathcal{O}(\ell n^2 d)$: на кожному шарі, маємо n^2 функцій-ваг, кожна з яких вимагає d операцій для обрахунку. Отже, для мереж з великою кількістю шарів та нейронів, КАН може бути швидшим за MLP.

5. **Кількість параметрів.** Асимтотично, КАН мережі мають більше параметрів, ніж MLP. Дійсно, для MLP маємо $\mathcal{O}(n^2 \ell)$ параметрів, у той час як КАН ще мають зберігати параметри для кожного B -сплайну, тобто складність стає $\mathcal{O}(n^2 \ell d)$. Проте, на практиці, можливо для КАН потрібно менше нейронів та шарів для досягнення аналогічної точності.

Висновки

В цій роботі були розглянуті фундаментальні та формалізовані принципи роботи нейронних мереж. Ми конкретизували основну проблематику машинного навчання та, зокрема, яка роль нейронних мереж у вирішенні цих проблем. Ми навели визначення та теореми, що показують дві парадигми побудови архітектур нейронних мереж: мультишарові перцептрони (MLP) та мережі Колмогорова-Арнольда (КАН). Для обох парадигм ми навели теореми, що доводять універсальність цих архітектур для апроксимації довільних функцій на гіперкубі $[0, 1]^m$. Ми також розглянули проблематику вибору кількості шарів та нейронів у кожному з них, а також вибору функції активації. Ми навели приклади використання нейронних мереж для розв'язання задач класифікації і показали на практиці, що теорема Цибенко дійсно працює для складної функції.

Попереду ще багато роботи. Зокрема, КАН мережі мають великий потенціал, проте кількість досліджень у цій галузі дуже обмежена. Окрім цього, багато речей з вище описаних можна формалізувати та описати більш строго: наприклад, чому додавання шарів у MLP архітектурі не порушує властивість універсальної апроксимації або чи зміна активації, скажімо, на ReLU не змінює універсальність мережі.

Література

- [Bar94] Andrew Barron. “Approximation and Estimation Bounds for Artificial Neural Networks”. B: *Machine Learning* 14 (січ. 1994), c. 115—133. DOI: [10.1007/BF00993164](https://doi.org/10.1007/BF00993164).
- [BH89] Eric B. Baum та David Haussler. “What Size Net Gives Valid Generalization?” B: *Neural Computation* 1.1 (бер. 1989), c. 151—160. ISSN: 0899-7667. DOI: [10.1162/neco.1989.1.1.151](https://doi.org/10.1162/neco.1989.1.1.151). epriprint: <https://direct.mit.edu/neco/article-pdf/1/1/151/811817/neco.1989.1.1.151.pdf>. URL: <https://doi.org/10.1162/neco.1989.1.1.151>.
- [Bro20] Tom B Brown. “Language models are few-shot learners”. B: *arXiv preprint arXiv:2005.14165* (2020).
- [Cyb89] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. B: *Mathematics of Control, Signals and Systems* 2.4 (1989), c. 303—314. DOI: [10.1007/BF02551274](https://doi.org/10.1007/BF02551274). URL: <https://doi.org/10.1007/BF02551274>.
- [Den12] Li Deng. “The mnist database of handwritten digit images for machine learning research”. B: *IEEE Signal Processing Magazine* 29.6 (2012), c. 141—142.
- [Goo+14] Ian Goodfellow та ін. “Generative adversarial nets”. B: *Advances in neural information processing systems* 27 (2014).
- [ITD23] Guillermo Iglesias, Edgar Talavera та Alberto Díaz-Álvarez. “A survey on GANs for computer vision: Recent research, analysis and taxonomy”. B: *Computer Science Review* 48 (2023), c. 100553.

- [Kap+20] Jared Kaplan ta ih. “Scaling laws for neural language models”. B: *arXiv preprint arXiv:2001.08361* (2020).
- [Kin14] Diederik P Kingma. “Adam: A method for stochastic optimization”. B: *arXiv preprint arXiv:1412.6980* (2014).
- [Kuz+24] Oleksandr Kuznetsov ta ih. “AttackNet: Enhancing biometric security via tailored convolutional neural network architectures for liveness detection”. B: *Computers & Security* 141 (2024), c. 103828. ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2024.103828>. URL: <https://www.sciencedirect.com/science/article/pii/S0167404824001299>.
- [KZF24] Oleksandr Kuznetsov, Dmytro Zakharov ta Emanuele Frontoni. “Deep learning-based biometric cryptographic key generation with post-quantum security”. B: *Multimedia Tools and Applications* 83.19 (2024), c. 56909—56938. DOI: [10.1007/s11042-023-17714-7](https://doi.org/10.1007/s11042-023-17714-7). URL: <https://doi.org/10.1007/s11042-023-17714-7>.
- [LB+95] Yann LeCun, Yoshua Bengio ta ih. “Convolutional networks for images, speech, and time series”. B: *The handbook of brain theory and neural networks* 3361.10 (1995), c. 1995.
- [Li+24] Yang Li ta ih. “Recent developments in recommender systems: A survey”. B: *IEEE Computational Intelligence Magazine* 19.2 (2024), c. 78—95.
- [Lip87] Richard Lippmann. “An introduction to computing with neural nets”. B: *IEEE ASSP Magazine* 4 (1987), c. 4—22. URL: <https://api.semanticscholar.org/CorpusID:8275028>.
- [Liu+24] Ziming Liu ta ih. “Kan: Kolmogorov-arnold networks”. B: *arXiv preprint arXiv:2404.19756* (2024).

- [Min+23] Shervin Minaee ta ih. “Biometrics recognition using deep learning: A survey”. B: *Artificial Intelligence Review* 56.8 (2023), c. 8647—8695.
- [Mor20] Sidney Morris. “Hilbert 13: Are there any genuine continuous multivariate real-valued functions?” B: *Bulletin of the American Mathematical Society* 58 (лип. 2020), c. 1. DOI: [10.1090/bull/1698](https://doi.org/10.1090/bull/1698).
- [Mur22] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL: probml.ai.
- [N57] Kolmogorov A. N. “On the Representation of Continuous Functions of one Variable and Addition”. B: *Doklady Akademii Nauk SSSR* 144 (1957), c. 679—681. URL: <https://cir.nii.ac.jp/crid/1571980075616322176>.
- [Qin+24] Libo Qin ta ih. “Large language models meet nlp: A survey”. B: *arXiv preprint arXiv:2405.12819* (2024).
- [Red+16] Joseph Redmon ta ih. “You Only Look Once: Unified, Real-Time Object Detection”. B: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, c. 779—788. DOI: [10.1109/CVPR.2016.91](https://doi.org/10.1109/CVPR.2016.91).
- [Vas17] A Vaswani. “Attention is all you need”. B: *Advances in Neural Information Processing Systems* (2017).
- [Wan+24] Yulin Wang ta ih. “Computation-efficient deep learning for computer vision: A survey”. B: *Cybernetics and Intelligence* (2024).
- [Zha+21] Aston Zhang ta ih. “Dive into deep learning”. B: *arXiv preprint arXiv:2106.11342* (2021).

- [ZKF24] Dmytro Zakharov, Oleksandr Kuznetsov ta Emanuele Frontoni. “Unrecognizable yet identifiable: Image distortion with preserved embeddings”. B: *Engineering Applications of Artificial Intelligence* 137 (2024), c. 109164. ISSN: 0952-1976. DOI: <https://doi.org/10.1016/j.engappai.2024.109164>. URL: <https://www.sciencedirect.com/science/article/pii/S0952197624013228>.

Додаток А

Програмний код для тренування мережі Цибенко

В цьому додатку, ми наведемо програмний код для тренування мережі Цибенко на прикладі задачі класифікації. Для цього ми використаємо бібліотеку TensorFlow на мові програмування Python. Для початку, ми імпортуємо необхідні бібліотеки:

```
1 # Standard imports
2 from __future__ import annotations
3 from pathlib import Path
4
5 # Tensorflow and numpy imports
6 import tensorflow as tf
7 import numpy as np
8
9 # Matplotlib imports
10 import matplotlib.pyplot as plt
11 from matplotlib import ticker, cm
12 from matplotlib.colors import LinearSegmentedColormap
```

Далі, створюємо функції для побудови графіків та генерації набору даних:

```
1 # Selecting primary colors
2 PRIMARY_COLOR_POSITIVE = 'mediumspringgreen'
3 PRIMARY_COLOR_NEGATIVE = 'violet'
4 CMAP = LinearSegmentedColormap.from_list("Custom",
5                                         [PRIMARY_COLOR_NEGATIVE, PRIMARY_COLOR_POSITIVE], N=20)
6
6 # Picking a number of points to draw
```

```

7 DATASET_SIZE = 1000
8 RADIUS = 0.8
9
10 def get_labels(x: np.ndarray) -> np.ndarray:
11     """
12         Based on array of  $R^2$  coordinates, returns an array of bits
13             ↪ indicating
14                 whether the point is included in the region
15
16     return 0.5*(x[:,1]-0.5)**2 - 0.4*(x[:,0]-0.5)**2 < 0.02
17
18 def generate_dataset() -> np.ndarray:
19     """
20         Generates a random dataset based on the curve provided
21
22
23     x = tf.random.uniform(shape=(DATASET_SIZE, 2))
24     return x, get_labels(x)
25
26 def display_dataset(x: np.ndarray, y: np.ndarray, save_path: Path =
27     ↪ None) -> None:
28     """
29         Displays the dataset in a form of a scatterplot
30
31         Args:
32             x - an array of points in  $R^2$ 
33             y - an array of bits, marking the class of each point
34
35         # Split the dataset into two parts
36         x_positive = np.array([x for x, y in zip(x, y) if y])
37         x_negative = np.array([x for x, y in zip(x, y) if not y])
38
39         # Display two scatterplots
40         fig, ax = plt.subplots()

```

```

41     ax.set_xlim([0.0, 1.0])
42     ax.set_ylim([0.0, 1.0])
43     plt.scatter(x_positive[:,0], x_positive[:,1],
44                   color=PRIMARY_COLOR_POSITIVE, edgecolors='black')
44     plt.scatter(x_negative[:,0], x_negative[:,1],
45                   color=PRIMARY_COLOR_NEGATIVE, edgecolors='black')
45     plt.grid()
46
47     # Showing the plot and saving if needed
48     if save_path is not None:
49         plt.savefig(save_path, transparent=True)
50
51     plt.show()
52
53 def display_dataset_with_heatmap(x: np.ndarray,
54                                  y: np.ndarray,
55                                  fn: callable,
56                                  save_path: Path = None) -> None:
57     """
58     Displays the dataset in a form of a scatterplot together
59     with the heatmap plotted using fn function
60
61     Args:
62         x - an array of points in R^2
63         y - an array of bits, marking the class of each point
64         fn - function from R^2 to R, according to which the heatmap
65             ↪ is built
66         save_path - path where image is saved. Select None to omit
67             ↪ saving
68     """
69
70     # Preparing the plot
71     fig, ax = plt.subplots()
72
71     # Show the heatmap
72     x_nodes = np.linspace(0.0, 1.0, 3000)

```

```

73     y_nodes = np.linspace(0.0, 1.0, 3000)
74     xx, yy = np.meshgrid(x_nodes, y_nodes)
75     r1, r2 = xx.flatten(), yy.flatten()
76     r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
77     grid = np.hstack((r1, r2))
78     prediction = np.array(fn(grid))
79     zz = prediction.reshape(xx.shape)
80     c = plt.contourf(xx, yy, zz, cmap=CMAP)
81     fig.colorbar(c)
82
83     # Split the dataset into two parts
84     x_positive = np.array([x for x, y in zip(x, y) if y])
85     x_negative = np.array([x for x, y in zip(x, y) if not y])
86
87     # Display two scatterplots
88     ax.set_xlim([0.0, 1.0])
89     ax.set_ylim([0.0, 1.0])
90     plt.scatter(x_positive[:,0], x_positive[:,1],
91                  color=PRIMARY_COLOR_POSITIVE, edgecolors='black')
92     plt.scatter(x_negative[:,0], x_negative[:,1],
93                  color=PRIMARY_COLOR_NEGATIVE, edgecolors='black')
94     plt.grid()
95
96     # Showing the plot and saving if needed
97     if save_path is not None:
98         plt.savefig(save_path, transparent=True)
99
100    plt.show()

```

Генеруємо датасет та зберігаємо його візуалізацію:

```

1 x, y = generate_dataset()
2 display_dataset(x, y, save_path='dataset.pdf')
3 display_dataset_with_heatmap(x, y, get_labels,
4                               save_path='./classification-example.pdf')
5 # Convert array of bits to array of 0.0's and 1.0's
6 y = tf.cast(y, tf.float32)

```

Далі, головна частина: клас для специфікації архітектури мережі Цибенко та її тренування:

```

1  class CybenkoNetwork:
2      """
3          Class representing the Cybenko network
4      """
5
6      ACTIVATION = 'sigmoid'
7      INITIALIZER = 'GlorotNormal'
8      LEARNING_RATE = 0.05
9
10     def __init__(self, hidden_layer_size: int = 6, learning_rate: float
11                  ↪ = 0.05) -> None:
12         """
13             Initializes the CybenkoNetwork instance.
14
15             Args:
16                 - hidden_layer_size: number of hidden neurons (n from paper)
17                 - learning_rate: how fast to train the network
18
19             self._hidden_layer = tf.keras.layers.Dense(HIDDEN_LAYER_SIZE,
20                                              activation=CybenkoNetwork.ACTIVATION,
21                                              bias_initializer=CybenkoNetwork.INITIALIZER,
22                                              kernel_initializer=CybenkoNetwork.INITIALIZER)
23             self._alpha = tf.Variable(tf.random.normal((1,
24                                              ↪ hidden_layer_size)), name='alpha')
25             self._optimizer =
26                 ↪ tf.keras.optimizers.legacy.Adam(learning_rate=learning_rate)
27             self._mean = 0.5 # Value needed for final classification
28
29     def predict(self, x: np.ndarray) -> np.ndarray:
30         """
31             Based on the batch of inputs, gives a batch of predictions

```

```

30
31     Args:
32         x - batch of inputs
33     """
34
35     z = self._hidden_layer(x)
36     return tf.matmul(self._alpha, tf.transpose(z))
37
38 def predict_binary(self, x: np.ndarray) -> np.ndarray:
39     """
40     Predicts the class of each x value in a form of bit
41
42     Args:
43         x - batch of inputs
44     """
45
46     prediction = self.predict(x)
47     return prediction > self._mean
48
49 def train(self, x: np.ndarray, y: np.ndarray, epochs: int = 5000,
50          ↪ batch_size: int = 1024) -> None:
51     """
52     Trains the model on given dataset (x, y) with the specified
53     ↪ number of epochs
54     and batch size.
55
56     Args:
57         x, y - array of R^2 coordinates and corresponding label
58         epochs - number of epochs to train with
59         batch_size - number of pairs for each gradient iteration step
60     """
61
62     for epoch in range(epochs):
63         for offset in range(0, len(x), batch_size):
64             # Getting the batch
65             xs, ys = x[offset: offset + batch_size], y[offset:

```

```

    ↵ offset + batch_size]

64

65     with tf.GradientTape() as tape:
66
67         # Forward pass: calculating the MSE loss
68
69         loss_value = tf.reduce_mean((self.predict(xs) -
70
71             ↵ np.array([ys]))**2)

72
73         # Use the gradient tape to automatically retrieve
74         # the gradients of the trainable variables with respect
75         # to the loss.
76
77         grads = tape.gradient(loss_value, [self._alpha,
78
79             ↵ *self._hidden_layer.trainable_variables])
80
81         # Run one step of gradient descent by updating
82         # the value of the variables to minimize the loss.
83
84         self._optimizer.apply_gradients(zip(grads, [self._alpha,
85
86             ↵ *self._hidden_layer.trainable_variables]))

87
88         if (epoch + 1) % 100 == 0:
89
90             print(f'Finished epoch {epoch+1}, loss value:
91
92                 ↵ {loss_value}...')

93
94         print('Training finished!')
95
96         # Calculating the mean score for the whole dataset
97
98         # (needed further to predict the class in the binary form)
99
100        self._mean = np.mean(self.predict(x))

```

Далі, починаємо тренування:

```

1 # Initializing and training the model
2 HIDDEN_LAYER_SIZE = 6
3 model = CybenkoNetwork(hidden_layer_size=HIDDEN_LAYER_SIZE,
4
4     ↵ learning_rate=0.05)
5 model.train(x, y, epochs=5000, batch_size=1024)

```

Далі, залишається лише відобразити неперервне передбачення мережі, дискретне передбачення та передбачення проміжних значень:

```
1 display_dataset_with_heatmap(x, y, model.predict,
2                               ↵ save_path='classification-cont-prediction.pdf')
3 display_dataset_with_heatmap(x, y, model.predict_binary,
4                               ↵ save_path='classification-discrim-prediction.pdf')
5 for i in range(HIDDEN_LAYER_SIZE):
6     display_dataset_with_heatmap(x, y,
7                                   lambda x: np.array([model._hidden_layer(x)[:, i]]),
8                                   save_path=f'layer-{i+1}-prediction.pdf')
```

Нарешті, виводимо параметри моделі:

```
1 print('Weights and biases:', model._hidden_layer.trainable_variables)
2 print('alpha weights:', model._alpha)
```