

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

**ВИКОРИСТАННЯ НЕЙРОМЕРЕЖ ДЛЯ РОЗПІЗНАВАННЯ
ДАКТИЛЕМ УКРАЇНСЬКОЇ ЖЕСТОВОЇ МОВИ**

Виконав студент 4-го курсу
Задорожний Михайло Олександрович

(підпис)

Науковий керівник:

член-кореспондент НАНУ, доктор фізико-математичних
наук, професор

Крак Юрій Васильович

(підпис)

Засвідчую, що в цій роботі немає запозичень
з праць інших авторів без відповідних
посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри теоретичної
кібернетики

«___» _____ 202_ р.,

протокол № ____

Завідувач кафедри

Ю. В. Крак

(підпис)

Київ-2024

РЕФЕРАТ

Обсяг роботи: 70 сторінок, 55 ілюстрацій, 26 джерел посилань у тексті.

Ключові слова:

ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ, ПРИНЦИПИ РОБОТИ ТА БУДОВА, ЗВОРОТНЕ ПОШИРЕННЯ ПОХИБКИ, MOBILENETV3-LARGE, РОЗПІЗНАВАННЯ ДАКТИЛЕМ, ТРЕНУВАННЯ МОДЕЛІ, ГІПЕРПАРАМЕТРИ.

Об'єктом дослідження є згорткові нейронні мережі та процес розв'язування задачі розпізнавання дактилем української жестової мови з їх використанням.

Предметом дослідження є особливості будови та принципу роботи згорткових нейронних мереж, зокрема на стадії навчання, а також усі аспекти їх тренування та використання на практиці.

Метою роботи є детальний розбір будови та принципів роботи згорткових мереж, а також огляд та використання реальної архітектури у реальній задачі – розпізнаванні дактилем української жестової мови.

Методи дослідження:

Вивчення численних публікацій фахівців, опрацювання запропонованої керівником літератури, пошук додаткової інформації із інших перевірених інтернет-джерел.

Результати роботи:

Виконано детальний розбір математичного підґрунтя, будови та принципів роботи згорткових нейронних мереж, на прикладах розписано деталі їх поведінки на етапі навчання. Проаналізовано й «розібрано до гвинтиків» реальний приклад архітектури, а саме MobileNetV3-Large, та навчено модель, яка дозволяє розпізнавати дактилями української жестової мови із точністю 98.8%.

ЗМІСТ

РЕФЕРАТ	2
ЗМІСТ	3
СКРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ	4
ВСТУП	5
РОЗДІЛ 1. Будова і принцип роботи ЗНМ	7
1.1 – коротка історія зародження структури обчислень НМ.....	7
1.2 – Математичне підґрунтя сучасних архітектур.....	9
1.3 – Функції активації як ключові елементи НМ.....	10
1.4 – Згорткові нейронні мережі та їх архітектурні особливості.....	14
1.4.1 – принцип роботи шарів згортки.....	16
1.4.2 – принцип роботи шарів пулінгу.....	19
1.4.3 – композиція ЗНМ: принципи використання згортки та пулінгу	21
1.5 – Навчання ЗНМ. Метод зворотного поширення похибки	22
1.5.1 – попередні уточнення необхідних деталей.....	22
1.5.2 – принцип роботи зворотного поширення похибки.....	25
1.5.3 – проведення ЗПП на прикладі тришарового персептрона	27
1.5.4 – ЗПП для шарів згортки та пулінгу	33
РОЗДІЛ 2. Огляд архітектури мережі MobileNetV3-Large.....	41
2.1 –поверхневий огляд структури	41
2.1.1 – коротко про нормалізацію даних	42
2.1.2 – поколонковий огляд вмісту таблиці-специфікації	42
2.2 – будова блоку bottleneck.....	43
2.2.1 - коротко про роздільну по глибині згортку	44
2.2.2 – принцип роботи SE	45
2.3 – класифікатор.....	46
РОЗДІЛ 3. Навчання MobileNetV3-Large та результати.....	48
3.1 – датасет.....	48
3.2 – критерій оптимізації	50

3.3 – градієнтний оптимізатор	52
3.4 – інші гіперпараметри мережі	54
3.5 – результати навчання	56
ВИСНОВКИ	59
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ	62
ДОДАТОК А: код програми для навчання моделі MobileNetV3-Large.....	64

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

НМ – Нейронна мережа;

ЗНМ – Згорткова нейронна мережа;

ЗПП – Зворотне поширення похибки;

RGB – Red, Green, Blue; спосіб зображення картинок, що подає колір як комбінацію певних інтенсивностей червоного, зеленого та синього тонів;

MIT – Massachusetts Institute of Technology; Массачусетський інститут технологій;

LSTM – Long Short-Term Memory; довга короткострокова пам'ять;

ReLU – Rectified Linear Unit; «Обрізана» лінійна функція;

ВСТУП

Оцінка стану. Уже не вперше те, що спочатку вважалося абсолютною фантастикою, пізніше стало реальністю. Це, на мою думку, стосується і використання нейромереж для виконання інтелектуальних проблем, зокрема задачі розпізнавання жестової інформації. Якщо раніше було очевидно, що (наприклад) взаємодіяти із комп'ютером як персонажі у «Зоряних Війнах» не вийде, то нині спеціалісти стають чим далі тим більш переконаними, що вирішити проблеми такого роду цілком можливо. Тому розвиток сфери розпізнавання (та галузі ШІ загалом) стрімко набирає темп.

Сфера застосування нейромереж дуже обширна, так як за їх підтримки можна досягти якісного асистування людині при виконанні нею, без перебільшення, будь-якої роботи. Ті ж згорткові мережі можна використовувати для розпізнавання об'єктів чи якогось руку на відео – наприклад, мережа може цілодобово аналізувати камери банку та автоматично викликати поліцію, як тільки в об'єктив попадуться грабіжники. Віртуальна реальність, нині так популярна, стане все якіснішою та іще популярнішою завдяки розпізнаванню – можна буде взаємодіяти з системою без усіляких контролерів у руках, на тілі та ногах. Хірурги зможуть збільшувати чи зменшувати зображення потрібної ділянки на моніторі використовуючи жести - без постійного мешкання до нього і від нього та витрачання на це купи критично важливого часу. Схожих прикладів купа, і всі вони - тільки початок.

Підстави для виконання. Вищеописаний потенціал нейромереж в цілому та поява сучасних прикладів, що іще більше цей потенціал розкривають, зацікавили та замотивували мене «закопатися» у цей світ «з головою».

Мета роботи:

- Пізнати суть роботи нейронних мереж і дати відповідь на питання чому їх структура має саме такий вигляд;
- Детально розібрати будову та обчислення що відбуваються всередині, зокрема їх природу та послідовність;
- Зрозуміти що саме представляє собою навчання нейронної мережі з математичної точки зору та детально описати його процес;
- Розібрати принцип роботи перевіреного практикою методу зворотного поширення похибки;

- Розв'язати задачу розпізнавання дактилем української жестової мови за допомогою нейромережі - підібравши для цього хороший датасет, мережу придатних для тренування «в домашніх умовах» розмірів, визначившись із усіма необхідними для тренування «модулями» та провівши повний цикл навчання.

Об'єктом дослідження є згорткові нейронні мережі та процес розв'язування задачі розпізнавання дактилем української жестової мови з їх використанням.

Предметом дослідження є особливості будови та принципу роботи згорткових нейронних мереж, зокрема на стадії навчання, а також усі аспекти їх тренування та використання на практиці.

Методами дослідження є вивчення численних публікацій фахівців, опрацювання запропонованої керівником літератури та пошук додаткової інформації із інших перевірених інтернет-джерел.

РОЗДІЛ 1. Будова і принцип роботи ЗНМ

1.1 – коротка історія зародження структури обчислень НМ

Ідея нейронних мереж виникла, як не дивно, як модель функціонування нейронів у мозку, названа «коннекціонізмом», і використовувала зв'язані схеми для імітації розумної поведінки. Вперше вона була представлена 1943 році нейрофізіологом Уорреном МакКаллохом і математиком Уолтером Піттсом у вигляді простої електричної схеми.

Дональд Хебб продовжив цю ідею, припустивши, що нейронні шляхи зміцнюються з кожним успішним використанням, особливо між нейронами, які мають тенденцію активуватися одночасно, - таким чином починаючи довгий шлях до кількісного визначення складних процесів мозку. У 1950-х роках дослідники почали намагатися перевести ці мережі на обчислювальні системи, і перша «мережа Хебба» була успішно реалізована в МІТ у 1954 році.

Приблизно в цей же час Френк Розенблатт, психолог у Cornell Aeronautical Laboratory, працював над розумінням відносно простіших систем прийняття рішень, присутніх в очі мухи, які лежать в основі та визначають її реакцію на втечу. Намагаючись зрозуміти та кількісно оцінити цей процес, він запропонував ідею перцептрона в 1958 році, назвавши його Mark I Perceptron (рис.3-4).

Це була система з простим принципом «input-output», змодельована за ідеєю ієрархічності будови ока мухи (ієрархічність яскраво зображена на рис.2). За основу було взято нейрон МакКаллоха-Піттса (рис.1) запропонований раніше представленими Уорреном МакКаллохом та Уолтером Піттсом у їх роботі. Цей нейрон приймає входні дані, отримує зважену суму та застосовує порогову функцію до отриманого - повертає «0» (чи «-1», версії різняться, хоча суть та сама), якщо результат нижче порогового значення, і «1» в іншому випадку. Тобто, у випадку елементарного перцептрона що «імітував реакцію мухи», на вхід мережі подаються дані, а потім на виході отримуємо 1 чи 0 – що означає присутність небезпеки чи її відсутність відповідно.

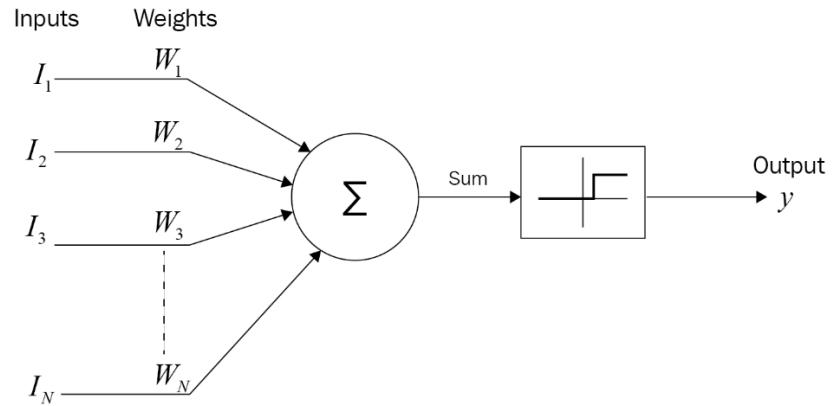


Рис. 1: Нейрон МакКаллоха-Піттса

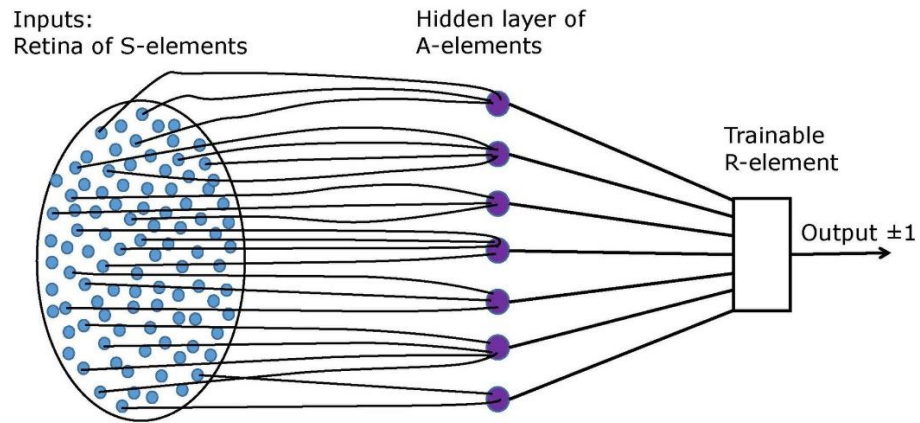


Рис. 2: "Елементарний" перцептрон Розенблатта

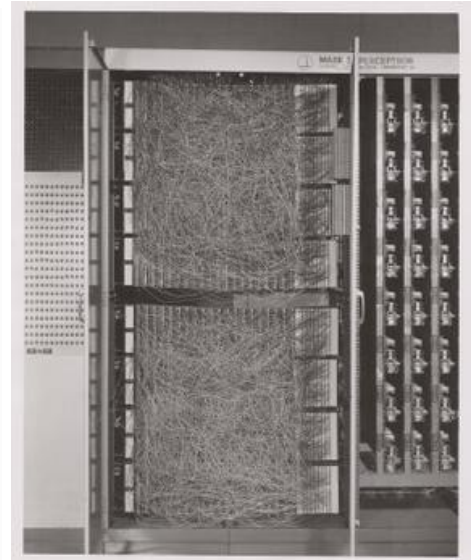
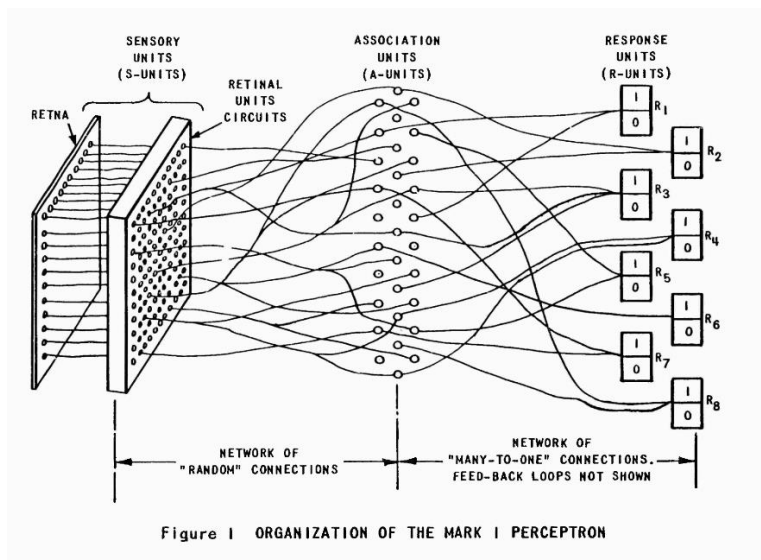


Рис. 3-4: Схема перспептрона Mark I та його реалізація

Персептрон став першою мережею, яка змогла навчатися - ваги «вивчалися» через послідовно передані вхідні дані, мінімізуючи при цьому різницю між бажаним і фактичним результатами. Така структура обчислень стала відправною точкою для численних подальших експериментів та вдосконалень, що врешті-решт привело штучні нейронні мережі до нині відомого нам стану.

1.2 – Математичне підґрунтя сучасних архітектур

На сьогоднішній день, нейрони на базі творіння МакКаллоха-Пітса чомусь дуже і дуже популярні. Так, для вирішення багатьох задач архітектури певних типів нейронів ускладнені до невпізнання. Ба більше, існує велика кількість варіацій нейронів, що використовують альтернативи зважених сум. Але це суті не змінює: чомусь у абсолютній більшості типів сучасних нейронів використовується зважена сума – а потім пропускається через якусь функцію. Виникають цілком логічні питання – Чому саме зважена сума? І для чого пропускати іще через функцію?

Історія доведення коректності даного підходу з математичної точки зору йшла рука об руку зі створенням перших мереж: у 1957 році учений А.М. Колмогоров довів теорему суперпозиції[5]. Відповідно до цієї теореми будь-яку функцію n змінних, $n \geq 2$, визначену на $E^n = [0; 1]^n$, можна подати *неперервними, монотонно зростаючими* функціями Φ_{pq} та g_q у формі:

$$f(x_1, \dots, x_n) = \sum_{q=0}^{2n} g_q \left(\sum_{p=1}^n \Phi_{pq}(x_p) \right),$$

Де Φ_{pq} та g_q - монотонно зростаючі та незалежні від f .

Пізніше до теореми багатьма іншими вченими були запропоновані покращення: було доведено що замість усіх g_q можна використовувати одну і ту ж g . Також було доведено[6], що g й Φ_{pq} не обов'язково мають бути монотонно зростаючими та що усі Φ_{pq} можна замінити на $\lambda_p \Phi_q$, де λ_p – незалежні константи. Таким чином, була отримана формула (2):

$$f(x_1, \dots, x_n) = \sum_{q=0}^{2n} g \left(\sum_{p=1}^n \lambda_p \Phi_q(x_p) \right).$$

Крім самих покращень формули точної суперпозиції, пропонувалися її апроксимації та покращення останніх. Врешті-решт, нижчеописана репрезентація однієї з апроксимацій сильно нагадує структуру персептрона:

Існує таке число H та набори чисел w_{ij} , u_i та v_i , що $f(x_1, x_2, \dots, x_n)$ можна подати формулою із похибкою не більшою за ε на всій області визначення:

$$f(x_1, x_2, \dots, x_n) = \sum_{i=1}^H v_i * \varphi(w_{i1}x_1 + w_{i2}x_2 + \dots + w_{im}x_m + u_i)$$

Яке відношення це має до нейромереж? У нашому, наприклад, випадку – класифікації жестів – ми отримуємо на вхід картинку RGB, тобто якусь кількість пікселів (ширина*висота*3). А на виході отримуємо маску яка показує наскільки жест відповідає тому чи іншому класу. Це значить, що нейромережа якимось чином вираховує свою впевненість у тому, що «картинка така-то належить класу такому-то» для кожного із класів (саме так ми і отримуємо потім маску). «Картинка така-то» – це будуть наші (x_1, x_2, \dots, x_n) , а результуюча впевненість це не що інше як якась $f(x_1, x_2, \dots, x_n)$, яку нам би хотілося знайти.

Тобто, у прикладі класифікації, по суті, задача нейромережі з точки зору математики зводиться до апроксимації «купки» функцій $f(x_1, x_2, \dots, x_n)$, кожна із яких, як член комісії, «дивиться на картинку» та показує наскільки остання належить такому-то класу.

Але усе не обмежується прикладом класифікації. $f(x_1, x_2, \dots, x_n)$ не обов'язково має виражати «впевненість». Вона може мати різний сенс. І цей сенс буде залежати від типу мережі та області її застосування. Тож, у такому випадку все так само зводиться до апроксимації мережею певної кількості функцій $f(x_1, x_2, \dots, x_n)$, просто кожна така функція буде уже не «членом комісії по класифікації», а «кимось» іншим.

1.3 – Функції активації як ключові елементи НМ

Так, задачу доведення правильності використання зваженої суми вирішено. Але взамін виникає інша нелегка задача – пошук функції $\varphi(x)$, від якої безпосередньо залежить якість апроксимації мережею.

Цю функцію було названо *функцією активації*. Базова її задача – покращувати апроксимацію. Це значить, що дана функція повинна робити діапазон можливих отриманих у мережі значень (внаслідок операцій зваженої суми) якомога більшим – що, у розумінні практичному, дозволить нейронній мережі виявляти складніші закономірності та поліпшить кінцевий результат.

У нейроні МакКаллоха-Пітса було використано найпростіший варіант, тобто поріг (рис. 5): якщо значення, отримане нейроном, надто мале, то («його здобутки не настільки важливі, і їх можна не враховувати») результатом був 0. А якщо «здобутки суттєві» - то 1. Значення порогу та результатів встановлюється творцем.

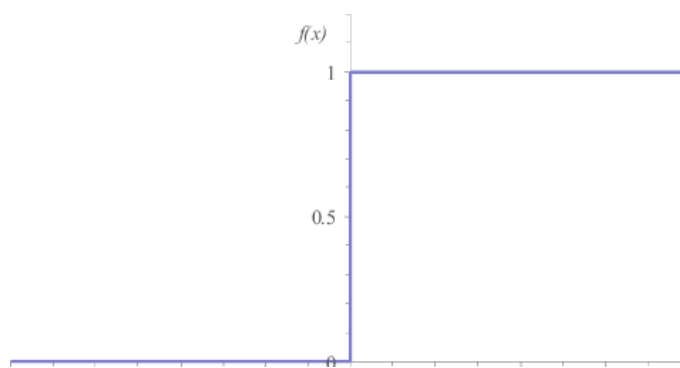


Рис. 5: Порогова функція активації

Сучасні архітектури використовують зовсім інші функції[7]. Причому вони залежать від задачі, для якої будується мережа. Крім того, проблема навчання теж залишається актуальною: якщо у випадку із першим варіантом персептрона можна було прописати методику навчання «в лоб», то для аналізу складніших паттернів цього недостатньо. По цій причині сучасні мережі використовують backpropagation – принцип зворотного поширення, що базується на обчисленні градієнтів. Про нього буде описано трохи пізніше, а зараз зупинимося на тому, що на функції активації накладаються додаткові обмеження – *неперервна диференційовність* та, у багатьох випадках, *можливість виразити похідну $\phi'(x)$ через саму $\phi(x)$* .

Наприклад, у рекурентних мережах, зокрема у архітектурі LSTM, використовується гіперболічний тангенс \tanh (рис. 6), який дозволяє добре відображати дані у діапазоні $[-1; 1]$:

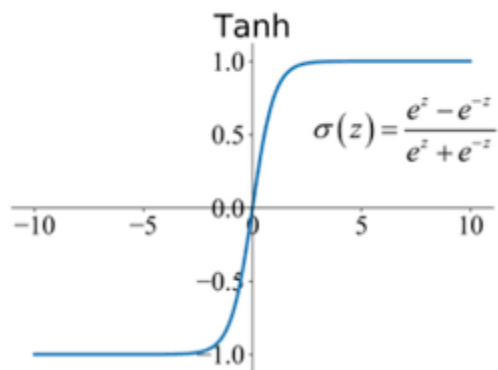


Рис. 6: Гіперболічний тангенс

Також у тих же рекурентних мережах популярністю користуються Sigmoid[7] (рис. 7), ефективний на межі $[0; 1]$ та Swish[7] (рис. 8), що рівний $x * \text{sigmoid}(x)$:

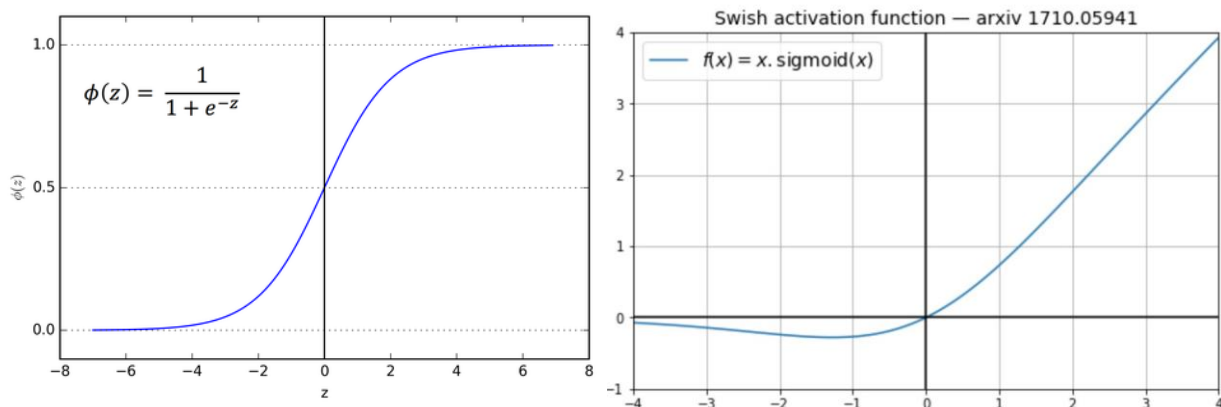


Рис. 7-8: ϕ -ї активації Sigmoid (зліва, формула на графіку) та Swish(справа)

У згорткових нейронних мережах, де обробляються картинки (чи відео) – тобто, величезні масиви даних – три вищезазначені ϕ -ї активації будуть працювати дуже і дуже повільно, оскільки обчислення експоненти потребує суттєвих зусиль. Ба більше, похідну таких функцій порахувати буде теж затратно. Тому були придумані легкі для обчислення варіанти. Так, вони містять точки у яких похідних не існує. Але, наскільки я розумію, цій проблемі було знайдено рішення, оскільки дані функції активації користуються великою популярністю.

ReLU (rectified linear unit, «обрізнана» лінійна функція) (рис. 9) є пороговою функцією активації із неймовірно швидким обчисленням. Але якість апроксимації такою функцією трохи програє вищеописаним

конкурентам (хоча там теж чимало нюансів...). Усе що вона робить – це «занулює» усі від’ємні значення.

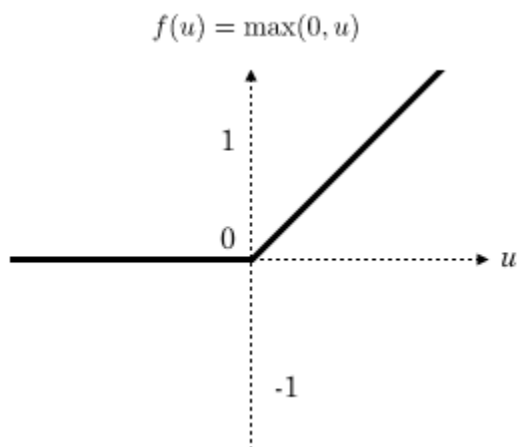


Рис. 9: ReLU

Двома найпопулярнішими модифікаціями до ReLU є Leaky ReLU (рис. 10) та ReLU6 (рис. 11), перша замість занулення від’ємних значень множить їх на малесенький коефіцієнт α . А інша встановлює верхню межу зі значенням «6».

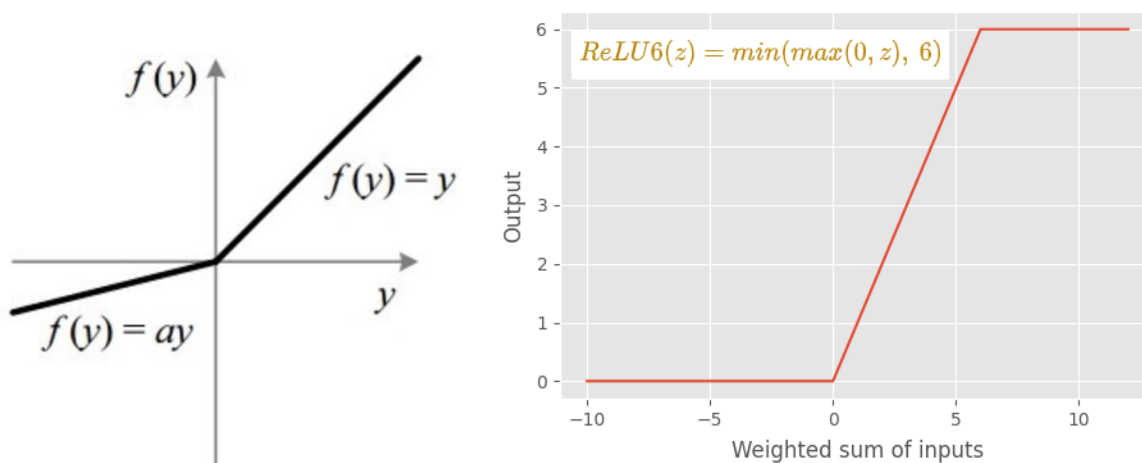


Рис. 10-11: LReLU(зліва) та ReLU6(справа)

На основі ReLU було також створено апроксимації на sigmoid та swish, названі hard sigmoid (h-sigmoid) (рис.12) та hard swish (h-swish) (рис.13) відповідно.

$$f(x) = \max\left(0, \min\left(1, \frac{(x+1)}{2}\right)\right)$$

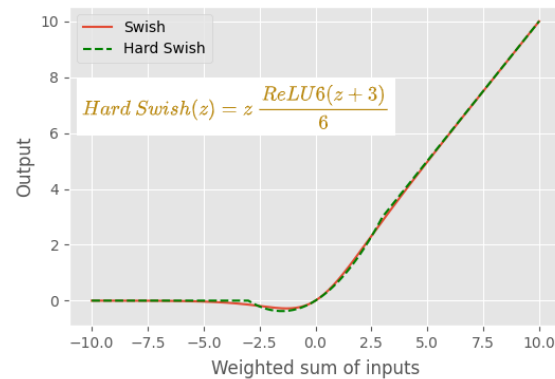
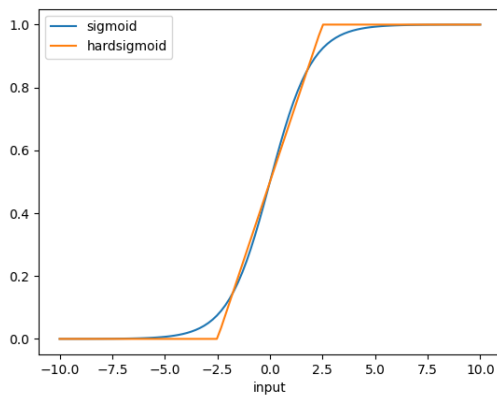


Рис. 12-13: h-sigmoid (зліва, графік + формула) та h-swish справа

Усі вищезазначені функції активації продемонстровані не просто як приклади, а як частинки архітектури MobileNetV3, яка була використана для вирішення задачі класифікації українських дактилем та розглядатиметься у другому розділі.

Важливо відмітити, що універсальної теорії вибору функції активації на даний момент не існує. Ефективність вибраної функції, звісно, значною мірою залежить від поставленої задачі та топографії архітектури, але високоякісні результати досягаються лише експериментальним шляхом.

1.4 – Згорткові нейронні мережі та їх архітектурні особливості

Якщо в попередній частині розділу розглядалися більш загальні поняття та властивості, то тематика даного параграфу уже прямує в сторону області обробки візуальних даних. Із цих пір мова йтиме про вид НМ, що створений конкретно для цього типу задач – Згорткові нейронні мережі (ЗНМ, або ж CNN).

Крім того, усі пояснення принципів роботи стосуватимуться саме зображень (архітектура для обробки відео, наприклад, матиме інший, хоча схожий, принцип роботи), так як задача розпізнавання дактилем української жестової мови зводиться до класифікації фото.

ЗНМ належать до мереж прямого поширення: дані до них подаються як у конвеєр – вхідна картинка подається в один шар, з того шару в другий, і так вглиб аж до виходу. Базова, «наївна» візуалізація мережі зображена на рис. 14.

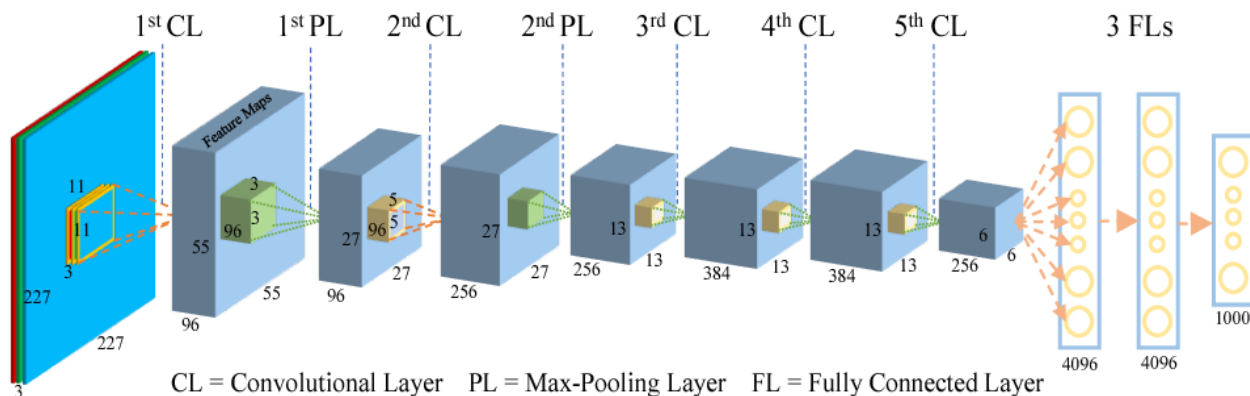


Рис.14: Візуалізація наївної архітектури ЗНМ

Перше що кидається в очі – темно-сині прямокутні паралелепіпеди. Вони відображають тензори – масиви даних (кількість вимірів масиву не обов’язково рівна трьом), над якими проводиться обробка. Також у такій архітектурі, крім класичних повноз’єднаних шарів, використовуються шари згортки (Convolution layer), зображеної жовтим кубиком що «ковзає», та пулінгу (Pooling layer), роботу якого позначено зеленим «ковзаючим» кубиком. Але навіщо? Діло в тому, що при роботі із візуальними даними виникають дві основні «стіни».

Першою із них є отримання змістовної інформації – потрібно якимось чином зрозуміти зміст картинки. Ми-от, коли дивимося на іншу людину, як розпізнаємо що це людина? Просто поглянувши на форму тіла (голова, плечі, руки, ноги) та колір шкіри? Наші можливості дійсно неймовірні: ми впізнаємо вивчені нами об’єкти миттєво, ніби «ігноруючи» масу проблем – вони можуть бути в різних позиціях у полі зору; можуть бути на різній відстані від очей, тобто мати різний масштаб; можуть бути повернуті у тривимірному просторі; можуть перебувати під різним освітленням (з однієї сторони світить сонце і колір предмету буде яскравішим, з іншої сторони може бути темнішим)... От тільки комп’ютер так «ігнорувати» все це не вміє.

Другою «стіною» виступає оптимізація обчислень, в тому числі розмір масиву (одна єдина RGB-картинка містить три мапи по $H * W$ пікселів кожна, де H і W – висота і ширина фото відповідно). Лобова обробка кожного із пікселів буде дуже і дуже довгою. Ми, наприклад, коли дивимося на чиєсь обличчя – ми ж не розглядаємо кожен міліметр шкіри та кожную волосинку на ньому, чи не так? Коли у цьому немає сенсу, ми навіть не вдивляємося. Така методика «фіксування лише ключових даних» дозволяє нам витратити значно

менше зусиль та зводити усе до суті замість «запам'ятовування волосинок на лиці». Схожа ситуація і з зображеннями.

1.4.1 – принцип роботи шарів згортки

Для подолання першої стіни із проблем було придумано застосовувати згортку (можливо, ця операція насправді має іншу назву, але нині прозвана вона саме так) зображення із якимось фільтром – маленькою (на практиці використовуються розміри від 1x1 до 7x7) матрицею.

Розглянемо принцип роботи базової згортки. Нехай ми маємо якийсь фільтр - наприклад, фільтр Собеля для апроксимації градієнту. Далі відбувається обхід фільтром зображення – це можна уявити як великий прямокутник та маленький квадрат (нехай буде 3x3 клітинки), що по ньому «їздить туди-сюди» (прямокутник це наше зображення, а квадрат – фільтр). Спочатку цей квадрат кладеться на верхній лівий кут прямокутника. Далі відбувається власне згортка - обчислення, аналогічне пошуку зваженої суми: кожна клітинка квадрата множиться на лежачу під нею відповідну клітинку прямокутника, а потім обчислюється сума добутків – отримується одне число, яке буде пікселем результуючої матриці – якогось іншого «прямокутника». Далі квадрат рухається вправо на $stride$ (крок руху) = 1 і операція згортки повторюється (рис. 15-17).

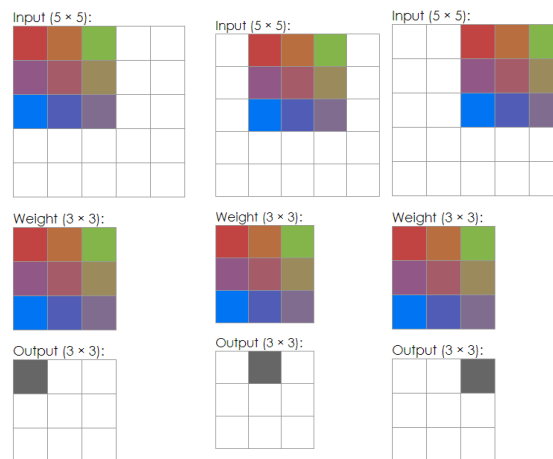


Рис.15-17: Демонстрація руху фільтру по зображенню зі $stride = 1$, верхній рядок [11]

Коли квадрат досягає краю справа, він переміщується в саму ліву позицію, але $stride=1$ рядків нижче. І далі відбувається той же прохід по рядку (рис. 18-20).

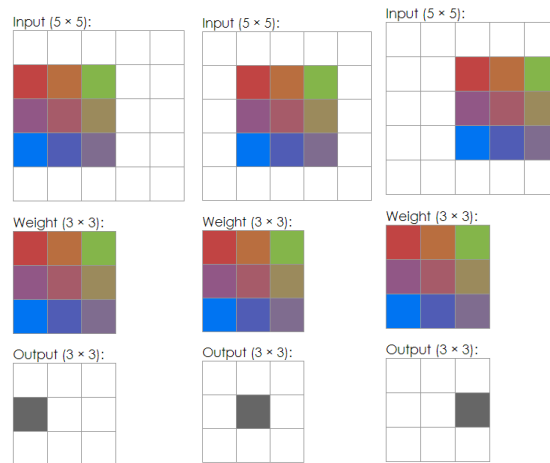


Рис.18-20. Демонстрація руху фільтра по зображенню зі $stride = 1$, середній рядок [11]

І далі усе йде аналогічно – згортка за згорткою, рядок за рядком, і до самого кінця. У випадку-ілюстрації із «зображенням» 5x5 та фільтром 3x3 залишився один рядок, прохід по якому зображено на рисунках 21-23.

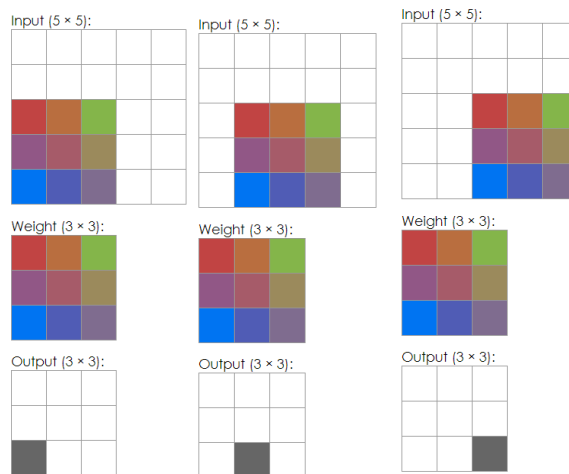


Рис.21-23. Демонстрація руху фільтра по зображенню зі $stride = 1$, нижній рядок [11]

Ось таким чином і отримується нове значення для кожної із комірок матриці Output. Але є декілька моментів:

- 1) крок $stride$ може бути більшим ніж 1. Наприклад, коли $stride = 2$ квадрат буде «скакати» по прямокутнику через 1 піксель. Коли $stride = 3$ – через два. І так далі.
- 2) Результируючий прямокутник (output на фото) менший за початковий (input на фото). У випадках коли потрібно зберегти розміри, до зображення перед проходженням фільтра додається «оббивка» нулями (padding).

3) Якщо встановити, наприклад, великий stride, то може виникнути ситуація, коли квадрат не доходить до краю прямокутника, «недочитує». Звісно ж, така поведінка не є бажаною, так як призводить до втрати даних. Така ситуація відбувається тому, що якщо квадрат перемістити далі по цьому самому рядку (з кроком stride), то він просто вилізе за межі прямокутника. Для цього випадку теж застосовується padding, тобто оббивка нулями.

Тож, така операція дозволяє нам перетворити величезний масив картинок на якийсь інший масив даних. А що означатиме цей масив даних – залежить від самого фільтра: у випадку-от фільтра Собеля це будуть приблизні значення градієнтів – тобто зміни кольорів - у кожній з областей 3×3 . А зміни кольорів, у контексті зображень, позначають не що інше як контури об'єктів. Тобто результатом використання оператора Собеля будуть контури – що уже є змістовною інформацією про картинку.

Тільки-от з'являється іще одна проблема – пошук потрібного фільтра. Якщо раніше фільтри виводилися вручну та являлися доволі абстрактними, то у 1980-х роках було запропоновано інтеграцію згортки у НМ – зі ставкою на можливість самостійного «навчання» коефіцієнтів фільтра. Це означає те, що мережа сама вчиться «втягувати» змістовну інформацію з картинки та «розробляє» свої конкретні, підходящі саме під її специфіку задачі, «методики». Тоді, при створенні мережі LeNet-5, і було вперше вжито термін Convolutional Neural Network, тобто Згорткова Нейронна Мережа.

У випадку простої одноканальної картинки операція згортки уже була повністю розписана. Але виникає питання щодо обробки 3-канальної RGB картини. Усе аналогічно – ці n (не обов'язково саме 3, може бути довільне число каналів) карт можна представити прямокутним паралелепіпедом (саме так вони на рисунку 14 і зображені). А фільтр буде уже не двовимірною матрицею, а 3-вимірною – «кубиком», у нашому RGB-випадку. І цей кубик буде за точно таким же принципом ковзати по всьому паралелепіпеду та обчислювати «попільські добутки» й додавати. Причому спочатку відбувається обчислення зваженої суми на кожному із каналів, а потім відбувається іще сумування усього отриманого в один піксель. Тобто, за один крок із кубика отримується один піксель, а при проході кубика по паралелепіпеду отримується одна карта, а не n .

Саме таку обробку даних і виконує шар згортки у нейромережі. Хоча є іще один нюанс – чому тоді на тому ж 14 рисунку після шару згортки отримано не одну карту а 96? Відповідь проста – використовується 96 різних «кубиків», тобто фільтрів. Кожен із них виконує повністю усе як описано абзацом вище. Це дозволяє сприймати отриману картинку «під різними кутами» та не впускати ніяких змістовних деталей.

1.4.2 – принцип роботи шарів пулінгу

Так як сучасні зображення мають дуже високий рівень якості та, відповідно, дуже великі розширення, в абсолютній більшості фото пікселі з однаковими інтенсивностями розташовані не поодинокі, а по групах, подекуди дуже великих. І виникає закономірне питання: а навіщо «копатися у великих групах пікселів», якщо замість таких груп можна залишити їх представників? Правильне використання такого підходу дозволяє в рази скоротити кількість обчислень. Також зменшення оброблюваного масиву дає можливість отримувати згорткою все абстрактніші ознаки – що вносить ключовий вплив на здатність мережі до генералізації. Крім того, завдяки цьому зменшується загальна кількість параметрів, що треба навчити.

Як бачимо на прикладі нижче (рис. 24), внаслідок застосування max-pooling розмір масиву зменшився від 400x400 до 200x200, тобто в 4 рази. А от зміст фото це ніяк не зачепило: ми бачимо kota, як і раніше. Ба більше – з першого погляду і якусь різницю між фото вгледіти важко.

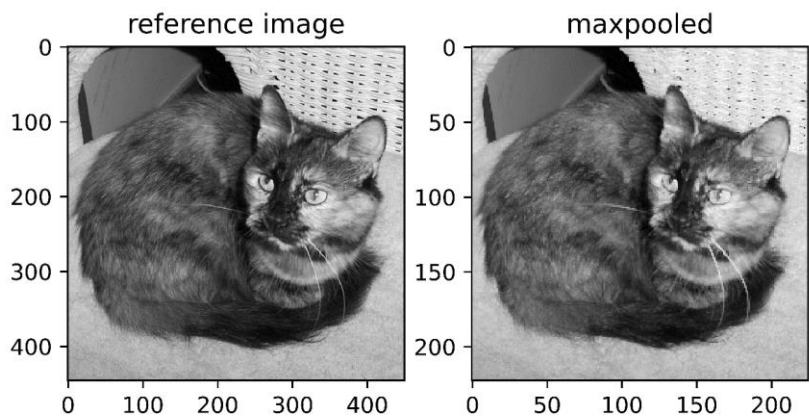


Рис. 24: Демонстрація max-pooling 2x2 на високоякісній картинці

Розглянемо детальний принцип роботи шару пулінгу. Він схожий зі згорткою у плані «ковзання квадрата по прямокутнику» - тобто квадратик (нехай буде розміром 2×2) «ковзає» із якимось кроком stride по усій мапі та на кожній своїй позиції «дивиться, що під ним». Але ніякого фільтра там немає і «попиксельні добутки» не обчислюються - квадратик просто «дивиться на групу пікселів під ним та проводить вибори представника». І потім переходить до іншої групи пікселів. І так квадрат «ковзає», аналогічно фільтру для згортки, по всій картинці. Нюанси:

- 1) Крок stride може бути довільним, але в абсолютній більшості випадків для оптимальності пулінгу використовується stride рівний розміру ядра (тобто квадрата). Якщо stride зробити меншим, то області будуть перекриватися, і відбудеться спотворення даних. А якщо більшим – то частину даних по всій картинці буде «обрізано».
- 2) «Представник» вибирається за принципом, вказаним творцем. Але найдієвішими на практиці є max-pooling (рис. 25, верх) (представником стає «найсильніший», піксель із найбільшим значенням) та average-pooling (рис. 25, низ) (шукається середнє значення усіх пікселів по області). Причому, як-от у випадку average-pooling , отриманий результат не зобов'язаний бути власне «вибраним із наявних пікселів» - це просто число отримане із певної формули, застосованої до області.

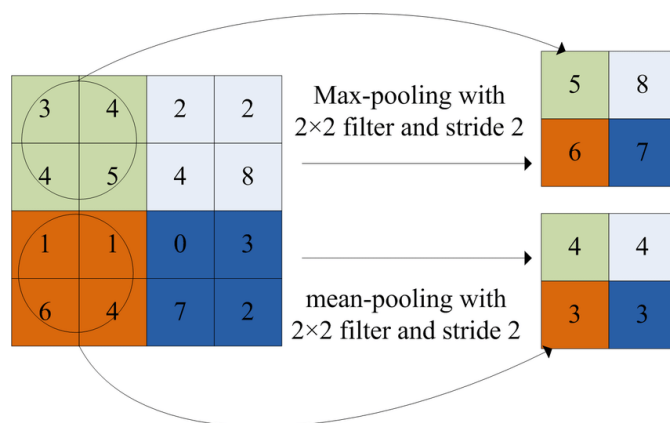


Рис. 25: Демонстрація max- та average- (mean-) пулінгів, 2×2

У випадку наявності багатьох карт – кожна із них матиме свій «квадратик», що «ковзатиме» по ній (тут швидш залежить від самої реалізації, але суть та сама). Тобто результуюча кількість карт буде збережена, при цьому

розміри кожної з них зменшаться в залежності від розміру ядра пулінгу (тобто кожного «квадратика»).

1.4.3 – композиція ЗНМ: принципи використання згортки та пулінгу

Вищеописані шари, на перший погляд, особливо нічого такого і не роблять: зменшив-от пулінг картинку, «дістала» згортка якісь особливості – і що змінилось? Діло в тому, що вся сила цих шарів розкривається у їх багаторазовому послідовному використанні.

Змоделюємо роботу якоїсь ЗНМ як приклад. На першій згортці мережа дістане із оригіналу (чи зжатої пулінгом версії) картинку якісь базові та дуже низькорівневі особливості. Далі буде виконуватися друга згортка – яка дивитиметься на отримані низькорівневі «паттерни» та складатиме із них уже трошки абстрактніші особливості. Третя згортка – особливості іще вищого рівня. І так далі по ієрархії - мережа (уявіть собі якесь фото людини в повний зріст) із ліній складає якісь форми, із тих форм іще «більші» форми, після певної кількості рівнів ознаки мають масштаби частин тіла... І на верху ієрархії із ознак «голова», «руки», «тулуб» та «ноги» уже складається ознака «людина».

Отже, згортка дозволяє мережі генералізувати побачене. Таке ієрархічне «витягування» ознак дозволяє обходити проблеми впізнання об'єкту незалежно від його положення на фото, масштабу, поворотів, та освітлення. Ось чому мережа названа згортковою.

Але згортка – не єдиний ключовий елемент. Однієї ієрархії згорток недостатньо – вона не буде працювати. Суть використання послідовності згорток полягає у забезпеченні генералізації, а генералізація сама по собі має на увазі змістовне перетворення оброблюваних даних у компактніший вигляд до тих пір, поки не залишиться декілька ознак найвищого рівня абстракції – що всередині НМ будуть представлені... буквально декількома числами. І тепер «гра починає крутитися» і навколо шарів пулінгу.

Так, згортка без оббивки із розміром ядра більшим за 1x1 теж зменшує розміри карт, але такий варіант зовсім неоптимальний – для досягнення потрібного розміру доведеться провести дуже велику кількість згорток, що обернеться завеликою кількістю шарів, параметрів для тренування та, як наслідок, проблемою вибухаючого/зникаючого градієнту - тобто адекватно

натренувати таку мережу не вийде. Це означає те, що ієрархія згорток ефективно абстрагує зображення тільки у співпраці з пулінгом.

Ближче до кінця «конвеєра» знаходяться звичайні повнозв'язні шари (як у персептрона). Коли картинку було «пройдено вздовж та впоперек» та отримано високорівневі ознаки (їх може бути не одна), усі вони передаються на повнозв'язні шари для класифікації.

Ось таким чином і працюють згорткові нейронні мережі. Розібрана у даному параграфі архітектура демонструє усі ключові особливості даного виду НМ, але є «наївною». Насправді ж, викликів набагато більше, як і видів шарів, що використовуються для протидії цим викликам.

Наприклад, для оптимізації обчислень та прискорення тренування фотографії мережі згодовують не поодинці, а якомога більшими «партіями» (batch). Для оптимального навчання мережі бажано було би обробити весь датасет одразу, але надто великі його розміри цього просто не дозволяють зробити – у сучасних ПК не вистачить пам'яті для такого розміру обчислень. Тому для навчання використовуються batch-і, менші «купки».

Варто відмітити, що попри описані вище основи побудови ЗНМ, універсальної теорії для створення ідеальної архітектури немає – усі послідовності шарів та їх параметри автори мереж отримують, проводячи численні експерименти.

1.5 – Навчання ЗНМ. Метод зворотного поширення похибки

Тепер перед нами постає іще цікавіша проблема – усю цю архітектуру треба якимось чином учить. Із розділу про математичне підґрунтя НМ слідує те, що навчання нейронної мережі (за уже «відшліфованої» архітектури із підбраною послідовністю шарів разом із функціями активації та гіперпараметрами кожного з них), по суті, зводиться до задачі оптимізації – тобто пошуку правильних коефіцієнтів (weights) що беруть участь у зважених сумах.

1.5.1 – попередні уточнення необхідних деталей

Для вирішення задач багатовимірної оптимізації популярним (відповідно, дієвим на практиці) є використання градієнтів, а точніше -

часткових похідних. Почну з нуля, продемонструвавши базову формулу часткової похідної:

$$\frac{\partial f}{\partial a_i}(a_1, \dots, a_n) = \lim_{h \rightarrow 0} \frac{f(a_1, \dots, a_i + h, \dots, a_n) - f(a_1, \dots, a_n)}{h}$$

Із неї слідує те, що часткова похідна функції від довільної кількості змінних по a_i відповідає на питання «А на скільки h - ок («ашок») зміниться функція $f(a_1, \dots, a_i, \dots, a_n)$ якщо «посунути» a_i на якесь малесеньке h уперед?». Тобто, часткова похідна показує вплив конкретної змінної на отримане із функції результуюче значення.

Часткові похідні володіють однією критично важливою у даному випадку властивістю, названу «ланцюговим правилом». Для часткових похідних це правило має вигляд [12]:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial x}; \quad \frac{\partial y}{\partial x} = \sum_{l=1}^n \frac{\partial y}{\partial q_l} \frac{\partial q_l}{\partial x}$$

На ньому, власне, і базується метод зворотного поширення похибки. Але перш ніж розбирати його принцип роботи, необхідно уточнити іще деякі важливі деталі.

У ЗНМ для задач класифікації – мережах, що навчаються з учителем - точність апроксимації «купки цільових функцій» нейронною мережею визначається певним критерієм – іншою функцією, яка отримує на вхід результати роботи мережі та показує, наскільки ці результати відрізняються відносно очікуваних (тобто тих, що надав учитель).

Також раніше було вказано, що ЗНМ є мережею прямого поширення, але не було детально розписано що саме це означає. Мережа, зазвичай, візуалізується орієнтованим графом, який показує потік даних. Причому зв'язки між нейронами нічого не обмежує – потік може мати довільну форму. Приклад такої зображено на рис. 26.

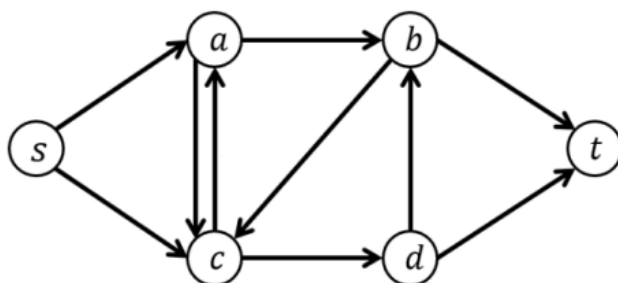


Рис. 26: Демонстрація "потоків даних" випадкової НМ

Не можу бути впевненим, але, швидш за все, наведений граф представляє собою якусь нерозумну архітектуру рекурсивної НМ (це вид мереж, у яких певні шари можуть приймати у якості входу не тільки нинішні дані, а і результати обробки попередніх, тобто працюють «у контексті»), так як містить цикли. Мережа прямого поширення (рис. 27), натомість, має інший вигляд.

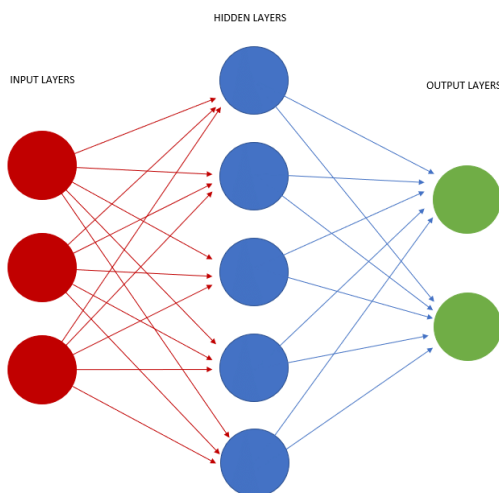


Рис. 27: Демонстрація потоків даних у мережі прямого поширення

Із візуалізації видно, що у мережі прямого поширення інформація рухається лише в одному напрямку, тобто зліва направо або ж «прямо»: із зеленого шару дані ніяк не можуть повернутися у синій чи червоний. Така архітектура має чітку, наочну послідовність виконання – ось чому ЗНМ раніше було прирівняно до «конвеєра». Крім того, після прямих обчислень та отримання результату, вона дозволяє ефективно застосовувати зворотне поширення для визначення «вкладів» кожного із коефіцієнтів мережі у похибку та подальшого покращення їх значень.

1.5.2 – принцип роботи зворотного поширення похибки

Для початку – як уже було зазначено вище, мережа прямого поширення «конвеєром» пропускає через себе якісь дані та отримується результат обробки. Наступним етапом є власне навчання, і складається воно із трьох підетапів:

- 1) Визначення помилки мережі, тобто числової відповіді на питання наскільки отриманий результат відрізняється від бажаного;
- 2) Пошук впливу кожної із ваг мережі на отримане значення помилки;
- 3) Зміна ваг в залежності від їх впливу.

За перший підетап відповідає спеціальна функція (яку ще називають критерієм), що обчислює помилку. За третій ж підетап відповідає оптимізатор, що на вхід приймає наші градієнти, тобто «впливи», та грамотним чином змінює ваги. Про виконавців першого та третього підетапів розписано «на місці» - у наступних розділах, так як вони відносно прості та пояснюються конкретикою реалізації. А от другий підетап є найскладнішим і повністю лежить на плечах зворотного поширення похибки.

Для початку запишемо задачу ЗПП конкретніше. Нехай маємо результат критерію $F(Y_{pred}, Y_{true})$, де Y_{true} – вектор констант, наданих учителем, та Y_{pred} - вектор значень, отриманих із мережі, тобто кожне значення y_{pred_i} отримане шляхом виконання операцій над вхідними даними (x_1, x_2, \dots, x_k) та вагами W_{ij} (образно, мається на увазі не конкретна матриця). Ціллю ЗПП є пошук впливу кожної із ваг на результат критерію F , тобто усіх $\frac{\partial F}{\partial W_{ij}}$.

Для демонстрації суті розв'яжемо просту не пов'язану з НМ версію такої задачі (рис. 28). Нехай ми маємо якусь функцію $f = (x + y) * z$ і пошук $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ та $\frac{\partial f}{\partial z}$ у якості мети. Розглянемо конкретний випадок (схожий на обчислення у НМ): нехай $x = -2, y = 5, z = -4$. Тоді розрахунки будуть проходити у послідовності $q = (x + y)$; $f = zq$.

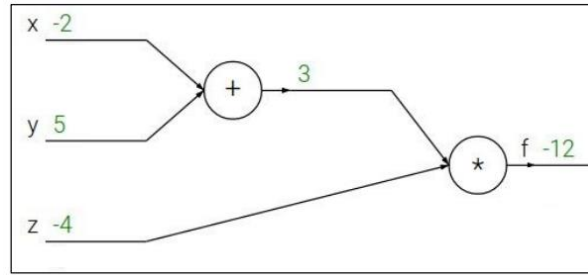


Рис. 28: Представлення обчислень $f=(x+y)*z$ «графом потоку даних» [14]

Пошук градієнтів починаємо з останнього «шару» (операції множення). Маємо $f = zq$, де q – вхід із попереднього шару, а z – «вага», та «глобальний градієнт» на вході: $\frac{\partial f}{\partial f} = 1$. Тоді із ланцюгового правила маємо:

$$\frac{\partial f}{\partial q} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial q} = 1 * z = z = -4; \quad \frac{\partial f}{\partial z} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial z} = 1 * q = q = (-2 + 5) = 3.$$

$\frac{\partial f}{\partial z}$ успішно отримано, а $\frac{\partial f}{\partial q}$ після обчислення відправляється у попередній шар, так як при прямому обчисленні q прийшло звідти. На цьому обчислення останнього шару завершено: маючи «глобальний градієнт» $\frac{\partial f}{\partial f} = 1$ на вході та обчисливши «локальні градієнти» $\frac{\partial f}{\partial q}$ і $\frac{\partial f}{\partial z}$, було обчислено «вплив» ваги z на f та попередньому шару було передано вплив його роботи на f .

Тепер, власне, попередній шар, а саме - операція додавання $q = (x + y)$. Насправді у НМ градієнти вхідних даних не обчислюються, але у цілях прикладу та демонстрації іншого шару обчислимо і їх. Тож, на вході маємо глобальний градієнт $\frac{\partial f}{\partial q} = -4$. Для обчислення $\frac{\partial f}{\partial x}$ та $\frac{\partial f}{\partial y}$ знайдемо локальні градієнти $\frac{\partial q}{\partial x}$ та $\frac{\partial q}{\partial y}$:

$$q = (x + y) \Rightarrow \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1$$

Для обчислення повних градієнтів залишилося виконати декілька множень за ланцюговим правилом:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = (-4) * 1 = -4; \quad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = (-4) * 1 = -4.$$

І все. На цьому зворотне поширення для нашої «мережі» завершено – отримано впливи кожної із ваг, що беруть участь в обчисленні критерію f .

У реальних випадках, коли мережі складаються із сотні а то й сотень шарів величезних розмірів, проведення безпомилкового обчислення градієнтів своїми руками не є можливим. Тому усе зводиться до реалізації локальних обчислень на кожному шарі, і саме по цій причині у прикладі часткова похідна бралася не одразу, а поетапно, «по шарах».

Тобто, весь алгоритм зворотного поширення похибки зводиться до локальної роботи кожного із шарів (візуалізовано на рис. 29), а саме:

- 1) отримання глобального градієнта (градієнта вихідних даних) від наступників;
- 2) обчислення локальних градієнтів;
- 3) обчислення повних градієнтів завдяки ланцюговому правилу;
- 4) поширення градієнтів вхідних даних попередникам.

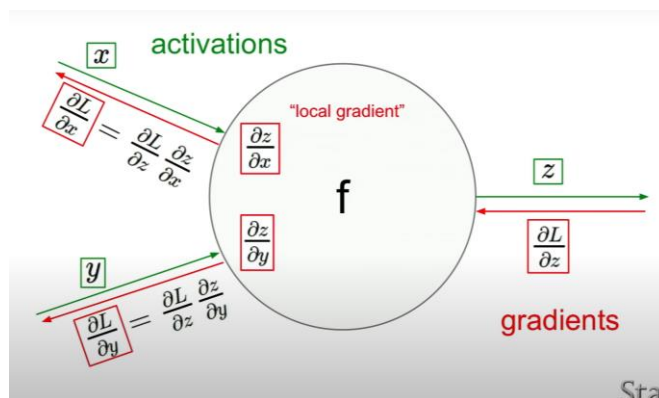


Рис. 29: Демонстрація локальної роботи шару у контексті ЗПП [15]

Даний метод навчання міг би досягти максимальної точності, якщо згодувати мережі весь датасет (вибірку даних для навчання) одразу, але, як уже було зазначено у пункті 1.4.3, сучасні розміри вибірок займають надто багато пам'яті. Тому для досягнення найкращої оптимальності ваг сучасні мережі навчаються зазначеним чином: на прямий прохід подається якась партія (batch) картинок; обчислюється середнє значення похибки із «вибірки похибок» по кожній картинці; далі з отриманим середнім проводиться процедура ЗПП та редагуються ваги.

1.5.3 – проведення ЗПП на прикладі тришарового персептрона

Розглянемо детальний приклад реальної НМ. Нехай ми маємо якусь повнозв'язну мережу, тобто простенький персептрон із декількох шарів - наприклад, 3:

- вхідний шар із вектором $X(x_1, x_2, \dots, x_k)^T$, (вектор подається стовпцем);
- прихований шар із активацією sigmoid, матрицею ваг $W^{(1)} (n \times k)$ та n нейронів;
- вихідний шар без активації, з матрицею ваг $W^{(2)} (m \times n)$ та m нейронів.

Так як це мережа прямого поширення, до неї зворотне поширення похибки успішно застосовне.

Проведення прямого поширення

Для початку – «пройдемо» мережу у звичайну сторону:

1. У прихованому шарі матимемо

$$z_i^{(1)} = W_{i1}^{(1)} x_1 + W_{i2}^{(1)} x_2 + \dots + W_{ik}^{(1)} x_k + b_i^{(1)},$$

$$a_i^{(1)} = \sigma(z_i^{(1)}), i = \overline{1, n},$$

$$\text{тобто } Z^{(1)} = W^{(1)} \times X, A^{(1)} = \sigma(Z^{(1)}) (\text{поелементно});$$

2. На виході матимемо

$$z_i^{(2)} = W_{i1}^{(2)} a_1^{(1)} + W_{i2}^{(2)} a_2^{(1)} + \dots + W_{in}^{(2)} a_n^{(1)} + b_i^{(2)},$$

$$y_{pred_i} = a_i^{(2)} = z_i^{(2)}, i = \overline{1, m},$$

$$\text{тобто } Z^{(2)} = W^{(2)} \times A^{(1)}, Y_{pred} = A^{(2)} = Z^{(2)} (\text{так як нема активації}).$$

Отримали вектор Y_{pred} із m значень.

Проведення зворотного поширення: вихідний шар

Нехай тепер маємо якийсь критерій $F(Y_{pred}, Y_{true})$. Зворотне поширення починаємо із пошуку впливу кожного значення з вектору Y_{pred} , тобто $a_i^{(2)}$, на F . Для зручності операцій із векторами, «вектор впливів» (вплив кожного елементу із вектору $A^{(2)}$ на F) будемо позначати таким чином:

$$\frac{\partial F}{\partial A^{(2)}} = \left[\frac{\partial F}{\partial a_1^{(2)}}, \frac{\partial F}{\partial a_2^{(2)}} \dots \frac{\partial F}{\partial a_m^{(2)}} \right]$$

$\frac{\partial F}{\partial a_i^{(2)}}$ обчислюються «в лоб» із критерію так як напряду присутні у його формулі.

Далі цікавіше - починаємо прохід останнього шару. Ми уже знаємо градієнт результату (у даному випадку це $\frac{\partial F}{\partial A^{(2)}}$), але виникає питання: як саме

вирахувати вплив ваг $W^{(2)}$, вільних членів $B^{(2)}$ та кожного із входів ($A^{(1)}$) на значення критерію?

Відповіддю є те саме ланцюгове правило (показано на прикладі W):

$$\frac{\partial F}{\partial W^{(2)}} = \frac{\partial F}{\partial A^{(2)}} \frac{\partial A^{(2)}}{\partial W^{(2)}}$$

Перший множник нам відомий – це раніше обчислений градієнт результату. А інший називається «локальним градієнтом». Його можна іще розкласти:

$$\frac{\partial A^{(2)}}{\partial W^{(2)}} = \frac{\partial A^{(2)}}{\partial Z^{(2)}} \frac{\partial Z^{(2)}}{\partial W^{(2)}}, \text{ де } \frac{\partial A^{(2)}}{\partial Z^{(2)}} = \left[\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}, \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \dots \frac{\partial a_m^{(2)}}{\partial z_m^{(2)}} \right],$$

причому $\frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} = \frac{\partial z_i^{(2)}}{\partial z_i^{(2)}} = 1.$

Таким чином ми можемо спокійно обчислити $\frac{\partial A^{(2)}}{\partial Z^{(2)}}$ і перенести його у «глобальний градієнт», спростивши пошук $\frac{\partial F}{\partial W^{(2)}}$ до знайдення $\frac{\partial Z^{(2)}}{\partial W^{(2)}}$.

На наступному кроці маємо складний випадок, тобто матрицю $W^{(2)}(m \times n)$, вектор $Z^{(2)}$ та необхідність пошуку $\frac{\partial Z^{(2)}}{\partial W^{(2)}}$. Якщо робити це «в лоб», тобто для кожного $z_i^{(2)}$ та $W_{jk}^{(2)}$ шукати $\frac{\partial z_i^{(2)}}{\partial W_{jk}^{(2)}}$, то отримаємо тривимірний яacobian $J(m \times (m \times n))$, що явно вказує на завелику кількість обчислень.

Але конкретика архітектур прямого поширення приховує один цікавий трюк. Для його реалізації розглянемо одну із похідних як приклад. Нехай $i=3$, $j=7$, $k=8$ (прошу не плутати k із розміром вхідного шару). Знайдемо $\frac{\partial z_3^{(2)}}{\partial W_{78}^{(2)}}$, розпочавши із формули прямого обчислення $z_3^{(2)}$:

$$z_3^{(2)} = W_{31}^{(2)} a_1^{(1)} + W_{32}^{(2)} a_2^{(1)} + \dots W_{38}^{(2)} a_8^{(1)} + \dots + W_{3n}^{(2)} a_n^{(1)} + b_3^{(2)}.$$

Як бачимо, для обчислення $z_3^{(2)}$ використовується лише третій рядок матриці $W^{(2)}$. Тобто $\frac{\partial z_3^{(2)}}{\partial W_{78}^{(2)}} = 0$. Аналогічним чином перебравши усі можливі комбінації в яacobіані, прийдемо до того, що:

$$J_{i,j,k} = \frac{\partial z_i^{(2)}}{\partial W_{jk}^{(2)}}, i = \overline{1, m}, j = \overline{1, m}, k = \overline{1, n};$$

$$J_{i,i,k} = a_k^{(1)}; \text{ решта значень рівні } 0.$$

Чому $a_k^{(1)}$? На прикладі обчислення $z_3^{(2)}$ чітко видно, що похідна $\frac{\partial z_i^{(2)}}{\partial W_{ik}^{(2)}} = a_k^{(1)}$, причому незалежно від рядка i . Тобто, якщо означимо матрицю $G_{i,k} = J_{i,i,k}$ розміром $(m \times n)$, то усі рядки G_i будуть абсолютно однаковими та, ба більше, мати ті ж значення, що й $A^{(1)}$. Але вони розміщені рядками, а $A^{(1)}$ – колонка. Значить, для представлення треба використати транспоновану версію $A^{(1)}$ - $A^{(1)T}$. Таким чином, $\frac{\partial Z^{(2)}}{\partial W^{(2)}}$ із тривимірного якобіана зводиться до змістовного подання одним єдиним рядковим вектором довжиною n - $A^{(1)T}$.

На цьому пошук $\frac{\partial F}{\partial W^{(2)}}$ завершено – його можна отримати перемноживши наявний глобальний градієнт $\frac{\partial F}{\partial Z^{(2)}} = \frac{\partial F}{\partial A^{(2)}} \frac{\partial A^{(2)}}{\partial Z^{(2)}}$ із локальним $\frac{\partial Z^{(2)}}{\partial W^{(2)}}$, тобто $A^{(1)T}$. Конкретика способу перемноження слідує із розмірностей та вмісту. Так як $\frac{\partial F}{\partial Z^{(2)}}$ є вектором розміром m , що містить значення $\frac{\partial F}{\partial z_i^{(2)}}$ та представлений стовпчиком, а $A^{(1)T}$ є представленим рядком вектором довжини n , що містить значення $\frac{\partial z_i^{(2)}}{\partial W_{ik}^{(2)}}$ у стовпчику k , то необхідно кожен з m елементів $\frac{\partial F}{\partial Z^{(2)}}$ перемножити на кожен з n елементів $A^{(1)T}$ отримавши у результаті необхідну $(m \times n)$ матрицю елементів $\frac{\partial F}{\partial W_{ik}^{(2)}}$.

Наступним кроком є пошук $\frac{\partial F}{\partial B^{(2)}}$, що повністю аналогічним чином зводиться до обчислення чергового локального градієнта - $\frac{\partial Z^{(2)}}{\partial B^{(2)}} = \left[\frac{\partial z_1^{(2)}}{\partial b_1^{(2)}}, \frac{\partial z_2^{(2)}}{\partial b_2^{(2)}} \dots \frac{\partial z_m^{(2)}}{\partial b_m^{(2)}} \right]$. Як бачимо, такі цепочки пошуків часткових похідних зводяться до повністю локальних обчислень – тобто обчислення локальних градієнтів за наявного глобального.

Із формули обчислень:

$$z_i^{(2)} = W_{i1}^{(2)} a_1^{(1)} + W_{i2}^{(2)} a_2^{(1)} + \dots + W_{in}^{(2)} a_n^{(1)} + 1 * b_i^{(2)}$$

Маємо: $\frac{\partial z_i^{(2)}}{\partial b_i^{(2)}} = 1$. Тобто $\frac{\partial Z^{(2)}}{\partial B^{(2)}} = [1, 1, \dots, 1]$, та $\frac{\partial F}{\partial B^{(2)}}$ буде матиме ті значення, що і $\frac{\partial F}{\partial Z^{(2)}}$.

Останнім локальним градієнтом, що необхідно знайти у вихідному шарі, є $\frac{\partial Z^{(2)}}{\partial A^{(1)}}$. Так як кожне $a_j^{(1)}$ впливає на кожне $z_i^{(2)}$, матимемо $(m \times n)$ матрицю. Із формули:

$$z_i^{(2)} = W_{i1}^{(2)} a_1^{(1)} + W_{i2}^{(2)} a_2^{(1)} + \dots + W_{in}^{(2)} a_n^{(1)} + 1 * b_i^{(2)},$$

Маємо: $\frac{\partial z_i^{(2)}}{\partial a_j^{(1)}} = W_{ij}^{(2)}$. А це значить, що результуюча $(m \times n)$ матриця - не що інше, як $W^{(2)}$: $\frac{\partial Z^{(2)}}{\partial A^{(1)}} = W^{(2)}$.

Усі градієнти на даному шарі знайдено, тепер обчислений $\frac{\partial F}{\partial A^{(1)}}$ (множенням «колонки» $\frac{\partial F}{\partial Z^{(2)}}$ на матрицю $W^{(2)}$) передається у попередній шар, так як у прямому поширенні $A^{(1)}$ було отримано саме звідти.

Проведення зворотного поширення: прихований шар

Як і в попередньому випадку, маємо глобальний градієнт:

$$\frac{\partial F}{\partial A^{(1)}} = \left[\frac{\partial F}{\partial a_1^{(1)}}, \frac{\partial F}{\partial a_2^{(1)}} \dots \frac{\partial F}{\partial a_m^{(1)}} \right]$$

Але у даному шарі присутня ф-я активації:

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

А тому:

$$\frac{\partial F}{\partial A^{(1)}} = \frac{\partial F}{\partial A^{(1)}} \frac{\partial A^{(1)}}{\partial Z^{(1)}}, \quad \frac{\partial A^{(1)}}{\partial Z^{(1)}} = \frac{\partial \sigma(Z^{(1)})}{\partial Z^{(1)}} = \left[\frac{\partial \sigma(z_i^{(1)})}{\partial z_i^{(1)}}, i = \overline{1, n} \right]$$

У випадку сигмоїда, тобто «хорошого» варіанту функції активації, $\frac{d\sigma(x)}{dx}$ нескладно виражається через $\sigma(x)$:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

З чого випливає те, що:

$$\frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} = \frac{\partial \sigma(z_i^{(1)})}{\partial z_i^{(1)}} = \sigma'(z_i^{(1)}) = (1 - \sigma(z_i^{(1)})) * \sigma(z_i^{(1)}) = (1 - a_i^{(1)}) * a_i^{(1)}.$$

А, отже, маємо:

$$\frac{\partial A^{(1)}}{\partial Z^{(1)}} = [(1 - a_i^{(1)}) * a_i^{(1)}, i = \overline{1, n}]$$

Тепер, коли обчислено $\frac{\partial A^{(1)}}{\partial Z^{(1)}}$ та отримано $\frac{\partial F}{\partial Z^{(1)}}$ шляхом поелементного помноження $\frac{\partial F}{\partial A^{(1)}}$ на $\frac{\partial A^{(1)}}{\partial Z^{(1)}}$, залишається обчислити локальні градієнти, тобто $\frac{\partial Z^{(1)}}{\partial W^{(1)}}$ та $\frac{\partial Z^{(1)}}{\partial B^{(1)}}$ аналогічними до розібраних у попередньому шарі способами.

У випадку $\frac{\partial Z^{(1)}}{\partial B^{(1)}}$ усе дуже просто:

$$z_i^{(1)} = W_{i1}^{(1)} x_1 + W_{i2}^{(1)} x_2 + \dots + W_{ik}^{(1)} x_k + b_i^{(1)}, \quad \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} = 1, \quad \frac{\partial Z^{(1)}}{\partial B^{(1)}} = [1, 1, \dots, 1].$$

Тобто $\frac{\partial F}{\partial B^{(1)}} = \frac{\partial F}{\partial Z^{(1)}}$.

Знайдемо $\frac{\partial Z^{(1)}}{\partial W^{(1)}}$. Із вищезаписаної формули обчислення $z_i^{(1)}$ маємо:

$$\frac{\partial z_i^{(1)}}{\partial W_{ij}^{(1)}} = x_j$$

Повністю аналогічним чином $\frac{\partial Z^{(1)}}{\partial W^{(1)}}$ зводиться із тривимірного якобіана до одного вектора. Але у даному випадку це буде не $A^{(1)T}$, а X^T , тобто транспонований із колонки в рядок вектор вхідних даних. І, виконавши множення кожного елемента представленого колонкою вектора $\frac{\partial F}{\partial Z^{(1)}}$ довжини n на кожен із k елементів «рядка» X^T , отримаємо бажану $(n \times k)$ матрицю $\frac{\partial F}{\partial W^{(1)}}$, кожен елемент якої матиме значення $\frac{\partial F}{\partial W_{ij}^{(1)}}$, $i = \overline{1, n}$, $j = \overline{1, k}$.

Тепер, коли було вираховано градієнти для кожної із ваг усього персептрона, оптимізатор буде використовувати ці дані для проведення навчання, тобто певним чином змінюватиме коефіцієнти. Але це уже діло алгоритму оптимізатора, а не зворотного поширення.

1.5.4 – ЗПП для шарів згортки та пулінгу

ЗПП для згортки

Вважаю найкращим поясненням роботи ЗПП на згортці демонстрацію прикладу. Нехай ми маємо карту $X(3 \times 3)$ та фільтр $F(2 \times 2)$. Повний етап прямого поширення по шару згортки було продемонстровано в пункті 1.4.1. Зараз для простоти представимо згортку даних X із фільтром F зображеним на рис. 30 чином.

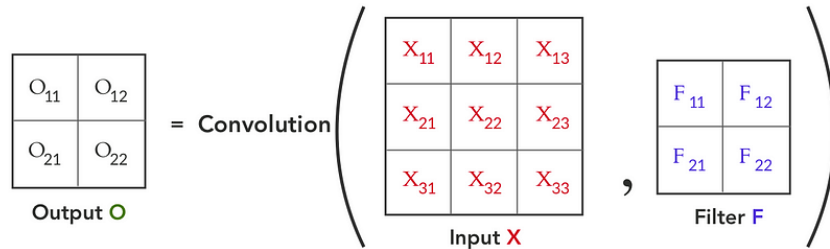


Рис. 30: візуальне позначення операції згортки [17]

Прямі обчислення матимуть такий вигляд:

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22};$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22};$$

$$O_{21} = X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22};$$

$$O_{22} = X_{22}F_{11} + X_{23}F_{12} + X_{32}F_{21} + X_{33}F_{22}.$$

Тобто, на вхід операції згортки мали дані X та ваги F (критерій у даному випадку позначимо як L). І на виході отримали інші дані, O . Суть ЗПП для згортки та сама – маючи глобальний градієнт $\frac{\partial L}{\partial o} = [\frac{\partial L}{\partial o_{ij}}, i, j \in \{1; 2\}]$, знайти локальні градієнти $\frac{\partial o}{\partial x}, \frac{\partial o}{\partial f}$ та отримати $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial f}$ із ланцюгового правила. Візуально ці обчислення представлені на рисунку 31.

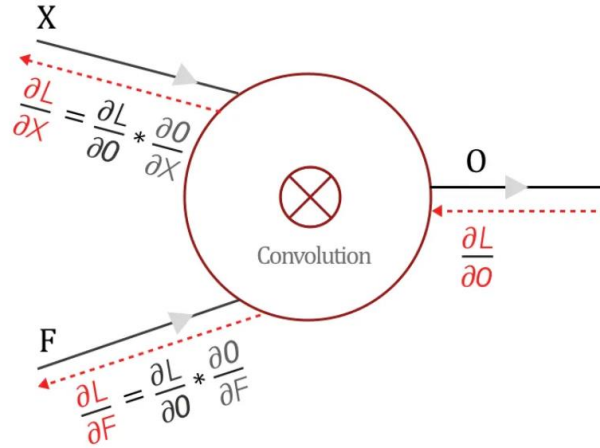


Рис. 31: Візуалізація ЗПП для згортки [17]

Тож, коли усе звелось до локальних градієнтів, виникає закономірне питання їх пошуку – згортка не є тривіальною операцією. Але спосіб обчислення градієнту згортки наявний, і його розкриття відбувається при обчисленні повних градієнтів, тобто $\frac{\partial L}{\partial X}$ та $\frac{\partial L}{\partial F}$.

Для початку - $\frac{\partial L}{\partial F}$. Із прямих обчислень O_{ij} маємо:

$$\begin{aligned} \frac{\partial O_{11}}{\partial F_{11}} &= X_{11}; \frac{\partial O_{11}}{\partial F_{12}} = X_{12}; \frac{\partial O_{11}}{\partial F_{21}} = X_{21}; \frac{\partial O_{11}}{\partial F_{22}} = X_{22}. \\ \frac{\partial O_{12}}{\partial F_{11}} &= X_{12}; \frac{\partial O_{12}}{\partial F_{12}} = X_{13}; \frac{\partial O_{12}}{\partial F_{21}} = X_{22}; \frac{\partial O_{12}}{\partial F_{22}} = X_{23}. \\ \frac{\partial O_{21}}{\partial F_{11}} &= X_{21}; \frac{\partial O_{21}}{\partial F_{12}} = X_{22}; \frac{\partial O_{21}}{\partial F_{21}} = X_{31}; \frac{\partial O_{21}}{\partial F_{22}} = X_{32}. \\ \frac{\partial O_{22}}{\partial F_{11}} &= X_{22}; \frac{\partial O_{22}}{\partial F_{12}} = X_{23}; \frac{\partial O_{22}}{\partial F_{21}} = X_{32}; \frac{\partial O_{22}}{\partial F_{22}} = X_{33}. \end{aligned}$$

Тепер обчислюємо $\frac{\partial L}{\partial F} = [\frac{\partial L}{\partial F_{ij}}, i, j \in \{1; 2\}]$ за ланцюговим правилом:

$$\frac{\partial L}{\partial F_{ij}} = \sum_{l=1, m=1}^2 \frac{\partial L}{\partial O_{lm}} * \frac{\partial O_{lm}}{\partial F_{ij}}$$

Тобто маємо:

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{11}};$$

$$\begin{aligned}\frac{\partial L}{\partial F_{12}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{12}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{12}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{12}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{12}}; \\ \frac{\partial L}{\partial F_{21}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{21}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{21}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{21}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{21}}; \\ \frac{\partial L}{\partial F_{22}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{22}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{22}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{22}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{22}}.\end{aligned}$$

Підставивши обчислені $\frac{\partial O_{ij}}{\partial F_{kl}}$, матимемо:

$$\begin{aligned}\frac{\partial L}{\partial F_{11}} &= \frac{\partial L}{\partial O_{11}} * X_{11} + \frac{\partial L}{\partial O_{12}} * X_{12} + \frac{\partial L}{\partial O_{21}} * X_{21} + \frac{\partial L}{\partial O_{22}} * X_{22}; \\ \frac{\partial L}{\partial F_{12}} &= \frac{\partial L}{\partial O_{11}} * X_{12} + \frac{\partial L}{\partial O_{12}} * X_{13} + \frac{\partial L}{\partial O_{21}} * X_{22} + \frac{\partial L}{\partial O_{22}} * X_{23}; \\ \frac{\partial L}{\partial F_{21}} &= \frac{\partial L}{\partial O_{11}} * X_{21} + \frac{\partial L}{\partial O_{12}} * X_{22} + \frac{\partial L}{\partial O_{21}} * X_{31} + \frac{\partial L}{\partial O_{22}} * X_{32}; \\ \frac{\partial L}{\partial F_{22}} &= \frac{\partial L}{\partial O_{11}} * X_{22} + \frac{\partial L}{\partial O_{12}} * X_{23} + \frac{\partial L}{\partial O_{21}} * X_{32} + \frac{\partial L}{\partial O_{22}} * X_{33}.\end{aligned}$$

А тепер, порівнявши це з обчисленнями у згортці при прямому поширенні, можемо переконатися, що ці формули – не що інше, як та ж згортка (рис. 32).

$$\begin{array}{|c|c|} \hline \frac{\partial L}{\partial F_{11}} & \frac{\partial L}{\partial F_{12}} \\ \hline \frac{\partial L}{\partial F_{21}} & \frac{\partial L}{\partial F_{22}} \\ \hline \end{array} = \text{Convolution} \left(\begin{array}{|c|c|c|} \hline X_{11} & X_{12} & X_{13} \\ \hline X_{21} & X_{22} & X_{23} \\ \hline X_{31} & X_{32} & X_{33} \\ \hline \end{array}, \begin{array}{|c|c|} \hline \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \hline \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \\ \hline \end{array} \right)$$

Рис. 32: Візуалізація проведених вище обчислень [17]

Залишилося обчислити $\frac{\partial L}{\partial X}$. Із прямих обчислень O_{ij} маємо:

$$\begin{aligned}O_{11} &= X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}; \\ O_{12} &= X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}; \\ O_{21} &= X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22}; \\ O_{22} &= X_{22}F_{11} + X_{23}F_{12} + X_{32}F_{21} + X_{33}F_{22}.\end{aligned}$$

$$\begin{aligned}\frac{\partial O_{11}}{\partial X_{11}} &= F_{11}; \frac{\partial O_{11}}{\partial X_{12}} = F_{12}; \frac{\partial O_{11}}{\partial X_{21}} = F_{21}; \frac{\partial O_{11}}{\partial X_{22}} = F_{22}; \frac{\partial O_{11}}{\partial X_{ij}} = 0 \text{ для решти комбінацій } ij; \\ \frac{\partial O_{12}}{\partial X_{12}} &= F_{11}; \frac{\partial O_{12}}{\partial X_{13}} = F_{12}; \frac{\partial O_{12}}{\partial X_{22}} = F_{21}; \frac{\partial O_{12}}{\partial X_{23}} = F_{22}; \frac{\partial O_{12}}{\partial X_{ij}} = 0 \text{ для решти комбінацій } ij;\end{aligned}$$

$$\frac{\partial O_{21}}{\partial X_{21}} = F_{11}; \frac{\partial O_{21}}{\partial X_{22}} = F_{12}; \frac{\partial O_{21}}{\partial X_{31}} = F_{21}; \frac{\partial O_{21}}{\partial X_{32}} = F_{22}; \frac{\partial O_{21}}{\partial X_{ij}} = 0 \text{ для решти комбінацій } ij;$$

$$\frac{\partial O_{22}}{\partial X_{22}} = F_{11}; \frac{\partial O_{22}}{\partial X_{23}} = F_{12}; \frac{\partial O_{22}}{\partial X_{32}} = F_{21}; \frac{\partial O_{22}}{\partial X_{33}} = F_{22}; \frac{\partial O_{22}}{\partial X_{ij}} = 0 \text{ для решти комбінацій } ij;$$

Для $\frac{\partial L}{\partial X} = [\frac{\partial L}{\partial X_{ij}}, i, j \in \{1; 2; 3\}]$ ланцюгове правило матиме такий вигляд:

$$\frac{\partial L}{\partial X_{ij}} = \sum_{l=1, m=1}^2 \frac{\partial L}{\partial O_{lm}} * \frac{\partial O_{lm}}{\partial X_{ij}}$$

Одразу підставивши $\frac{\partial O_{lm}}{\partial X_{ij}}$ та спростивши, матимемо:

$$\frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial O_{11}} * F_{11};$$

$$\frac{\partial L}{\partial X_{12}} = \frac{\partial L}{\partial O_{11}} * F_{12} + \frac{\partial L}{\partial O_{12}} * F_{11};$$

$$\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial O_{12}} * F_{12};$$

$$\frac{\partial L}{\partial X_{21}} = \frac{\partial L}{\partial O_{11}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{11};$$

$$\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial O_{11}} * F_{22} + \frac{\partial L}{\partial O_{12}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{12} + \frac{\partial L}{\partial O_{22}} * F_{11};$$

$$\frac{\partial L}{\partial X_{23}} = \frac{\partial L}{\partial O_{12}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{12};$$

$$\frac{\partial L}{\partial X_{31}} = \frac{\partial L}{\partial O_{21}} * F_{21};$$

$$\frac{\partial L}{\partial X_{32}} = \frac{\partial L}{\partial O_{21}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{21};$$

$$\frac{\partial L}{\partial X_{33}} = \frac{\partial L}{\partial O_{22}} * F_{22}.$$

Хоч це і не очевидно від слова зовсім, але такі обчислення теж можна представити операцією згортки. Дивної згортки, що має дві особливості.

По-перше, її фільтр буде перевернутою версією стандартного F (позначимо його як F^*) (рис. 33) - спочатку його потрібно перевернути на 180 градусів по горизонталі, а потім іще на 180 по вертикалі.

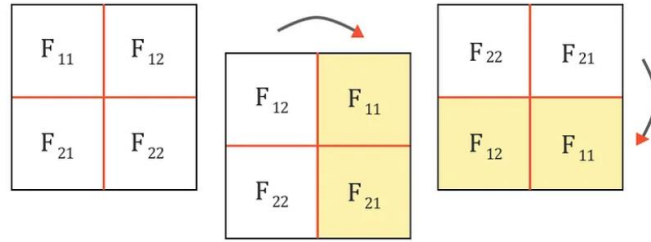


Рис. 33: Візуалізація "переворотів" фільтру F [17]

По-друге, у даній згортці «квадрату» (фільтру) дозволяється вилазити за межі «прямокутника» (картинки), тобто до тих пір, поки у квадрата є хоч одна клітинка перетину із прямокутником. Причому починає квадрат теж із крайового положення, не всередині прямокутника.

Таку згортку називають повною. А «прямокутником, по якому буде рух», у ній виступає матриця $\frac{\partial L}{\partial O}$. Прохід такої згортки зображений на рисунках 34-43.

$$\frac{\partial L}{\partial X} = \text{Full Convolution} \left(\begin{array}{|c|c|} \hline F_{22} & F_{21} \\ \hline F_{12} & F_{11} \\ \hline \end{array}, \begin{array}{|c|c|} \hline \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \hline \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \\ \hline \end{array} \right)$$

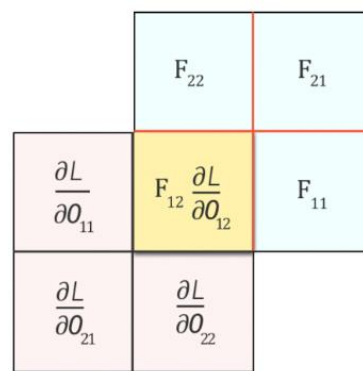
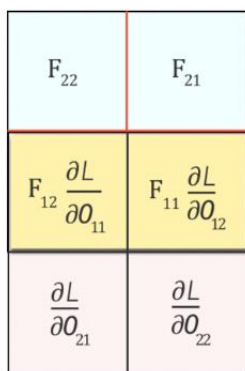
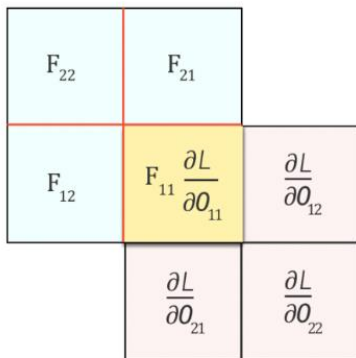
Filter F

Loss Gradient $\frac{\partial L}{\partial O}$

$$\frac{\partial L}{\partial X_{11}} = F_{11} * \frac{\partial L}{\partial O_{11}}$$

$$\frac{\partial L}{\partial X_{12}} = F_{12} * \frac{\partial L}{\partial O_{11}} + F_{11} * \frac{\partial L}{\partial O_{12}}$$

$$\frac{\partial L}{\partial X_{13}} = F_{12} * \frac{\partial L}{\partial O_{12}}$$



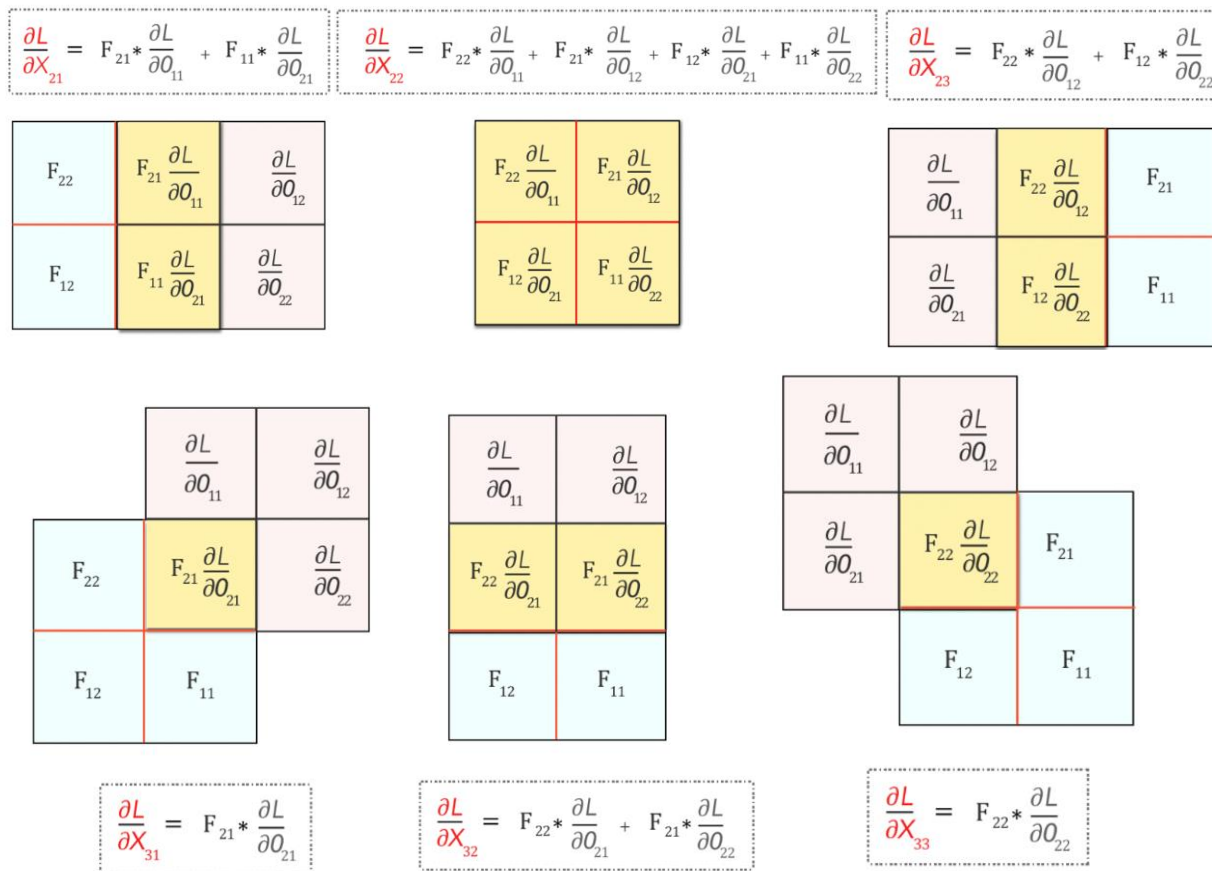


Рис. 34-43: Візуалізація повної згортки фільтру F^* із dL/dO [17]

Виникає проблема реалізації алгоритму повної згортки. Проте ненадовго, бо рішення дуже просте – оббивка (padding). У нашому випадку матрицю $\frac{\partial L}{\partial O}$ можна оббити одним рядком нулів, і згортка перетвориться на звичайну – фільтр перестане вилазити за межі «прямокутника», а результати обчислень ніяк не зміняться. У загальному випадку, коли розмір фільтра буде відрізнятися від 2×2 (нехай це буде `kernel_size`) – розмір необхідної оббивки становитиме `kernel_size-1` рядків.

Таким чином зворотне поширення похибки для шару згортки зводиться до використання двох згорток з відомими фільтрами та даними.

ЗПП для пулінгу

У прямому поширенні пулінг розміром 2×2 (stride теж поставимо стандартним), наприклад, ділить отриману карту на області 2×2 та обчислює представника для кожної.

У зворотному поширенні від наступників отримується матриця «впливів» кожного із представників на значення критерію. Тобто, усе зводиться до локальної операції над кожною з областей – маючи градієнт представника (тобто «глобальний»), обчисленого за якоюсь формулою, знайти вплив кожної клітинки області на критерій.

Для цього спочатку обчислюємо локальні градієнти, тобто вплив кожної клітинки області на отримане значення представника.

Наприклад, у max-pooling (рис. 44) представником вибирається максимальний елемент, тобто локальний градієнт відповідної клітинки =1, а решти = 0. Ну а далі просте перемноження за ланцюговим правилом.

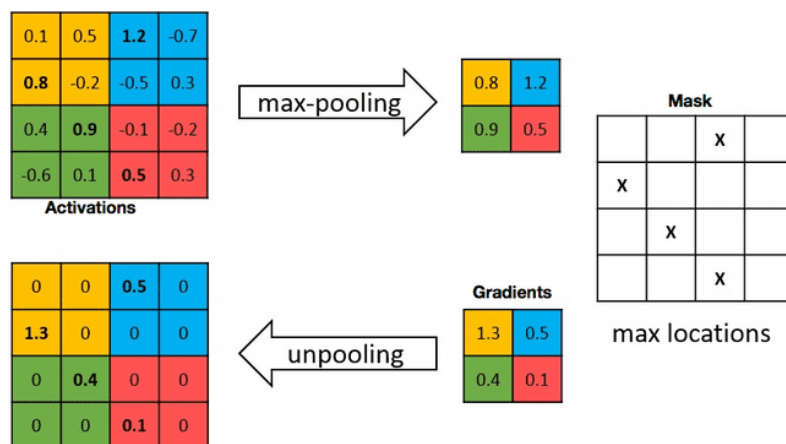


Рис. 44: Приклад ЗПП для max-pooling

У випадку average-pooling, наприклад, представник \check{a} обчислюється за формулою:

$$\check{a} = \frac{\sum_{i=1, j=1}^k a_{ij}}{k^2}$$

Відповідно, $\frac{\partial \check{a}}{\partial a_{ij}} = \frac{1}{k^2}$ для кожної клітинки a_{ij} в області із k^2 таких. Причому, так як області однакові за розмірами, локальний градієнт є однозначним для усіх клітинок карти. А далі кожен $\frac{\partial \check{a}_l}{\partial a_{ij}}$ кожної з областей множиться із відповідним їй глобальним градієнтом $\frac{\partial F}{\partial \check{a}_l}$. Для 2×2 average-pooling, наприклад, матимемо представлені на рисунку 45 обчислення.

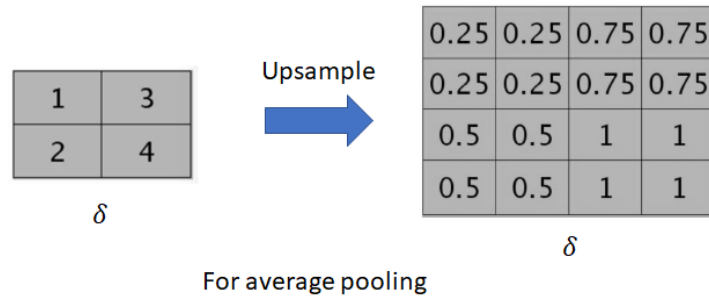


Рис. 45: Приклад ЗПП для *average-pooling*

Якщо вибрати *stride* менший за розмір сторони області, то один елемент a_{ij} може належати декільком областям. У такому випадку ланцюгове правило передбачає обчислення «впливів» a_{ij} у кожній з областей та їх сумування. А якщо обрати *stride* більший, то повні градієнти для a_{ij} , що не належать жодній області, будуть нулями. Але такі способи пулінгу на практиці якщо й використовуються, то у неймовірно рідкісних випадках, тому не вважаю за доцільне проводити повний їх розбір.

РОЗДІЛ 2. Огляд архітектури мережі MobileNetV3-Large

Тож, для розпізнавання дактилем української жестової мови було використано легкий варіант згорткової нейронної мережі, а саме MobileNetV3-Large[18]. Чому вона? Тому що MobileNetV3-Large за розмірами та спроможностями оптимізована під роботу на «легких» системах і по цій причині піддається тренуванню за адекватний проміжок часу навіть на звичайних ПК, на відміну від важких архітектур ЗНМ, для навчання яких знадобиться суперкомп'ютер.

Також авторами публікації[18] була запропонована MobileNetV3-Small, що іще більше платить точністю за вдвічі менші розміри, але цей варіант теж не є оптимальним. Так як цільовою платформою системи розпізнавання є ПК, навіть бюджетні їх варіанти відмінно справляються з версією Large.

2.1 –поверхневий огляд структури

Будова архітектури MobileNetV3-Large зображена на рисунку 46:

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Рис. 46: Специфікація архітектури MobileNetV3-Large [18]

Розберемо її по порядку. Пряме поширення відбувається зверху вниз. На вхід мережа приймає триканальні (RGB) фотографії 224×224 пікселя. Сучасні мережі досягають дуже високої точності апроксимації за рахунок великої кількості шарів. По цій причині використовується модульна структура: мережа складається із блоків, кожен із яких представляє собою якусь підпоследовність шарів.

Навіть під conv2d (назва першого блоку, колонка Operator) мається на увазі не просто шар згортки на зображеннях, а ціла послідовність:

- Власне згортка;
- Функція активації;
- Шар нормалізації даних[19,20].

2.1.1 – коротко про нормалізацію даних

Для чого потрібна нормалізація даних? У пункті 1.5 було виведено, що градієнти ваг у шарі напряду залежать від вхідних даних на шар. А вхідні дані на шар – це вихідні дані із попереднього шару, що залежать від його ваг. А ваги цього попереднього шару теж змінюються при ЗПП. Тобто, маємо таку ситуацію: було обчислено градієнти для нинішнього шару із розрахунком на те, що вхід на нинішній шар не зміниться. А потім, так як ваги у попередньому шарі змінилися, вхід на нинішній шар теж змінюється, чого ваги нинішнього шару «не очікують» - і видають менш передбачуваний (відносно попереднього шару) у плані якості результат.

Цей процес має накопичувальний ефект. Якщо у мережах із невеликою кількістю шарів ним можна знехтувати, то у глибоких мережах, де кількість шарів перевищує за сотню, маленькі зміни ваг на перших шарах поступово породжують колосальні непередбачувані входи у шари в кінці мережі - виникає проблема вибухання/зникання градієнту.

Нормалізація даних, тобто зміна середнього та дисперсії вибірки на потрібні значення (ці значення, що найцікавіше, визначаються мережею по ходу тренування, тобто теж є параметрами до навчання), сприяє полегшенню ситуації, дозволяючи мережі «підлаштовувати потік даних під себе» та «спрощуючи життя вагам пізніх шарів», тобто не дозволяючи градієнту вийти з-під контролю. Таким чином досягається «безболісне» збільшення кількості шарів у мережі - що позитивно впливає на точність - та водночас значне пришвидшення процесу градієнтного спуску, тобто зменшення часу тренування.

2.1.2 – поколонковий огляд вмісту таблиці-специфікації

Перша колонка (Input) показує параметри тензорів, тобто кількість та розміри карт, що входять на шар/блок. Усі карти архітектури квадратні - по цій

причині для опису розмірів використовується нотація $h^2 \times c$, де h – ширина та висота кожної із квадратних карт, а c – кількість каналів.

Друга колонка (Operator) показує вид блоку, що використовується. Як бачимо, основу архітектури становлять так звані bneck-и (bottleneck-и, «шийки пляшки»). Повний їх опис проведено у пункті 2.2. Приставка NBN означає відсутність шару нормалізації даних у блоці згортки. Відповідь на питання «навіщо прибирати нормалізацію?» описана у пункті 2.3.

На третій колонці (exp size, expansion size) відображено «обсяг розширення» - один із параметрів bottleneck-ів, описаний у 2.2.

Четверта колонка (#out) описує кількість карт, що повертає шар. Для пулінгу вона не вказується, так як він ніяк не впливає на цю кількість.

П'ята колонка (SE) зображує необхідність використання процедури squeeze-and-excitation («зчавлення й насичення») у bottleneck. Процедура SE описана у пункті 2.2.2.

Шоста колонка (NL, nonlinearity) показує використану у блоці функцію активації.

Сьома колонка (s, stride) вказує значення параметру stride(крок) для згортки/згорток блоку.

2.2 – будова блоку bottleneck

Як уже було зазначено вище, ці блоки (рис. 47) являють собою основу архітектур MobileNetV3. Назва «шийка пляшки» походить від їх архітектурної будови.

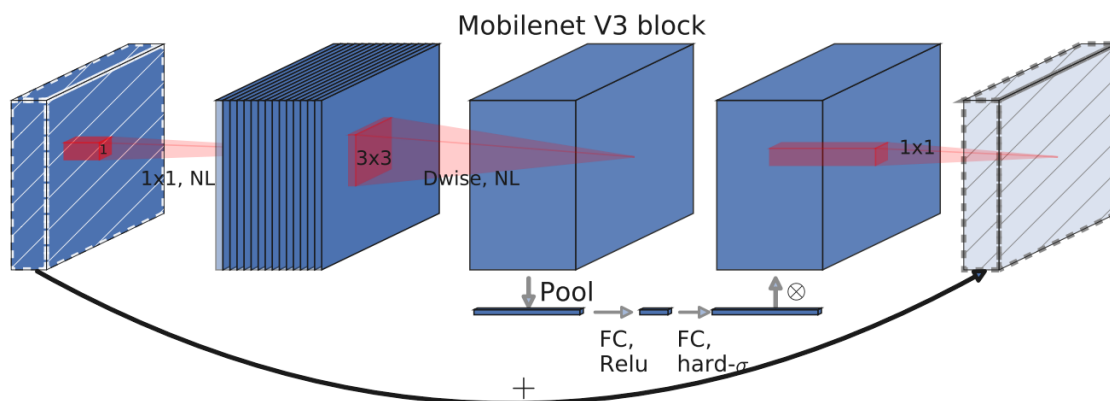


Рис. 47: Візуалізація архітектури блоку bottleneck[18]

Розпишемо послідовно його роботу. На початку маємо «вузенький прямокутний паралелепіпед», тобто невелику кількість карт. До неї застосовується звичайна згортка із фільтрами 1×1 . Зазначу, що фільтр має сенс – він насправді тривимірний, і розмір по третьому виміру відповідає кількості каналів, тобто карт. Причому кількість цих фільтрів значно перевищує початкову кількість каналів – дістається велика кількість ознак «з різного ракурсу» (кожна ознака=карта є наслідком роботи окремого фільтра), тобто розмір паралелепіпеда зростає.

Далі до отриманого тензору застосовується роздільна по глибині згортка (depthwise separable convolution).

2.2.1 - коротко про роздільну по глибині згортку

У звичайній згортці для отримання n карт використовується n різних фільтрів, кожен із яких має розмір $k \times k \times c$, де k – розмір ядра, та c – кількість карт на вході. Тобто, кількість параметрів до навчання становить $k * k * c * n$. Роздільна по глибині згортка (рис. 48) використовує трохи іншу стратегію, розбиваючи обчислення на два етапи.

У першому етапі, замість створення n фільтрів, вона проходить усі карти одним фільтром тих самих параметрів $k \times k \times c$, де k - розмір ядра згортки – є тим же параметром, що і у звичайної згортки. Причому відбуваються тільки поканальні зважені суми – без кінцевого сумування в один піксель, тобто із проходу «кубика» по початковому паралелепіпеду $H \times W \times c$ отримується паралелепіпед $H_1 \times W_1 \times c$, де H_1 і W_1 будуть трохи меншими від оригінальних параметрів карт H і W через особливості згортки. Ось чому така згортка названа роздільною по глибині.

А на другому етапі створюється n «точкових» $1 \times 1 \times c$ фільтрів, і кожен із цих n фільтрів проводить звичайну згортку на отриманому $H_1 \times W_1 \times c$ тензорі, тобто «проходить кубиком по паралелепіпеду». Таким чином отримується необхідна кількість карт. Присутні втрати точності, зате кількість множень та, відповідно, ваг до навчання падає в рази, значно «полегшуючи» мережу.

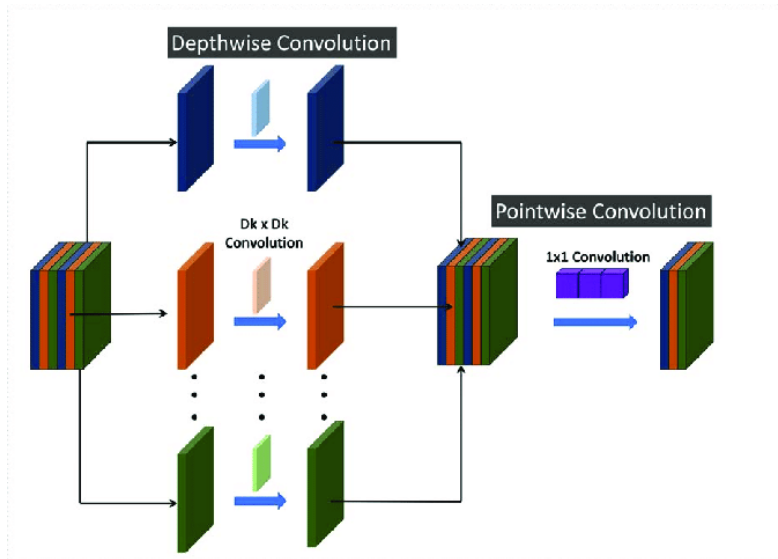


Рис. 48: Візуалізація роздільної по глибині згортки[21]

2.2.2 – принцип роботи SE

Після проведення роздільної згортки отримали певний тензор, тобто певну кількість карт. Причому зазвичай усі карти розглядаються як рівноправні. Суть SE («Squeeze-and-Excitation»[22], «зчавлення й насичення») (рис. 49) полягає у наданні картам «важливостей».

Працює цей підблок дуже просто: кожену із отриманих n карт «зчавлюють» до одного пікселя (обчислюють середнє значення), таким чином отримуючи вектор $1 \times 1 \times n$ (цю операцію на візуалізації названо пулінгом, що дійсно правдиво). Далі отриманий вектор пропускається через двошаровий перцептрон із певними функціями активації, у випадку MobileNetV3 це ReLU та H-sigmoid для 1 та 2 повнозв'язного шару відповідно.

На виході такої міні-мережі (що теж навчається при тренуванні) отримуємо вектор довжини n , кожне значення якого відповідатиме вазі відповідної карти. Для отримання результуючого «зваженого тензора» в кінці процедури відбувається попіксельне множення, тобто «насичення» кожної карти зі значенням відповідної їй ваги.

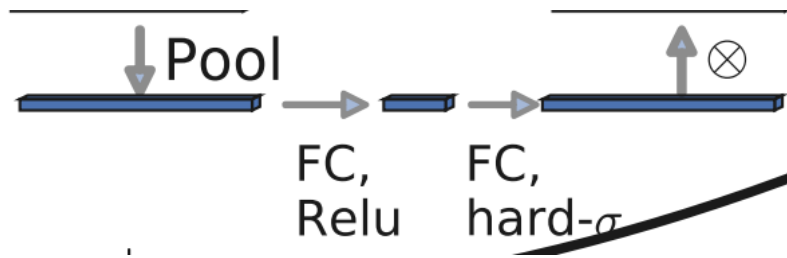


Рис. 49: Збільшення рисунку 47, візуалізація міні-мережі Squeeze-and-Excitation

Така проста за суттю операція із маленькою вартістю обчислень (буквально 1% часу виконання мережі) дозволяє значним чином підвищити кінцеву точність апроксимації – у році її створення, 2018, мережі на базі SE показали на 25% кращі результати!

Повернемося до блоку bottleneck. Після проведення зчавлення-насичення над отриманим із глибинно роздільної згортки тензором застосовується іще одна, кінцева згортка – стандартна згортка із фільтрами 1×1 . І цим архітектура блоку завершується.

Конкретизація архітектури bottleneck-у включає встановлення таких параметрів:

- 1) Розміри вхідного тензора: $H_{in} \times W_{in} \times C_{in}$ (як було зазначено раніше, розбираємо роботу мережі на обробці однієї картинки);
- 2) Розміри вихідного тензора: $H_{out} \times W_{out} \times C_{out}$;
- 3) Міра внутрішнього розширення *exp size* (кількість каналів, до якої збільшується тензор всередині мережі);
- 4) SE – наявність або відсутність;
- 5) Тип функції активації, що застосовується у шарах блоку; на візуалізації (рис.47) місця використання нелінійності позначено як NL;
- 6) s – параметр stride для згорток всередині блоку – однаковий для всіх.

2.3 – класифікатор

У таблиці-специфікації (рис. 46) класифікатор (рис. 50) представляється послідовністю блоків згорток (і пулінгом), що слідує після останнього bottleneck-а. Виникають закономірні питання – чому відсутні повнозв'язні шари? Хіба вони не є обов'язковими для класифікації?

$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Рис. 50: Специфікація MobileNetV3-Large для етапу класифікації

Насправді ж, вони нікуди не ділися. Повнозв'язні шари присутні, але обчислення, що у них проводяться, тобто отримання $y = activation(Wx + B)$, представлені згортками.

Розглянемо специфікацію для двох останніх блоків згорток. Перший блок має назву conv2d 1x1, NBN, та hard-sigmoid у якості активації. На вхід отримує 960 карт 1×1 , і застосовує фільтр $1 \times 1 (\times 960)$ для згортки. Пригадавши обчислення всередині фільтру, отримаємо аналог пошуку зваженої суми по вектору із 960 значень. Обчислення під одним фільтром дуже сильно нагадують обчислення всередині одного нейрона, чи не так?

Так як на виході отримуємо 1280 карт, тобто маємо 1280 таких «нейронів», і для кожного обчислюємо зважену суму із отриманого на вхід «вектора довжиною 960». Потім отримані суми у цих нейронах пропускаються через функцію активації і передаються на наступний блок. Без нормалізації (вказано NBN – no batch normalization).

Для другого блоку усе аналогічно – отримує на вхід 1280 карт 1×1 , тобто 1280 значень, застосовує k, де k – кількість класів до класифікації, фільтрів $1 \times 1 (\times 1280)$, тобто обчислює k зважених сум і видає результат (функція активації відсутня).

Висновки очевидні: два останні блока це не що інше, як аналоги повнозв'язних шарів. А перед цими двома «повнозв'язними шарами» проводиться «препроцесінг», тобто конвертація даних із 2D формату в «1D вектор» – за неї відповідає блок conv2d, 1x1 із hard-sigmoid у якості активації та шар average pooling, що «зчавлює» карти 7×7 до необхідного 1×1 вигляду.

РОЗДІЛ 3. Навчання MobileNetV3-Large та результати

Тепер перейдемо до власне вирішення задачі розпізнавання дактилем української жестової мови. Для цього необхідно провести навчання MobileNetV3-Large, аби та розпізнавала жести.

Програму для навчання було написано із використанням технології pytorch (а саме версії із підтримкою CUDA – використання цієї технології дозволило перенести множення матриць на відеокарту та зменшити цикл тренування до двох тижнів). За основу було використано реалізацію [23]. Код програми міститься у додатку А.

3.1 – датасет

Першим і ключовим завданням для тренування виступає підбір правильної множини даних, на яких воно відбуватиметься - датасету. У нашому випадку, тобто навчанню з учителем, така множина містить пари (X, Y_{true}) , де X – зображення, Y_{true} – мітка класу (подана як «маска», тобто вектор значень, який очікується на виході мережі), до якого належить фотографія X . Правильність датасету є критичною складовою у буквальному сенсі – недостатньо репрезентативна вибірка «спровокує» мережу сформувати неправильне представлення того, що від неї вимагатиметься. Це значить те, що вибірка має бути такою, що приведе мережу до «суті завдання» - аби та знайшла правильні закономірності.

Наприклад, якщо «показувати» мережі жести тільки на одній конкретній відстані від камери (рука матиме конкретний масштаб), то в кінці кінців вона навчиться розпізнавати руку лише у тому масштабі. Якщо постійно згодовувати мережі фото жесту в одному ракурсі, то потім при маленькій його зміні мережа теж «посипеться».

Також, крім репрезентативності, важливими є розміри датасету. Практикою підтверджено, що результати тренування мережі ростуть пропорційно розмірам навчальної вибірки (поки мережа не «впреться у свій ліміт», так як розмір архітектури теж впливає на якість апроксимації). Замала кількість фотографій призведе до того, що мережа просто їх «запам'ятає». Причому йдеться не про десятки фотографій, і не про сотні. Навіть відносно маленькій мережі MobileNetV3-Large для «вияснення суті» потрібні «перегляди» тисяч і тисяч екземплярів.

У задачах класифікації зображень є важливою іще одна особливість – датасет повинен бути збалансованим. Це означає те, що кількість фотографій кожного із класів має бути приблизно однаковою і необхідно для забезпечення адекватного розпізнавання кожного із класів.

Для розуміння суті третьої ознаки розглянемо приклад: нехай маємо датасет із двома класами, де показані фотографії банку - із грабіжниками всередині та без них. Звісно, записаний із реальних даних сет міститиме 99% фото без грабіжників та 1% фото із ними. Тоді, при навчанні, мережа може (і буде) просто видавати лінійний результат «це фото без грабіжників» - і бути правою у 99% випадків! Але ж увесь сенс розпізнавання тоді пропадає – навіщо здалася мережа, яка не розуміє, чи з'явилися грабіжники на фото.

Тож, для навчання було використано збалансований датасет[24] із 46 тисяч фотографій. Попри те, що він використовувався для 3D -згортки (це згортки, що працюють із часовою послідовністю картинок), його зображення без проблем (вибірка дуже велика та репрезентативна) можна використовувати для подання на звичайні ЗНМ, включаючи MobileNetV3.

Тренування (візуалізовано на рис. 51) проводиться таким чином:

- 1) Датасет розбивається на три підвибірки – train, validation, test. Без перетинів;
- 2) На підвибірці train відбувається тренування мережі на певних гіперпараметрах;
- 3) На підвибірці validation тестується якість розпізнавання мережею нових раніше не зустрітих даних;
- 4) Якщо результати незадовільні, пункти 2)-3) проводяться заново зі зміненими гіперпараметрами. Зміна параметрів відбувається залежно від поведінки мережі. Що і наскільки змінювати – та іще задача.
- 5) Коли гіперпараметри відрегульовані та результати на validation максимізовані, мережа перевіряється на фінальному наборі даних, який іще ні разу не зустрічала – test. І по цій підвибірці даних визначається фінальна продуктивність мережі.

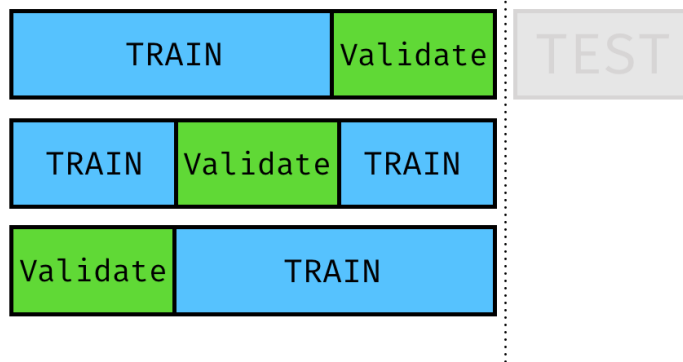


Рис.51: Візуалізація циклу train/validation/test

Для навчання MobileNetV3-Large вищезазначений датасет було розбито на вибірки train/validation/test у співвідношенні 85/10/5 (%). Здавалося б, 5% - хіба настільки малою підвибіркою можна перевірити продуктивність? А 5% від 46 тисяч це приблизно 2 тисячі фото – що для невеликої мережі (всього 5.4 млн параметрів відносно 50 млн у «важчих» моделей) уже суттєво.

3.2 – критерій оптимізації

Другим із ключових завдань тренування є вибір критерію оптимізації (його ще називають функцією втрат), тобто функції $F(Y_{pred}, Y_{true})$, де Y_{pred} – вектор, отриманий із мережі, та Y_{true} – мітка, надана учителем (подана як «маска»). Ця функція повертає значення помилки, тобто міру того, наскільки наявний Y_{pred} відрізняється від бажаного Y_{true} . Ну і це та сама F , на яку зворотним поширенням похибки шукається вплив кожної із ваг мережі.

Для тренування MobileNetV3-Large використовувалася перехресна ентропія[25]. Термін «ентропія» було запропоновано в дослідженнях у сфері термодинаміки, але корисність отриманих результатів поширила використання формули ентропії у математичних задачах. Замість розписування визначення ентропії, продемонструю усе «в лоб», тобто з точки зору обчислень у мережі.

Тож, перехресна ентропія задається ось такою формулою (вважаємо, що нумерація починається із числа 1):

$$F(Y_{pred}, Y_{true}) = - \sum_{i=1}^n y_{true_i} * \log(y_{pred_i}) = \sum_{i=1}^n y_{true_i} * -\log(y_{pred_i})$$

Ця формула має обмеження – усі y повинні лежати у межах $[0; 1]$, та $\sum y_{pred_i} = \sum y_{true_j} = 1$, тобто усі $y \in$ ймовірностями. Також варто відмітити, що база логарифму не має конкретного припустимого значення, тобто різниться в залежності від цілі використання ентропії. У реалізаціях НМ використовується натуральний логарифм.

Для початку потрібно розібратися з обмеженнями. У випадку міток усе просто – для класу, якому належить зображення, встановлюється одиниця, а для решти значень у масці – нулі. Для передбачення ж, тобто даних із мережі, використовується нормалізація вихідного вектору. Спеціально для випадку ентропії створено шар softmax, що нормалізує вектор даних X за такою формулою:

$$y(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Виникає закономірне питання – який сенс ускладнювати рівняння експонентами, якщо можна просто поділити значення x_i на суму x_j ? Діло в тому, що «експонентна варіація нормалізації» дуже добре «сходиться» із раніше зазначеною формулою ентропії (у НМ як раз використовується натуральний логарифм), значно спрощуючи пошук часткових похідних. Крім того, як наслідок спрощення, градієнти також певним чином «нормалізуються» - що дозволяє уникнути проблем зникання і вибухання.

Тож, принцип роботи. Підставивши «надану учителем маску» Y_{true} у формулу перехресної ентропії, матимемо те, що для картинки, класифікованої учителем у клас i , $y_{true_j} = 0$, для $j \neq i$. Тобто, формула ентропії спрощується із суми до:

$$F(Y_{pred}, Y_{true}) = y_{true_i} * -\log(y_{pred_i}) = -\log(y_{pred_i})$$

Розглянемо графік функції $L(y) = -\log_e(y) = \ln(y)$ у межах від 0 до 1 (рис. 52-53)

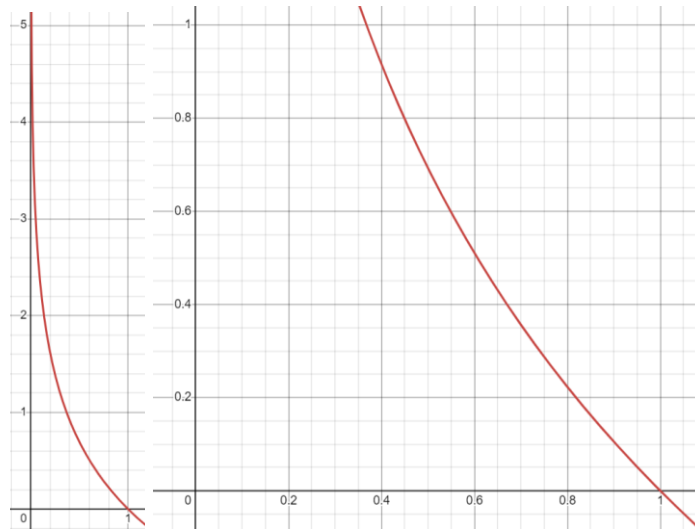


Рис. 52-53: Графік $F(y) = -\ln(y)$ у двох масштабах

Із нього чітко видно, що $L \rightarrow \infty$ при $y \rightarrow 0$ та $L \rightarrow 0$ при $y \rightarrow 1$. Це значить, що при прямує до одиниці впевненості мережі y_{pred_i} матимемо мінімальну похибку, тобто різницю між очікуваним $y_{true_i} = 1$ та отриманим y_{pred_i} . А при $y_{pred_i} \rightarrow 0$, що вважається дуже поганим результатом, матимемо дуже велике значення результату.

Отже, при $y_{pred_i} \rightarrow 1$ матимемо $F \rightarrow 0$. Також, при $y_{pred_i} \rightarrow 1$ матимемо $y_{pred_j} \rightarrow 0, j \neq i$, так як $\sum y_{pred_j} = 1$. Із цього випливає те, що перехресна ентропія визначає «різницю» отриманого вектора Y_{pred} із бажаним Y_{true} та, відповідно, повністю відповідає означенню функції втрат.

3.3 – градієнтний оптимізатор

Після отримання градієнтів через ЗПП відносно вищеприписаного критерію залишається питання «а що із ними робити?». Цю задачу покликано вирішувати градієнтні оптимізатори.

По суті, усе що вони роблять – змінюють кожну вагу на пропорційну її впливу на критерій величину. Аби пізнати як саме, повернемося до фундаменту.

Як було зазначено раніше, похідна відповідає на питання «на скільки h -ок зміниться функція F , якщо, вважаючи решту змінних константами, збільшити обрану змінну на малесеньке h ?». Нехай при збільшенні змінної x на h значення $F(x)$ збільшиться на $12h$, тобто $\frac{\partial F}{\partial x} = 12$. Наша задача полягає у

мінімізації $F(x)$, а це, у нашому випадку, має на увазі зменшення аргументу x на якусь величину, прямо пропорційну значенню похідної (відповідь на питання «чому» продемонстрована декількома абзацами нижче).

Розглянемо тепер випадок спадної функції від однієї змінної, тобто при збільшенні x на h функція зменшується (нехай маємо $\frac{\partial F}{\partial x} = -12$). Тоді для мінімізації $F(x)$ аргумент x потрібно збільшувати. Знову віднявши від аргументу пропорційну значенню часткової похідної величину (яка від'ємна), отримаємо збільшення аргументу x .

Із двох вищеописаних прикладів слідує те, що незалежно від того, як змінюється $F(x)$ при збільшенні x на h , віднімаючи від аргументу пропорційну $\frac{\partial F}{\partial x}$ величину ми завжди наближатимемо $F(x)$ до мінімуму. У багатовимірних випадках вектор часткових похідних показує конкретний (значно складніший) напрямок збільшення $F(x)$, а тому зменшувати аргументи доцільно тільки на пропорційні значенням вектора величини. Тобто, значення ваг (інакше кажучи, аргументів) для виконання градієнтного спуску в найпростішому його варіанті потрібно змінювати ось таким чином:

$$x = x - \alpha * \frac{\partial F}{\partial x}$$

Де x – вага, що підлягає зміні, та α – швидкість навчання мережі, один із гіперпараметрів.

Для навчання MobileNetV3-Large було використано оптимізатор Adaptive Moment Estimation[26], або ж «Адам». Його спосіб зміни ваг виглядає дещо складніше та відбувається у декілька стадій.

Спочатку відбувається розрахунок першого та другого «затухаючих моментів» градієнтів. Ці величини «зберігають» значення раніше отриманих градієнтів, таким чином апроксимуючи середнє значення градієнту по всьому датасету. Так як дані подаються на вхід партіями, такі апроксимації «оптимального випадку» дозволяють значно зменшити кількість кроків до збігання. Обчислення моментів виглядають таким чином:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

де m_t та v_t – перший та другий затухаючі моменти (на кроці навчання t) відповідно, g_t – обчислене значення часткової похідної для ваги θ на кроці t , β_1 та β_2 – «темпи розпаду», одні із гіперпараметрів оптимізатора.

Так як на першому кроці навчання значення затухаючих моментів ініціалізуються нулями, авторами було запропоновано додаткові операції для уникнення «упередженості» моментів (зміщеність надто сильно впливає на результат навчання у початкових ітераціях, а особливо при $\beta_1, \beta_2 \rightarrow 1$) до нуля:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1}; \hat{v}_t = \frac{v_t}{1 - \beta_2}$$

І тепер отримані виправлені значення моментів можна використовувати для редагування ваги θ :

$$\theta_{t+1} = \theta_t - \frac{\alpha * \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

де θ_{t+1} – нове значення ваги, θ_t – нинішнє значення ваги, α – швидкість навчання мережі, ϵ – «чутливість оптимізатора», тобто маленьке значення для запобігання діленню на нуль.

Зазначу, що усі вищезазначені формули були продемонстровані для однієї ваги. Але суть ніяк не зміниться, якщо замість однієї часткової похідної подати одразу вектор таких, тобто градієнт, і обчислювати нові значення одразу для вектора (чи матриці) ваг.

Тож, Адам має 4 гіперпараметра:

- 1) α – швидкість навчання мережі; було встановлено $\alpha = 0.00005$.
- 2) β_1 та β_2 – «темпи розпаду»; було залишено запропоновані авторами оптимізатора значення - $\beta_1 = 0.9$; $\beta_2 = 0.999$.
- 3) ϵ , «чутливість оптимізатора», було теж залишено із запропонованим авторами значенням $\epsilon = 10^{-8}$.

3.4 – інші гіперпараметри мережі

Серед інших гіперпараметрів НМ слід виділити такі:

- 1) batch size – розмір партії, яка «згодовується» мережі за раз;
- 2) epochs – кількість епох, тобто послідовних проходжень мережі по усьому датасету;

- 3) width multiplier – множник ширини мережі. Дозволяє пропорційно збільшити/зменшити «розміри» кожного із шарів;
- 4) Dropout – ймовірність «вибування» нейрону із повнозв'язного шару. Не дозволяє нейронам сильно адаптуватися одне під одного, поліпшуючи стабільність мережі і результату.

Розберемо кожен із них по порядку. Batch size, розмір партії, вплив якого був уже неодноразово згаданий, відповідає за кількість фотографій, що мережа обробляє за раз при прямому поширенні. Чим більша партія проходить пряме поширення, тим більш статистично точним буде обчислене середнє значення похибки та, відповідно, якіснішим буде результат зміни ваг. А якісна зміна ваг впливає на швидкість збіжності градієнтів при спуску, себто швидкість тренування мережі. З іншого боку, наявне обмеження пам'яті системи – при занадто великому розміру batch-у пам'яті найшвидших кешів не вистачить і доведеться використовувати інші, більші за розміром та на порядки повільніші джерела (RAM, наприклад), що призведе до недопустимо сильного сповільнення обчислень.

Для тренування MobileNetV3-Large було встановлено batch size = 50. Експериментальним чином було визначено, що при більших розмірах партії спостерігаються кратні сповільнення обчислень, що свідчить про нестачу пам'яті.

Кількість епох, epochs – гіперпараметр, що регулює скільки разів мережа проходить увесь датасет. Тобто, мережа взяла одну партію фото, обробила, провела зворотне поширення, відредагувала дані, далі взяла іншу партію, далі третю, п'яту, десятую, соту, і т.д. Коли усі наявні у датасеті фотографії пройдено та «висновки» зроблено, одна епоха вважається закінченою. Чому потрібно проводити більш ніж один обхід? Бо мережа не встигає визначити абсолютно усі закономірності «на льоту» - люди теж так не вміють, нам для засвоєння чогось потрібно багато практики. З іншого боку, якщо встановити занадто велику кількість параметрів, мережа буде запам'ятовувати не суть, а усілякі непотрібні деталі. Відповідно, при поданні на неї нових даних вона працюватиме погано. Приблизно оптимальною кількістю епох для навчання MobileNetV3 є сотня, тобто epochs = 100.

Множник ширини мережі (width multiplier) дозволяє «звужити» або «розширити» архітектуру мережі. Це можна візуалізувати як «розтягнення

графу обробки даних». Тобто, кількість нейронів/фільтрів/тощо у кожному (чи певних, залежить від реалізації) шарі мережі множиться на коефіцієнт `width multiplier`. Цей параметр потрібен для ефективного підлаштування мережі під розміри даних для навчання – якщо для виконання задачі необхідно запам'ятовування надто великої (відносно базових розмірів мережі) кількості контекстуальних дрібниць, то пропорційне збільшення розмірів сильно спрощує життя. Для MobileNetV3-Large було залишено стандартне значення `width multiplier = 1.0`.

Коефіцієнт виключення (`dropout`) позначає ймовірність нейрону повнозв'язного шару бути виключеним із обчислення, «зануленим». Така практика «ставити нейрони в умови виживання» - ніколи не відомо, хто «зникне безвісти». Відповідно, нейрони стають самостійнішими - перестають підлаштовуватися під помилки одне одного і, як наслідок, видають кращі результати - та стабільнішими, що позитивно відображається на якості розпізнавання.

Встановлення `dropout`-у у межах 0.2-0.4 трохи сповільнить тренування, зате результати будуть кращими. Якщо ж встановити надто велике значення коефіцієнту, то градієнти «вестимуть казна куди» та значення помилки перестане збігатися до мінімуму. Тож, для MobileNetV3-Large було вибрано `dropout = 0.2`.

3.5 – результати навчання

Для визначення продуктивності мережі на тестовій вибірці було обчислено нормалізовану матрицю плутаниць (рис. 54).

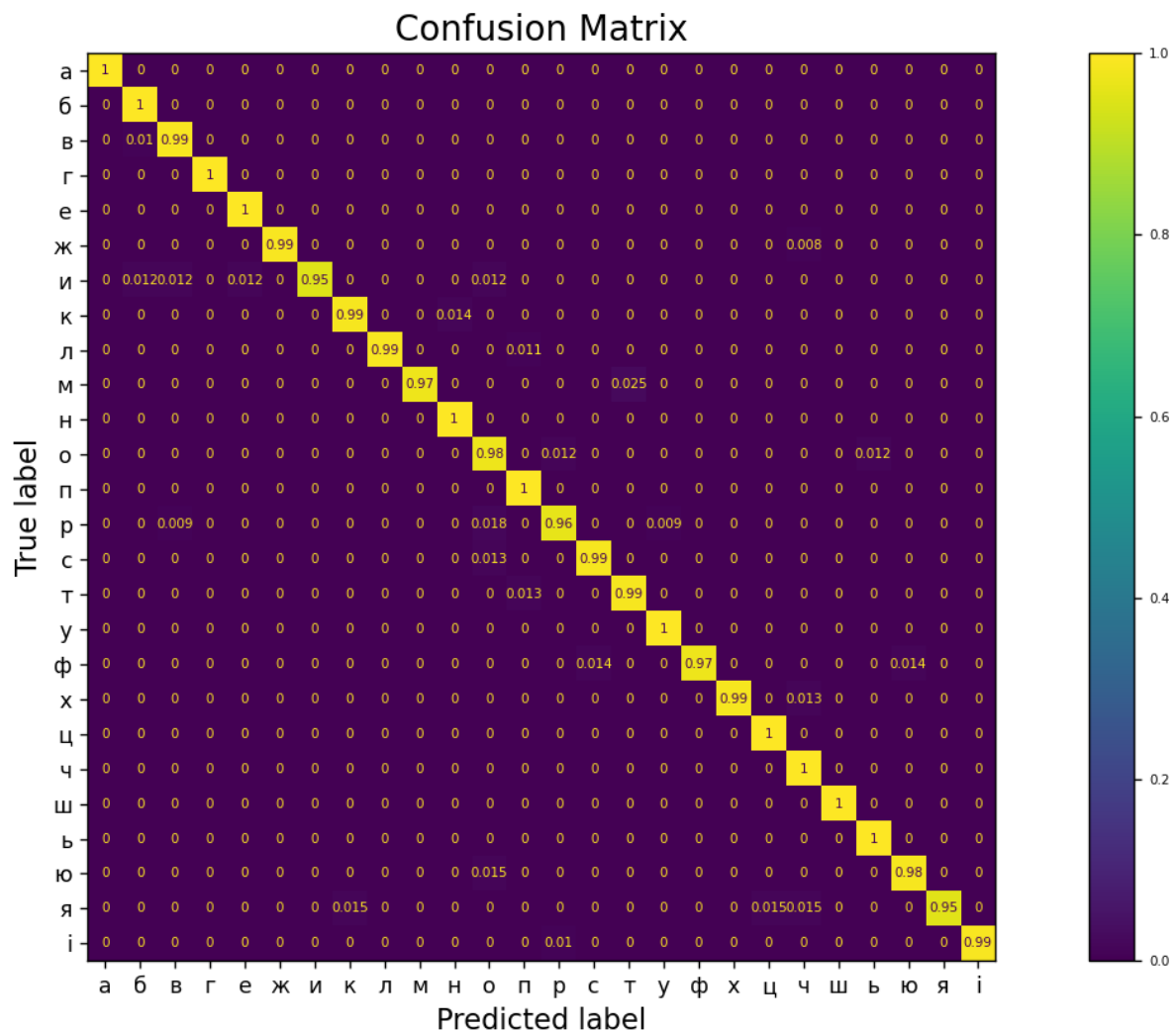


Рис. 54: матриця плутаниць MobileNetV3-Large на вибірці test

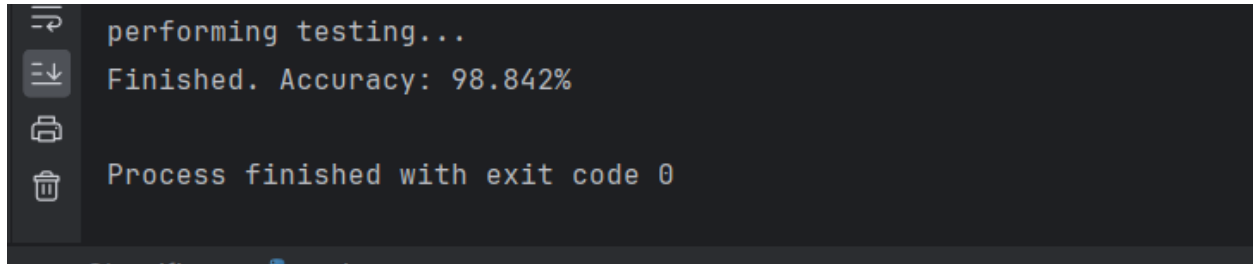
Клітинка рядка i колонки j даної матриці показує скільки відсотків картинок класу i (відносно усієї кількості картинок класу i) мережа розпізнала як клас j . Тобто, значення на діагоналях показують відсоток правильних розпізнавань картинок відповідного класу мережею.

Можна переглянути, що із чим мережа плутає: наприклад, жест літери «я» НМ подекуди вважала жестами літер «ц», «ч», чи «к». Також із матриці бачимо, що значення на діагоналях коливаються у межах від 0.95 до 1, що свідчить про приблизно збалансоване розпізнавання кожного із класів.

Так як для тренування було використано збалансований датасет, для визначення загальної продуктивності натренованої мережі можна використати метрику точності (accuracy), показник якої визначається дуже простим чином:

$$accuracy = \frac{\text{к — ть правильно розпізнаних фото}}{\text{загальна к — ть фото вибірки } test} * 100\%$$

Після проведення відповідних обчислень було отримано вражаючий результат (рис. 55).

A screenshot of a terminal window with a dark background. On the left side, there is a vertical toolbar with icons for undo, redo, copy, and delete. The terminal text shows the execution of a testing process. The first line is 'performing testing...'. The second line is 'Finished. Accuracy: 98.842%'. The third line is 'Process finished with exit code 0'.

```
-> performing testing...  
Finished. Accuracy: 98.842%  
Process finished with exit code 0
```

Рис. 55: результуюча точність

ВИСНОВКИ

Дана робота детально описує аспекти використання згорткових нейронних мереж для задачі розпізнавання дактилем української жестової мови.

Опис починається із історії виникнення нейронних мереж як таких та пояснення їх базової структури у пункті 1.1. Пункт 1.2 детально відповідає на питання працездатності раніше зазначеної структури обчислень та пояснює загальну суть і ціль роботи нейромереж з точки зору математики, а саме апроксимацію однієї чи декількох функцій від багатьох змінних, кожна із яких має певний практичний сенс. У задачах класифікації жестів цим сенсом виступає впевненість у тому, чи належить певна фотографія такому-то класу.

Пункт 1.3 докладно розписує призначення функції активації, необхідні її критерії та у якості прикладів розписує кожен з функцій активації, що міститься в архітектурі ЗНМ MobileNetV3. У пункті 1.4 відбувається глибоке занурення у конкретику архітектури згорткових нейронних мереж, зокрема проводиться детальне пояснення принципів роботи й використання базових шарів ЗНМ.

Пункт 1.5 покриває нині найдієвіший на практиці метод навчання мереж прямого поширення, тобто спосіб обчислення часткових похідних критерію по кожній із ваг для пошуку оптимальної для апроксимації комбінації цих ваг. Проводяться повні пояснення суті ЗПП і того, що усе зводиться до зручного в реалізації вирішення локальної задачі на кожному із шарів:

- 1) отримання глобального градієнта (градієнта вихідних даних) від наступників;
- 2) обчислення локальних градієнтів;
- 3) обчислення повних градієнтів завдяки ланцюговому правилу;
- 4) поширення градієнтів вхідних даних попередникам.

Крім теоретичних пояснень, у підпункті 1.5.3 проводиться повне зворотне поширення похибки на прикладі тришарового персептрона, яке включає вирішення «задач спрощення обчислень», що виникають у процесі. Підпункт 1.5.4 на прикладах розписує специфіку зворотного поширення для шарів згорткових нейронних мереж, тобто пулінгу (max-pooling та average-pooling) та згортки.

У розділі 2 розглядається архітектура, що використана для вирішення поставленої задачі розпізнавання дактилем – MobileNetV3-Large. Вибір пояснено підходящою «легкістю» архітектури – яка дозволяє проводити тренування на звичайному ПК.

Пункт 2.1 проводить огляд наданої авторами [18] таблиці-специфікації MobileNetV3-Large, «розбираючи вказані в ній дані по гвинтикам» та описуючи деталі перших блоків мережі, у тому числі зачіплюючи важливість нормалізації даних.

Пункт 2.2 проводить повний розбір основного будівельного блоку MobileNetV3, що має назву «bottleneck». Зокрема, в підпункті 2.2.1 було пояснено принцип роботи роздільної по глибині згортки, що лежить в його основі. Відносно MobileNetV2, даний блок зазнав покращень – було додано «підмережу» Squeeze-and-Excite (роботу якої повністю розписано у пункті 2.2.2), яка значно поліпшила точність третьої версії відносно другої та дозволила використати додаткові оптимізації швидкості роботи.

Пункт 2.3 розвіює плутанину із відсутніми в кінці раніше розглянутої специфікації повнозв'язними шарами, що насправді були представлені аналогічною з точки зору внутрішніх обчислень комбінацією згорток (та пулінгів для препроцесингу).

Третій розділ описує навчання моделі MobileNetV3 та усе, що для нього використовувалося – від вибірки даних для тренування до кожного із параметрів, що встановлювався.

У першому пункті було розписано що собою представляє датасет, тобто вибірка розмічених (у нашому випадку) даних, на якій проводиться навчання, та якою ця вибірка повинна бути. Другий пункт розписує усе про критерій оптимізації - функцію, що обчислює різницю між отриманим із мережі та бажаним результатами. Зокрема було розібрано принцип роботи використаного для навчання критерію – перехресної ентропії.

Третій пункт розписує оптимізатор – алгоритм, що проводить градієнтний спуск. Оптимізатор змінює кожен із ваг шляхом віднімання від неї пропорційної значенню її «впливу на критерій», тобто часткової похідної, величини. У якості такого було вибрано Adam[26] і пояснено усі обчислення, що він виконує.

У четвертому пункті детально розписано базові гіперпараметри нейронної мережі – розмір «партії» (batch size), кількість епох (epochs), множник ширини (width multiplier) та dropout – включаючи вплив кожного із них на результати тренування моделі.

Тож, із пунктів 3.1-3.4 отримуємо такі параметри для тренування моделі:

- Batch size = 50; epochs = 100; width multiplier = 1.0; dropout = 0.2;
- Конфігурація Adam: $\alpha = 0.00005$; $\beta_1 = 0.9$; $\beta_2 = 0.999$; $\epsilon = 10^{-8}$.
- Критерій – Cross-Entropy Loss.
- Розбиття датасету[24] для train/validation/test = 85%/10%/5% відповідно.

П'ятий, останній пункт, демонструє результати навчання, отримані внаслідок застосування вищезазначених гіперпараметрів. Обчислена матриця плутаниць вказує на хорошу стабільність обчислень – усі значення на діагоналях (точність розпізнавань по кожному із класів) лежали у межах від 0.95 до 1. Крім того, отримана точність Ассигасу (датасет збалансований) становить аж 98.842%, що свідчить про вражаюче високу якість розпізнавання дактилем української жестової абетки натренованою у рамках роботи моделлю MobileNetV3-Large, pytorch-реалізацію якої було взято із [23] та код навчання якої висвітлено у додатку А.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. <https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/>
2. <https://en.wikipedia.org/wiki/Perceptron>
3. https://en.wikipedia.org/wiki/Frank_Rosenblatt
4. <https://anatomiesofintelligence.github.io/posts/2019-06-21-organization-mark-i-perceptron>
5. [Vasco Brattka. A Computable Kolmogorov Superposition Theorem](#)
6. [Manuela Nees. Journal of Computational and Applied Mathematics. Approximative versions of Kolmogorov's superposition theorem, proved constructively](#)
7. <https://sites.gatech.edu/omscs7641/2024/01/31/navigating-neural-networks-exploring-state-of-the-art-activation-functions/>
8. Prajit Ramachandran, Barret Zoph, Quoc V. Le. Searching for activation functions. 2017. arXiv: 1710.05941
9. [cs231n.stanford.edu: Deep Learning for Computer Vision. Neural Networks part 1: Setting up the architecture. 2024.](#)
10. [cs231n.stanford.edu: Deep Learning for Computer Vision. Convolution Neural Networks. 2024.](#)
11. <https://ezyang.github.io/convolution-visualizer/>
12. <https://web.mit.edu/wwmath/vectorc/scalar/chain.html>
13. [Bengio, Yoshua and Glorot, Xavier. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of AISTATS 2010, volume 9, pp. 249– 256, May 2010](#)
14. [cs231n.stanford.edu: Deep Learning for Computer Vision. Backpropagation, intuitions. 2024.](#)
15. [cs224n.stanford.edu: Natural Language Processing with Deep Learning. Lecture 5 – Backpropagation and Project Advice. Slides. 2017.](#)
16. [cs231n.stanford.edu: Deep Learning for Computer Vision. Backpropagation, intuitions - Vector, Matrix, and Tensor Derivatives. 2024.](#)
17. <https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c>
18. Andrew Howard et al. Searching for MobileNetV3. 2019. arXiv: 1905.02244
19. S. Ioffe, C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015. arXiv: 1502.03167v3

- 20.S. Santurkar et al. How Does Batch Normalization Help Optimization? 2019.
arXiv: 1805.11604
- 21.[https://www.researchgate.net/publication/347533712_AMC-IoT Automatic Modulation Classification Using Efficient Convolutional Neural Networks for Low Powered IoT Devices](https://www.researchgate.net/publication/347533712_AMC-IoT_Automatic_Modulation_Classification_Using_Efficient_Convolutional_Neural_Networks_for_Low_Powered_IoT_Devices)
- 22.Jie Hu et al. Squeeze-and-Excitation Networks. 2019. arXiv: 1709.01507
- 23.<https://github.com/kuan-wang/pytorch-mobilenet-v3>
- 24.Kondratiuk, S., Krak, I., Kylias, A., Kasianiuk, V., Fingerspelling alphabet recognition using cnns with 3d convolutions for cross platform applications, 2021, Advances in Intelligent Systems and Computing, 1246 AISC, pp. 585-596. DOI: 10.1007/978-3-030-54215-3_37
- 25.<https://medium.com/@siddharth.4oct/intuition-behind-cross-entropy-baee05911e9f>
- 26.D. P. Kingma and J. L. Ba. Adam : A method for stochastic optimization. 2014.
arXiv:1412.6980v9

ДОДАТОК А: код програми для навчання моделі MobileNetV3-Large

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
from PIL import Image
import numpy as np
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
#__all__ = ['MobileNetV3', 'mobilenetv3']

def conv_bn(inp, oup, stride, conv_layer=nn.Conv2d,
norm_layer=nn.BatchNorm2d, nlin_layer=nn.ReLU):
    return nn.Sequential(
        conv_layer(inp, oup, 3, stride, 1, bias=False),
        norm_layer(oup),
        nlin_layer(inplace=True)
    )

def conv_1x1_bn(inp, oup, conv_layer=nn.Conv2d, norm_layer=nn.BatchNorm2d,
nlin_layer=nn.ReLU):
    return nn.Sequential(
        conv_layer(inp, oup, 1, 1, 0, bias=False),
        norm_layer(oup),
        nlin_layer(inplace=True)
    )

class Hswish(nn.Module):
    def __init__(self, inplace=True):
        super(Hswish, self).__init__()
        self.inplace = inplace

    def forward(self, x):
        return x * F.relu6(x + 3., inplace=self.inplace) / 6.

class Hsigmoid(nn.Module):
    def __init__(self, inplace=True):
        super(Hsigmoid, self).__init__()
        self.inplace = inplace

    def forward(self, x):
        return F.relu6(x + 3., inplace=self.inplace) / 6.

class SEModule(nn.Module):
    def __init__(self, channel, reduction=4):
        super(SEModule, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
```



```

        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            Hsigmoid()
            # nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

class Identity(nn.Module):
    def __init__(self, channel):
        super(Identity, self).__init__()

    def forward(self, x):
        return x

def make_divisible(x, divisible_by=8):
    import numpy as np
    return int(np.ceil(x * 1. / divisible_by) * divisible_by)

class MobileBottleneck(nn.Module):
    def __init__(self, inp, oup, kernel, stride, exp, se=False, nl='RE'):
        super(MobileBottleneck, self).__init__()
        assert stride in [1, 2]
        assert kernel in [3, 5]
        padding = (kernel - 1) // 2
        self.use_res_connect = stride == 1 and inp == oup

        conv_layer = nn.Conv2d
        norm_layer = nn.BatchNorm2d
        if nl == 'RE':
            nlin_layer = nn.ReLU # or ReLU6
        elif nl == 'HS':
            nlin_layer = Hswish
        else:
            raise NotImplementedError
        if se:
            SELayer = SEModule
        else:
            SELayer = Identity

        self.conv = nn.Sequential(
            # pw
            conv_layer(inp, exp, 1, 1, 0, bias=False),
            norm_layer(exp),
            nlin_layer(inplace=True),
            # dw
            conv_layer(exp, exp, kernel, stride, padding, groups=exp,
bias=False),

```

```

        norm_layer(exp),
        SELayer(exp),
        nlin_layer(inplace=True),
        # pw-linear
        conv_layer(exp, oup, 1, 1, 0, bias=False),
        norm_layer(oup),
    )

    def forward(self, x):
        if self.use_res_connect:
            return x + self.conv(x)
        else:
            return self.conv(x)

class MobileNetV3(nn.Module):
    def __init__(self, n_class=26, input_size=224, dropout=0.2, mode='large',
width_mult=1.0):
        super(MobileNetV3, self).__init__()
        input_channel = 16
        last_channel = 1280
        if mode == 'large':
            # refer to Table 1 in paper
            mobile_setting = [
                # k, exp, c, se, nl, s,
                [3, 16, 16, False, 'RE', 1],
                [3, 64, 24, False, 'RE', 2],
                [3, 72, 24, False, 'RE', 1],
                [5, 72, 40, True, 'RE', 2],
                [5, 120, 40, True, 'RE', 1],
                [5, 120, 40, True, 'RE', 1],
                [3, 240, 80, False, 'HS', 2],
                [3, 200, 80, False, 'HS', 1],
                [3, 184, 80, False, 'HS', 1],
                [3, 184, 80, False, 'HS', 1],
                [3, 480, 112, True, 'HS', 1],
                [3, 672, 112, True, 'HS', 1],
                [5, 672, 160, True, 'HS', 2],
                [5, 960, 160, True, 'HS', 1],
                [5, 960, 160, True, 'HS', 1],
            ]
        elif mode == 'small':
            # refer to Table 2 in paper
            mobile_setting = [
                # k, exp, c, se, nl, s,
                [3, 16, 16, True, 'RE', 2],
                [3, 72, 24, False, 'RE', 2],
                [3, 88, 24, False, 'RE', 1],
                [5, 96, 40, True, 'HS', 2],
                [5, 240, 40, True, 'HS', 1],
                [5, 240, 40, True, 'HS', 1],
                [5, 120, 48, True, 'HS', 1],
                [5, 144, 48, True, 'HS', 1],
                [5, 288, 96, True, 'HS', 2],
                [5, 576, 96, True, 'HS', 1],
                [5, 576, 96, True, 'HS', 1],
            ]

```

```

        else:
            raise NotImplementedError

        # building first layer
        assert input_size % 32 == 0
        last_channel = make_divisible(last_channel * width_mult) if
width_mult > 1.0 else last_channel
        self.features = [conv_bn(3, input_channel, 2, nlin_layer=Hswish)]
        self.classifier = []

        # building mobile blocks
        for k, exp, c, se, nl, s in mobile_setting:
            output_channel = make_divisible(c * width_mult)
            exp_channel = make_divisible(exp * width_mult)
            self.features.append(MobileBottleneck(input_channel,
output_channel, k, s, exp_channel, se, nl))
            input_channel = output_channel

        # building last several layers
        if mode == 'large':
            last_conv = make_divisible(960 * width_mult)
            self.features.append(conv_1x1_bn(input_channel, last_conv,
nlin_layer=Hswish))
            self.features.append(nn.AdaptiveAvgPool2d(1))
            self.features.append(nn.Conv2d(last_conv, last_channel, 1, 1, 0))
            self.features.append(Hswish(inplace=True))
        elif mode == 'small':
            last_conv = make_divisible(576 * width_mult)
            self.features.append(conv_1x1_bn(input_channel, last_conv,
nlin_layer=Hswish))
            # self.features.append(SEModule(last_conv)) # refer to paper
Table2, but I think this is a mistake
            self.features.append(nn.AdaptiveAvgPool2d(1))
            self.features.append(nn.Conv2d(last_conv, last_channel, 1, 1, 0))
            self.features.append(Hswish(inplace=True))
        else:
            raise NotImplementedError

        # make it nn.Sequential
        self.features = nn.Sequential(*self.features)

        # building classifier
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout), # refer to paper section 6
            nn.Linear(last_channel, n_class),
            nn.Softmax(dim=1)
        )

        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.mean(3).mean(2)
        x = self.classifier(x)
        return x

    def _initialize_weights(self):

```

```

# weight initialization
for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out')
        if m.bias is not None:
            nn.init.zeros_(m.bias)
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.ones_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.Linear):
        nn.init.normal_(m.weight, 0, 0.01)
        if m.bias is not None:
            nn.init.zeros_(m.bias)

def custom_loader(path):
    return Image.open(path, formats=["JPEG"])

def perform_training(net, training_set, ep, lr, bs, pretrained=False):
    print("training preparation...")

    train_loader = DataLoader(training_set, batch_size=bs, shuffle=True)

    if pretrained:
        print("Loading pretrained model...")
        state_dict = torch.load('my_first_train.pth')
        net.load_state_dict(state_dict, strict=True)

    epochs = ep
    lF = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)

    net.train()
    print("done.")

    for epoch in range(epochs):
        print(f"epoch {epoch + 1}/{epochs} processing...")
        correct = 0
        total = 0

        m_batches_count = 1
        for inputs, labels in train_loader:
            inputs, labels = inputs.cuda(), labels.cuda()

            optimizer.zero_grad()
            outputs = net(inputs)

            pred_vals, pred_classes = torch.max(outputs.data, 1)
            correct += (pred_classes == labels).sum().item()
            total += labels.size(0)

            loss = lF(outputs, labels)

            loss.backward()
            optimizer.step()

        m_batches_count += 1

```

```

        print(f"done. epoch accuracy: {round(100 * correct / total, 3)}%\n")

    torch.save(net.state_dict(), 'my_first_train.pth')
    return net

def perform_validation(net, validation_set):
    print("performing testing...")
    validation_loader = DataLoader(validation_set, batch_size=32,
    shuffle=True)

    state_dict = torch.load('my_first_train.pth')
    net.load_state_dict(state_dict, strict=True)

    net.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        targets = []
        predictions = []
        for inputs, labels in validation_loader:
            inputs, labels = inputs.cuda(), labels.cuda()
            outputs = net(inputs)

            pred_vals, pred_classes = torch.max(outputs.data, 1)
            correct += (pred_classes == labels).sum().item()
            total += labels.size(0)

        targets = np.concatenate([targets,
labels.detach().cpu().numpy()])
        predictions = np.concatenate([predictions,
pred_classes.detach().cpu().numpy()])

        cm = confusion_matrix(y_true=targets, y_pred=predictions,
normalize="true")
        cm = np.round(cm, 3)
        class_names = ['a', 'б',
'в', 'г', 'е', 'ж', 'и', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т', 'у', 'ф', 'х', 'ц', 'ч', 'ш', 'ь', 'ю', 'я', 'і']
        cmp = ConfusionMatrixDisplay(cm, display_labels=class_names)

        ax = plt.subplot()
        plt.rcParams.update({'font.size': 6})
        label_font = {'size': '13'}
        ax.set_xlabel('Predicted labels', fontdict=label_font)
        ax.set_ylabel('Observed labels', fontdict=label_font)

        title_font = {'size': '16'}
        ax.set_title('Confusion Matrix', fontdict=title_font)
        cmp.plot(ax=ax)
        plt.show()

    print(f"Finished. Accuracy: {round(100 * correct / total, 3)}%")

if __name__ == '__main__':
    net = MobileNetV3(n_class=26, dropout=0.2, width_mult=1.0)

```

```
net.cuda(0)

#print('mobilenetv3:\n', net)
#print('Total params: %.2fM' % (sum(p.numel() for p in
net.parameters())/1000000.0))

data_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

dataset = datasets.ImageFolder('dataset', transform=data_transform,
loader=custom_loader)

train_subset, validation_subset, test_subset =
random_split(dataset=dataset, lengths=(0.85, 0.1, 0.05),
generator=torch.Generator().manual_seed(42))

#perform_training(net, train_subset, ep=100, lr=0.00005, bs=50,
pretrained=False)
#perform_validation(net, validation_subset)
perform_validation(net, test_subset)
```