

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ
РОЗПІЗНАВАННЯ ОБ'ЄКТІВ НА ЗОБРАЖЕННІ НА ОСНОВІ
СІАМСЬКИХ НЕЙРОННИХ МЕРЕЖ
RESEARCH AND SOFTWARE IMPLEMENTATION OF
OBJECT RECOGNITION IN IMAGES BASED ON SIAMESE
NEURAL NETWORKS

Виконав: студент 2 курсу, групи КНТ-223м

Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Системи штучного інтелекту

ПРИМАКОВ В.В.

(ПРІЗВИЩЕ та ініціали)

Керівник

СТЕПАНЕНКО О.О.

(ПРІЗВИЩЕ та ініціали)

Рецензент

ЗЕЛЕНЬОВА І.Я.

(ПРІЗВИЩЕ та ініціали)

(назва освітньої програми (спеціалізації))

« » 2024 року

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СТЕПАНЕНКО О.О., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання «20» вересня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	4-5 тижні	Розділ 2
4	Розробка архітектури програми.	6 тиждень	Розділ 3
5	Розробка програми.	7-8 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програм	9 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	10-11 тижні	Додатки
8	Нормоконтроль та рецензування.	12 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка) _____ Владислав ПРИМАКОВ
(підпис) (Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи) _____ Олександр СТЕПАНЕНКО
(підпис) (Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістр:
90 с., 12 табл., 42 рис., 2 дод., 11 джерел.

СІАМСЬКІ НЕЙРОНІ МЕРЕЖІ, ЗГОРТКОВІ МЕРЕЖІ,
КОНТРАСТНА ФУНКЦІЯ ВТРАТ, ТРИПЛЕТНА ФУНКЦІЯ ВТРАТ,
ОНЛАЙН СТРАТЕГІЇ ВІДБОРУ ПАР/ТРИПЛЕТІВ, DRLIM, PYTORCH,
MNIST.

Об'єктом дослідження є нейронні мережі для зіставлення об'єктів на основі їхніх ембеддингів, зокрема сіамські мережі, що застосовуються для класифікації, пошуку схожих об'єктів та зменшення розмірності даних.

Предметом дослідження є методи, архітектури та стратегії навчання сіамських мереж для задач класифікації образів.

Метою роботи є розробка, реалізація та аналіз ефективності сіамських нейронних мереж для задач зіставлення об'єктів на основі їхніх ембеддингів.

Матеріали, методи та технічні засоби: мова програмування Python, фреймворк Pytorch, вибірка MNIST.

Результати. Розроблено архітектури сіамських мереж з контрастною та триплетною функціями втрат. Мережі реалізовані з використанням PyTorch, що забезпечило гнучкість і масштабованість. Створено механізм вибору позитивних і негативних пар, як Batch Hard.

Наукова новизна роботи полягає у застосуванні генетичного алгоритму для оптимізації параметра відступу в контрастних і триплетних функціях втрат. Це враховує специфіку кожного пакета даних, підвищуючи адаптивність моделі, її здатність до узагальнення та точність у задачах пошуку схожих об'єктів чи класифікації.

Висновки. У результаті роботи було розроблено та протестовано сіамські нейронні мережі для задач розпізнавання об'єктів на зображенні.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 90 pages, 12 tables, 42 pictures, 2 appendixes, 11 sources.

SIAMESE NEURAL NETWORKS, CONVOLUTIONAL NETWORKS, CONTRASTIVE LOSS FUNCTION, TRIPLET LOSS FUNCTION, ONLINE PAIRS/TRIPLETS SELECTION STRATEGIES, DRLIM, PYTORCH, MNIST.

The object of research is neural networks for comparing objects based on their embeddings, in particular Siamese networks used for classification, searching for similar objects and reducing the dimensionality of data.

The subject of the research is methods, architectures and strategies for training Siamese networks for image classification tasks. The aim of the study is to develop, implement and analyse the effectiveness of Siamese neural networks for object matching tasks based on their embeddings.

Materials, methods and techniques: Python programming language, Pytorch framework, MNIST dataset.

Results. The architectures of Siamese networks with contrast and triplet loss functions are developed. The networks are implemented using PyTorch, which ensures flexibility and scalability. A mechanism for selecting positive and negative pairs, such as Batch Hard, was created.

The scientific novelty of the work lies in the use of a genetic algorithm to optimise the indentation parameter in contrast and triplet loss functions. This takes into account the specifics of each data set, increasing the adaptability of the model, its ability to generalise and accuracy in tasks of finding similar objects or classification.

Conclusions. As a result of this work, Siamese neural networks for object recognition in images have been developed and tested.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок.....	8
Вступ.....	9
1 Аналіз предметної області.....	11
1.1 Архітектура сіамської мережі.....	11
1.2 Зменшення розмірності даних шляхом навчання інваріантного відображення	13
1.3 Контрастна функція втрат	15
1.4 Триpletна функція втрат.....	16
1.5 Генерація пар/триpletів онлайн	18
1.6 Використання генетичного алгоритму для оптимізації відступу у функціях втрат	20
1.7 Формулювання завдання до роботи	22
1.8 Висновок	23
2 Матеріали і методи.....	24
2.1 Вибір фреймворку для формування нейронних мереж	24
2.1.1 TensorFlow.....	24
2.1.2 PyTorch	26
2.1.3 Порівняння фреймворків.....	27
2.2 Вибірка для розпізнавання образів.....	29
2.3 Вибір структур нейронних мереж	30
2.3.1 Згорткова мережа для класифікації.....	30
2.3.2 Сіамська мережа з сигмоїдальною функцією активації.....	32
2.3.2 Згорткова мережа для ембеддингів	34
2.4 Висновок	36
3 Опис програми.....	37
3.1 Формування вибірок	37
3.2 Формування функцій втрат	38
3.3 Формування нейронних мереж.....	44

3.4 Етап навчання на тестування	45
3.4 Генетичний алгоритм для оптимізації відступів	49
3.5 Висновок	50
4 Тестування та експериментальне дослідження програм.....	51
4.1 Тестування згорткової мережі	51
4.2 Тестування сіамської нейронної мережі із сигмоїдальною функцією активації	53
4.3 Тестування сіамських нейронних мереж з контрастними функціями втрат.....	54
4.4 Тестування сіамських нейронних мереж з триплетними функціями втрат.....	59
4.5 Порівняння результатів	65
Висновки	66
Перелік джерел посилань	68
Додаток А Текст програми.....	69
Додаток Б Слайди презентації	83

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

BA	– Batch All;
BH	– Batch Hard;
CNN	– Convolutional Neural Network;
DrLIM	– Dimensionality Reduction by Learning an Invariant Mapping;
LA	– Lifted Embedding;
ReID	– Re-Identification;
ГА	– генетичний алгоритм;
НМ	– нейрона мережа;
ШІ	– штучний інтелект;
ШНМ	– штучна нейрона мережа.

ВСТУП

З самого зародження комп'ютерних наук дослідники шукали статистичні інструменти для порівняння двох списків елементів суто математичним або семантичним способом. Для цього в минулому вчені розробили кілька показників і коефіцієнтів подібності, пристосовувавши їх до різних значень у різних контекстах [1].

Для суто статистичного порівняння двох елементів найкраще підходять такі метрики, як евклідова відстань, косинусна відстань або Манхеттенська відстань, оскільки вони підкреслюють геометричні відмінності між двома елементами. Якщо порівняння зосереджене на кореляції, коефіцієнт кореляції Пірсона, коефіцієнт рангової кореляції Спірмена, тощо [1].

Усі ці показники та міри добре працюють, коли їх використовують для порівняння двох списків елементів з однаковими значеннями та типами даних. Однак вони не можуть працювати у випадках, коли два списки містять елементи з різними типами даних та/або різними значеннями, які відносяться до різних ознак. Наприклад, якби нам потрібно було порівняти дані рукописних підписів, зібраних за допомогою блокнотного пристрою, щоб перевірити, чи належать вони одній особі, то вищезгадані заходи не підійшли б. Саме для вирішення такого роду проблем розпізнавання підробок Бромлей та ін. представили сіамську нейронну мережу в 1994 році [1].

З моменту своєї появи ця модель була використана в кількох різних сферах і набула широкого розповсюдження в комп'ютерних науках та інформаційних технологіях. Ця робота має на меті розглянути структури побудови сіамських мереж та процес їх навчання, а також дослідити процес навчання сіамських нейронних мереж на популярній виборці даних MNIST.

Об'єктом дослідження є процес навчання сіамських нейронних мереж для розпізнавання образів.

Предметом дослідження є розробка та оптимізація архітектури сіамських мереж для класифікації даних на основі вибірки MNIST, а також аналіз їх ефективності у вирішенні завдань пошуку схожих зображень.

Для досягнення поставленої мети у роботі вирішувалися такі завдання:

- аналіз теоретичних основ роботи сіамських мереж та їх застосування для задач класифікації та пошуку схожих об'єктів;
- розробка та впровадження кількох архітектур сіамських мереж для роботи з вибіркою MNIST;
- визначення ефективних функцій втрат для навчання сіамських мереж та порівняння їх впливу на результати;
- порівняльний аналіз продуктивності розроблених архітектур на основі таких показників, як точність класифікації;
- висновки щодо доцільності використання кожної з розроблених архітектур для задач класифікації на вибірці MNIST.

Методика дослідження охоплює аналіз літературних джерел, розробку прототипу системи, її тестування та оцінку ефективності.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Архітектура сіамської мережі

Алгоритм сіамської нейронної мережі (НМ) вперше був представлений Бромлі та ін. для виявлення підроблених підписів у 1994 році. До цього Балді та Шовен представили схожу штучну нейронну мережу (ШНМ), здатну розпізнавати відбитки пальців, але під іншою назвою. У дослідженні Бромлі та ін. порівнюючи два рукописні підписи, сіамська нейронна мережа змогла визначити, чи обидва підписи є оригінальними, чи один з них є підробкою [1].

Штучна нейронна мережа – це модель машинного навчання, що складається з декількох шарів нейронів, які є обробними одиницями, натхненними біологічними нейронами. Типові керовані нейронні мережі називаються прямими (feedforward) і базуються на моделі парцептрона. У нейронній мережі прямого поширення кожен нейрон першого шару зчитує вхідне реальне значення, множить його на вагу і надсилає результат усім нейронам наступного шару. У загальноприйнятих уявленнях про нейронні мережі вчені вважають крайній лівий шар першим, а крайній правий – вихідним. Нейрони кожного шару за вхідним шаром виконують ту саму роботу і надсилають результати нейронам наступного шару, поки не дійдуть до останнього шару. У керованій нейронній мережі кінцевий шар надсилає свої результати до одонеуронного вихідного шару, який виробляє результат нейронної мережі в реальних значеннях (рис. 1.1) [1].

Під час навчання НМ порівнює значення, отримані нейронною мережею, з відповідною базовою істиною та обчислює статистичну похибку (зазвичай середньоквадратичну похибку або перехресну ентропійну похибку). Після цього нейронна мережа надсилає помилку назад на попередні шари і відповідно оновлює ваги нейронів, використовуючи техніку, яка називається зворотним поширенням помилки [1].

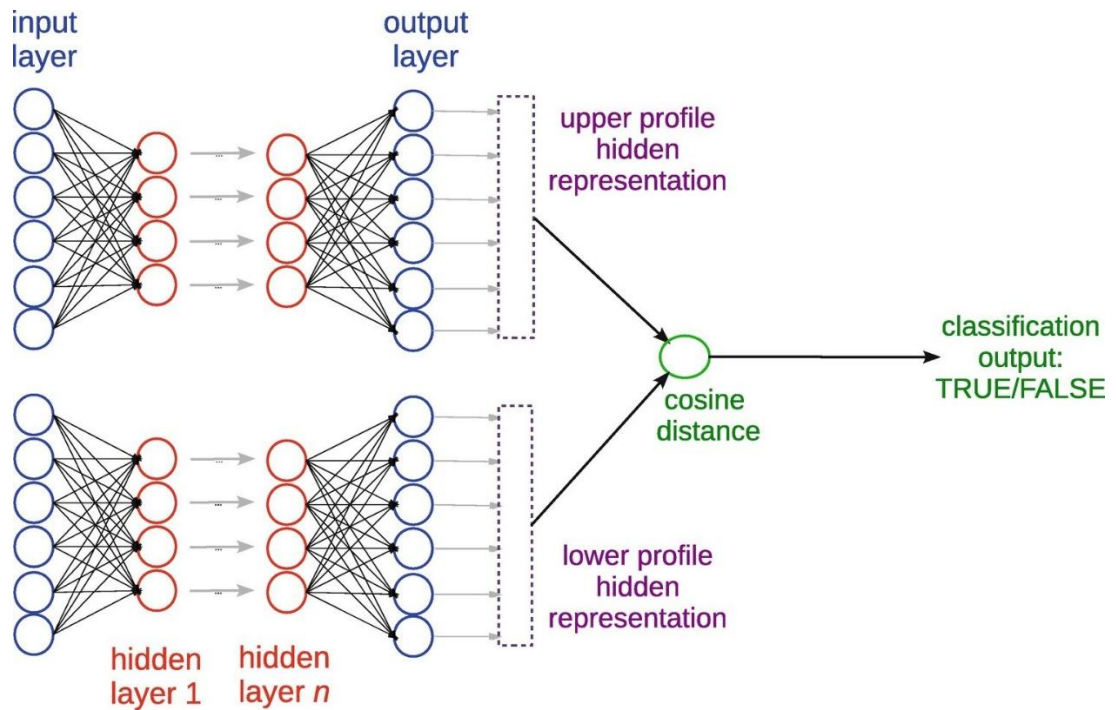


Рисунок 1.1 – Представлення структури моделі сіамської нейронної мережі [1]

Навчання зупиняється, коли нейронна мережа досягає максимальної кількості ітерацій, заданої спочатку. Після того, як модель навчена, її можна застосувати до тестового набору: нейронна мережа обробить кожен екземпляр тестових даних і згенерує прогнозоване значення. Після того, як НМ згенерувала прогнозоване значення для кожного примірника тестового набору, програміст може використати його для обчислення матриці похибок [1].

Прямі нейронні мережі зі зворотним поширенням помилки використовуються і в сіамських нейронних мережах. Архітектура сіамської нейронної мережі, по суті, містить дві ідентичні НМ прямого поширення, з'єднані на виході (рис. 1.1), які працюють паралельно в тандемі. Кожна нейронна мережа містить традиційну модель парцептрона. Під час навчання кожна нейронна мережа зчитує профіль, складений з реальних значень, і обробляє його значення на кожному шарі. Нейронна мережа активує деякі нейрони на основі цих значень, оновлює свої ваги за допомогою зворотного

поширення помилки і в кінці генерує вихідний профіль, який порівнюється з вихідним профілем іншої нейронної мережі [1].

Алгоритм порівнює вихід першої нейронної мережі та вихід другої нейронної мережі через метрику відстані (рис. 1.1) – косинусну відстань у вихідній моделі або якусь іншу. За допомогою цієї міри подібності нейромережа визначає, що два профілі є різними або подібними. Остаточне вихідне значення можна порівняти з відповідним істинним значенням.

1.2 Зменшення розмірності даних шляхом навчання інваріантного відображення

Сучасні програми постійно розширюють використання складних, високовимірних даних. Зменшення розмірності має на меті перетворення даних високої розмірності у низьковимірне представлення таким чином, щоб схожі об'єкти вхідних даних відображалися у близькі точки на просторі. Більшість існуючих методів зменшення розмірності мають два недоліки. По-перше, вони не створюють функцію (або відображення) від вхідних даних до множини, яку можна застосувати до нових точок, зв'язок яких з навчальними точками невідомий. По-друге, багато методів припускають наявність значущої (і обчислюваної) метрики відстані в вхідний простір [2].

Іншим обмеженням сучасних методів є те, що вони мають тенденцію до кластеризації точок у вихідному просторі, іноді досить щільно, щоб вважатися виродженими розв'язками. Натомість, іноді бажано знайти простір, який рівномірно покритий вибірками [2].

Метод, який називається Dimensionality Reduction by Learning an Invariant Mapping (DrLIM), забезпечує розв'язання вищезазначених проблем. DrLIM – це метод вивчення глобально когерентної нелінійної функції, яка відображає дані на низьковимірний простір. Метод має чотири основні характеристики:

– йому потрібні лише сусідські зв'язки між навчальними вибірками. Ці зв'язки можуть базуватися на попередніх знаннях або ручному маркуванні і не залежати від будь-якої метрики відстані;

– він може навчатися функціям, інваріантним до складних нелінійних перетворень вхідних даних, таких як зміни освітлення та геометричні викривлення;

– навчена функція може бути використана для відображення нових зразків, які не були помічені під час навчання, без попередніх знань;

– відображення, що генерується функцією, є в деякому сенсі «гладкими» і когерентними у вихідному просторі [2].

Метод використовується для вивчення параметрів W параметризованої функції G_W . Метод використовує енергетичну модель, яка базується на заданих відношеннях сусідства для навчання функції відображення. Для сімейства функцій G , параметризованої W , метою є знайти таке значення W , яке відображає набір високовимірних вхідних даних на простір так, щоб евклідова відстань між точками за формулою (1.1) у просторі наближала «семантичну схожість» вхідних даних у вихідному просторі, як це задано набором відношень сусідства:

$$D_W(X_1, X_2) = \|G_W(X_1) - G_W(X_2)\|_2. \quad (1.1)$$

Жодних припущень про функцію G_W не робиться, окрім того, що вона є диференційованою за W [2]. Більш конкретно, нехай задано набір вхідних векторів I за формулою (1.2):

$$I = \{X_1, \dots, X_n\}, X_i \in R^D, \forall i = 1, \dots, n, \quad (1.2)$$

де n – кількість екземплярів;

D – розмірність екземпляру.

Треба знайти параметричну функцію G_W за формулою (1.3):

$$G_W: R^D \rightarrow R^d, d \ll D. \quad (1.3)$$

Яка має такі властивості:

- прості міри відстані у вихідному просторі (наприклад, евклідова відстань) повинні апроксимувати відносини сусідства у вхідному просторі;
- відображення не повинно обмежуватися реалізацією простих мір відстані у вхідному просторі і повинно мати можливість вивчати інваріанти до складних перетворень;
- функція має бути вірна навіть для зразків, для яких невідомі відносини між сусідніми зразками [2].

1.3 Контрастна функція втрат

На відміну від традиційних систем навчання, де функція втрат є сумою над екземплярами, контрастна функція втрат пробігає над парами екземплярів. Нехай $X_1, X_2 \in I$ – пара вхідних векторів, що подаються на вхід системи. Нехай Y – бінарна мітка, що присвоюється цій парі. $Y = 0$, якщо X_1 та X_2 вважаються подібними, і $Y = 1$, якщо вони вважаються відмінними. Визначимо параметризовану функцію відстані, що навчається D_W між X_1, X_2 як евклідову (1.1). Для скорочення записів $D_W(X_1, X_2)$ записується D_W . Тоді функція $L(W)$ втрат у найзагальнішому вигляді має вигляд за формулами (1.4-1.5):

$$L(W) = \sum_{i=1}^P L(W, (Y, X_1, X_2)^i), \quad (1.4)$$

$$L(W, (Y, X_1, X_2)^i) = (1 - Y)L_S(D_W^i) + YL_D(D_W^i), \quad (1.5)$$

де $(Y, X_1, X_2)^i$ – i -та помічена пара екземплярів;

L_S – функція часткових втрат для пари схожих екземплярів;

L_D – функція часткових втрат для пари відмінних екземплярів;

P – кількість навчальних пар [2].

L_S та L_D повинні бути розраховані таким чином, щоб мінімізація L по відношенню до W призвела до низьких значень D_W для схожих пар та високих значень D_W для відмінних пар.

Точна функція $L(W, Y, X_1, X_2)$ втрат має вигляд за формулою (1.6):

$$L(W, Y, X_1, X_2) = (1 - Y) \frac{1}{2} (D_W)^2 + Y \frac{1}{2}, m > 0, \quad (1.6)$$

де m – відступ який визначає радіус навколо $G_W(X)$ [2].

Тобто відмінні пари дають внесок у функцію втрат лише тоді, коли їхня відстань знаходиться в межах цього радіуса.

1.4 Триpletна функція втрат

Останніми роками person re-identification (ReID) привертає значну увагу в спільноті комп'ютерного зору. Особливо з розвитком глибокого навчання було запропоновано багато нових підходів для вирішення цього завдання. У багатьох аспектах ідентифікація осіб подібна до пошуку зображень, де було досягнуто значного прогресу, і де глибоке навчання занесло багато змін. Одним з яскравих прикладів у недавній літературі є FaceNet [3], convolutional neural network (CNN), яка використовується для навчання ембеддингів для зображень. Ключовим компонентом FaceNet є використання триплетних втрат, введених Вайнбергером та Солом, для навчання ШНМ як функції ембеддингу. Триpletна функція втрат оптимізує простір ембеддингів таким чином, що точки даних з однакового класу знаходяться ближче одна до одної, ніж точки із різних класів (рис. 1.2) [4].

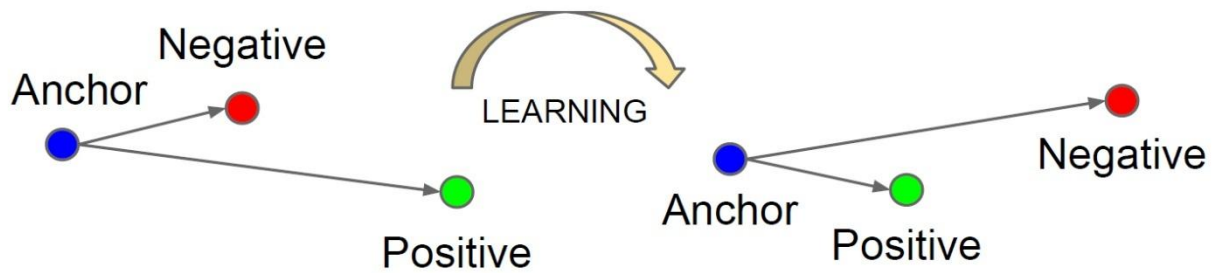


Рисунок 1.2 – Триплетна функція втрат [3]

Вайнбергер та Сол досліджують цю тему з явною метою виконання класифікації k -найближчих сусідів у вивченому просторі ембеддингів та пропонують Large Margin Nearest Neighbor loss (LMNN) для оптимізації G_W [4]. Функцію втрат можна представити у вигляді формул (1.7-1.9):

$$L_{LMNN}(W) = (1 - \mu)L_{pull}(W) + \mu L_{push}(W), \quad (1.7)$$

$$L_{pull}(W) = \sum_{i,j \in T(i)} D_{i,j}, \quad (1.8)$$

$$L_{push}(W) = \sum_{\substack{a,n \\ y_a \neq y_n}} [m + D_{a,T(a)} - D_{a,n}]_+. \quad (1.9)$$

Функція $L_{LMNN}(W)$ складається з тягнучого члена, що притягує точки даних і до їх цільового сусіда $T(i)$ з того ж класу, та штовхаючого члена, що відсуває точки даних з іншого класу k ще далі. Оскільки мотивацією було використання методу найближчих сусідів для класифікації розрізнених кластерів одного й того ж класу було, досягнуто шляхом вибору фіксованих цільових сусідів на початку навчання. Оскільки ця властивість є шкідливою для завдань пошуку, таких як ідентифікація обличчів і ReID, FaceNet [3] запропонував модифікацію функції втрат LMNN під назвою триплетна функція втрат (Triplet loss), яка має таку формулу (1.10) [4]:

$$L_{tri}(W) = \sum_{y_a=y_p \neq y_n} a_{p,n} [m + D_{a,p} - D_{a,n}]_+. \quad (1.10)$$

Ця функція втрат забезпечує, що, маючи якірну точку X_a , проєкція позитивної точки X_p , яка належить до того ж класу Y_a , буде ближчою до проєкції якірної точки, ніж проєкція негативної точки, яка належить до іншого класу Y_n , принаймні на відстань, що перевищує деякий поріг m . Якщо цю функцію втрат оптимізувати на всьому наборі даних достатньо довго, то зрештою всі можливі пари (X_a, X_p) будуть опрацьовані та притягнуті одна до одної, роблячи тягнучого члена зайвим. Перевага цього формулювання полягає в тому, що хоча зрештою всі точки одного класу утворять єдиний кластер, їм не обов'язково стискатися в одну точку, їм лише потрібно бути ближчими одна до одної, ніж до будь-якої точки з іншого класу [4].

1.5 Генерація пар/триплетів онлайн

Основним застереженням щодо втрати триплетів є те, що зі збільшенням набору даних можлива кількість триплетів зростає в кубічній прогресії, що робить досить тривале навчання непрактичним. Що ще гірше, G_W відносно швидко навчається правильно відображати більшість тривіальних триплетів, що робить значну частину всіх триплетів неінформативними. Таким чином, видобуток складних триплетів стає критично важливим для навчання. Інтуїтивно зрозуміло, що повторення того, що люди в одязі різного кольору – це різні люди, нічому не вчить, тоді як бачення схожих, але різних людей (жорсткі негативи) або фотографій однієї і тієї ж людини в абсолютно різних позах (жорсткі позитиви) суттєво допомагає зрозуміти поняття «та сама людина» [4].

У класичній реалізації, як тільки вибрано певний набір B триплетів, їх образи складаються у пакет розміром $3B$, для якого обчислюються $3B$ ембеддингів, які, в свою чергу, використовуються для створення B термів, що

вносять свій внесок у функцію втрат. Враховуючи той факт, що формують пакети (batch), випадковим чином відбираючи P класів, а потім випадковим чином відбираючи K зображень кожному класу, в результаті чого отримують партію PK зображень. Тепер, для кожної вибірки a в пакеті, ми можемо вибрати найбільш важкі позитивні та найбільш важкі негативні зразки в пакеті, які називають Batch Hard (BH), можна знайти за формулою (1.11):

$$L_{BH}(W; X) = \overbrace{\sum_{i=1}^P \sum_{a=1}^K}^{all\ anchors} \left[m + \overbrace{\max_{p=1 \dots K} D(G_W(X_a^i), G_W(X_p^i))}^{hardest\ positive} - \overbrace{\min_{\substack{j=1 \dots P \\ n=1 \dots K \\ j \neq i}} D(G_W(X_a^i), G_W(X_n^j))}^{hardest\ negative} \right]_+, \quad (1.11)$$

де X – партія екземплярів у пакеті;

x_i^j – відповідає j -тому екземпляру з i -того класу [4].

Стратегія ВН призводить до того, що втрати на вибірку екземплярів, втричі довші порівняно з традиційною формулою. Крім того, вибрані трійки можна вважати помірними трійками, оскільки вони є найскладнішими в межах невеликої підмножини даних, а це саме це найкраще підходить для навчання з втратою трійок. Це нове формулювання вибірки партії одразу наштовхує на іншу альтернативу – просто використовувати всі можливі $PK(PK - K)(K - 1)$ комбінації триплетів, яку називають Batch All (BA) і які можна знайти за формулою (1.12) [4]:

$$L_{BA}(W; X) = \overbrace{\sum_{i=1}^P \sum_{a=1}^K}^{all\ anchors} \overbrace{\sum_{\substack{p=1 \\ p \neq a}}^K}^{all\ pos} \overbrace{\sum_{\substack{j=1 \\ j \neq i}}^P \sum_{n=1}^K}^{all\ negatives} \left[m + D(G_W(X_a^i), G_W(X_p^i)) - D(G_W(X_a^i), G_W(X_n^j)) \right]_+. \quad (1.12)$$

Важливо зазначити, що як L_{BH} , так і L_{BA} все ще точно відповідають стандартній функції втрат трійок у межі нескінченного навчання. Обидві функції – max і min – є неперервними та диференційованими майже всюди, що означає, що їх можна використовувати в моделі, яка навчається за допомогою стохастичного (суб-)градієнтного спуску, без будь-яких проблем.

Насправді, ці функції вже широко доступні в популярних фреймворках для глибокого навчання для реалізації max-пулінгу та нелінійності ReLU [4].

Найбільш схожою на функції L_{BG} та L_{BA} є функція втрат Lifted Embedding (LE) яка формує пакет із трійок, але розглядає всі інші, крім пари «якір-позитив», як негативи. Крім того, розглядаються традиційні пакети 3B, використовуючи всі можливі негативні приклади, але лише одну позитивну пару на кожну трійку. Це підводить нас до пропозиції узагальнити функцію L_{LE} , засновану на пакетах PK , які враховують усі пари «якір-позитив», наступним чином за формулою (1.13) [4]:

$$L_{LE}(W; X) = \frac{\overbrace{\sum_{i=1}^P \sum_{a=1}^K}^{all\ anchors}} \left[m + \frac{\overbrace{\log \sum_{\substack{p=1 \\ p \neq a}}^K e^{D(G_W(X_a^i), G_W(X_p^i))}}^{all\ positive}} - \frac{\overbrace{\log \sum_{\substack{j=1 \\ j \neq i}}^P \sum_{n=1}^K e^{m-D(G_W(X_a^i), G_W(X_n^j))}}^{all\ negative}} \right]_+ \quad (1.13)$$

1.6 Використання генетичного алгоритму для оптимізації відступу у функціях втрат

Одним із важливих гіперпараметрів контрастних і триплетних функцій втрат є відступ (margin), який визначає мінімальну різницю між відстанями між позитивними та негативними парами. Відступ значно впливає на ефективність навчання, оскільки надто мале чи надто велике значення може спричинити погане узагальнення або ненадійну збіжність.

Для автоматичного пошуку оптимального значення відступу пропонується використовувати генетичний алгоритм (ГА), який здатний ефективно знаходити глобальні оптимуми в складних просторах параметрів.

Генетичний алгоритм для оптимізації відступів працює таким чином: для кожного пакета тренувальних даних визначається оптимальний параметр відступу. Спочатку генерується початкова популяція випадкових значень відступу. Для кожного значення з цієї популяції обчислюється функція втрат, яка враховує відступ як параметр. Після цього застосовуються основні оператори генетичного алгоритму: селекція для вибору найкращих індивідів,

кросовер для створення нових значень на основі батьків, і мутація для внесення випадкових змін. Процес повторюється протягом заданої кількості поколінь або до досягнення збіжності. Після завершення оптимізації для кожного пакета отримане значення відступу додається до списку. На етапі тестування для всієї вибірки розраховується середнє значення відступу, яке використовується для оцінки роботи моделі. Такий підхід дозволяє врахувати локальні особливості кожного пакета і забезпечити узгодженість під час тестування. Класичну схему роботи ГА можна побачити на рис. 1.3.



Рисунок 1.3 – Класична схема роботи ГА

1.7 Формулювання завдання до роботи

Метою даної роботи є створення та дослідження різних архітектур сіамських нейронних мереж для порівняння їх ефективності при розв'язанні задачі класифікації на якійсь базі даних. Для досягнення цієї мети необхідно виконати наступні завдання.

Створити декілька архітектур сіамських мереж із різними налаштуваннями, що включають:

- згорткові нейронні мережі як базові блоки для побудови ембеддингів;
- різні функції втрат, такі як контрастна та триплетна, для навчання моделей;
- варіації гіперпараметрів, наприклад, розмірності ембеддингів, кількості шарів, функцій активації та оптимізаторів.

Розробити алгоритми для формування навчальних даних, включаючи стратегії вибору:

- формування набору пар;
- формування триплетів;
- Реалізація стратегії відбору ембеддингів у пакеті Batch Hard.

Налаштувати параметри навчання для кожної архітектури, зокрема:

- початкові значення ваг;
- розмір пакету;
- коефіцієнт навчання;
- стратегії оптимізації.

Експериментальне порівняння моделей:

- оцінити продуктивність кожної моделі за допомогою тестової вибірки;
- визначити точність та інші показники якості роботи моделей;
- дослідити вплив різних параметрів архітектури та функцій втрат на результат;

– порівняти отримані результати для різних сіамських мереж.

1.8 Висновок

У даному розділі було розглянуто архітектуру сіамських нейронних мереж, які використовуються для задач пошуку схожості між об'єктами. Було детально проаналізовано функції втрат, такі як контрастна функція та триплетна функція, які забезпечують навчання моделей шляхом оптимізації відстаней між ембеддингами. Окрім цього, увагу було приділено стратегіям онлайн-вибірки прикладів із пакету, що дозволяють ефективніше використовувати дані та пришвидшують процес навчання.

Розглянуті концепції та методи створюють основу для подальшого проєктування й аналізу сіамських мереж у задачах класифікації та порівняння зображень.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір фреймворку для формування нейронних мереж

Python є одним із найпопулярніших мов програмування у сфері машинного навчання, зокрема для побудови та експериментів із нейронними мережами. Python має широкий набір фреймворків, які значно спрощують розробку та навчання моделей глибокого навчання. Найпопулярнішими серед них є TensorFlow, PyTorch, Keras, та JAX. Вони забезпечують ефективну реалізацію складних математичних операцій, автоматичну обробку градієнтів та підтримку паралельних обчислень. Python має інструменти для роботи з даними, візуалізації та експериментального аналізу, такі як NumPy, Pandas, Matplotlib і Seaborn. Вони дозволяють зручно працювати з даними на всіх етапах дослідження [5].

2.1.1 TenserFlow

Google Brain створив фреймворк TensorFlow з відкритим вихідним кодом, який широко використовується в багатьох проєктах зі штучного інтелекту, машинного та глибокого навчання. Розгалужена та адаптована екосистема інструментів, бібліотек і ресурсів спільноти TensorFlow дозволяє дослідникам покращувати стан машинного навчання. У той же час розробники можуть легко створювати та впроваджувати програми на основі машинного навчання [5].

Плюси Tensorflow:

- відкритий код. TensorFlow – це фреймворк із відкритим вихідним кодом, що означає, що він безкоштовний і може використовуватися будь-ким без необхідності укладати договір із постачальником заздалегідь;

- налагодження. Однією з ключових функцій TensorFlow є TensorBoard. TensorBoard – це те, що забезпечує візуалізацію та інструменти, необхідні для експериментів;

- tensorBoard забезпечує візуалізацію та інструменти, необхідні для експериментів з машинним навчанням у глибокому навчанні, що значно полегшує налагодження коду TensorFlow;

- коли Google оголосила про випуск TensorFlow 2.0, вони також оголосили Keras офіційним API високого рівня TensorFlow. Це дозволяє користувачам додавати функціональні можливості високого рівня до свого коду та користуватися перевагами простоти Keras;

- масштабованість. TensorFlow не обмежується одним пристроєм. Користувачі можуть розробляти системи за допомогою фреймворку та очікувати, що він працюватиме на периферійних пристроях так само ефективно, як і на будь-якій іншій складній машині;

- сумісність. Фреймворк сумісний з кількома мовами програмування; Таким чином, користувачі можуть працювати мовою програмування, яка їм найбільш зручна;

- tensorflow пропонує надійний API для Python, а також API для Javascript, C++ і Java, які не гарантують зворотну сумісність. Крім того, існують пакети сторонніх розробників, які надають прив'язки до мови для C#, Haskell, Julia, MATLAB, Object Pascal, R, Scala, Rust, OCaml і Crystal;

- паралелізм. TensorFlow розглядається як бібліотека апаратного прискорення. Фреймворк використовує різні стратегії розподілу в системах GPU і CPU. Це означає, що користувач може вільно використовувати будь-яку архітектуру, яку він хоче використовувати. Зауважте, що система за замовчуванням використовує GPU, якщо це не зазначено явно; цей процес певною мірою зменшує використання пам'яті [6].

Недоліки Tensorflow:

- часті оновлення. Накладні витрати користувача на встановлення та прив'язку TensorFlow до існуючої системи збільшуються через часті оновлення;

- неузгоджені омоніми. TensorFlow пропонує омоніми, які важко зрозуміти та використовувати, оскільки вони мають ідентичні назви, але різні

реалізації. Наприклад, `tf.nn.conv2d` і `tf.layers.Conv2d` мають однакові назви, але дещо різні налаштування, які можуть змінити ваші результати;

- відсутні символічні петлі. TensorFlow називають API низького рівня – і це не дарма. Фреймворк не обробляє автоматично ітерації, які призводять до символічних циклів. Однак вона використовується для певних послідовностей, що робить її зручною системою;

- обмежена підтримка GPU. Фреймворк підтримує лише NVIDIA та Python для програмування GPU. Додаткової підтримки немає;

- низька швидкість впровадження. У всіх апаратних конфігураціях TensorFlow постійно вимагає найбільше часу для навчання різних типів нейронних мереж порівняно з конкурентами [6].

2.1.2 PyTorch

PyTorch – це найновіший серед сучасних фреймворків для глибокого навчання. Розроблений дослідницькою групою штучного інтелекту Facebook (тепер Meta) та відкритий для загального доступу у 2016 році, PyTorch тепер є частиною Linux Foundation [6].

За ці роки PyTorch здобув репутацію фреймворка, що поєднує простоту, гнучкість та ефективність. Він дозволяє розробникам швидко створювати складні нейронні мережі для таких застосувань, як комп'ютерний зір і обробка природної мови [6].

Переваги PyTorch:

- *pythonic*. PyTorch має «пітонічний» характер, що робить його дуже зрозумілим для розробників на Python. Ця особливість є перевагою, оскільки значна частина роботи з глибоким навчанням виконується саме на Python;

- легкість у навчанні. Синтаксис PyTorch надзвичайно простий та інтуїтивно зрозумілий. Фахівці часто відзначають, що PyTorch має пологий навчальний поріг – дехто навіть вважає його одним із найлегших для освоєння фреймворків глибокого навчання;

- простота відлагодження. PyTorch підтримує використання інструментів відлагодження Python, таких як pdb та ipdb. Програмісти також можуть використовувати PyCharm, IDE для Python, для відлагодження, оскільки PyTorch створює обчислювальний графік у реальному часі;

- сильна спільнота. Попри те, що PyTorch є одним із найновіших фреймворків глибокого навчання, він швидко сформував віддану спільноту розробників. Ці люди наполегливо працюють над тим, щоб документація була добре структурованою та корисною для інших розробників;

- паралелізм даних. Паралелізм даних у PyTorch надзвичайно ефективний, оскільки він дозволяє користувачам розбивати дані на пакети та надсилати їх на обробку на різні GPU. Завдяки цьому підходу PyTorch може значно розвантажити процесор, передаючи основну частину роботи на графічний процесор [6].

Недоліки PyTorch:

- відсутність візуалізації даних. TensorBoard у TensorFlow дозволяє користувачам візуалізувати процес навчання, але в PyTorch подібного інструменту немає – користувачі змушені використовувати сторонні інструменти;

- потрібен сервер API для розгортання. Немає вбудованого фреймворка для безпосереднього розгортання моделей у вебсередовищі, тому потрібно використовувати сторонній бекенд-сервер, такий як Flask або Django.

2.1.3 Порівняння фреймворків

У табл. 2.1 зведено основні критерії для порівняння розглянутих фреймворків.

Таблиця 2.1 – Порівняльна характеристика фреймворків [5]

Критерії	PyTorch	TensorFlow
Ключові відмінності	Підходить для тестування та дослідження	Готовий до розгортання та виробництва
Архітектура	Динамічний обчислювальний графік, який можна змінити в будь-який час	Граф статичних обчислень, який визначається лише один раз і використовується знову
Простота використання	Інтуїтивно зрозумілий інтерфейс користувача на основі Python	Швидше навчання, але більше функцій, ніж раніше
Гнучкість і філософія дизайну	Простота та адаптивність	Продуктивність і масштабованість
Вплив на практичну побудову моделі	Швидкі ітерації та налагодження моделі	Функції надійного обслуговування TensorFlow і розгортання TensorFlow Lite
Швидкість і ефективність	Для невеликих моделей і розробок це, як правило, швидше	Він призначений для масштабних моделей
Масштабованість	Ідеально підходить для малих і середніх програм і досліджень	Висока масштабованість для широкого розосередженого навчання.
Популярність	Стає все більш популярним у наукових і наукових колах	Він широко використовується в бізнесі та промисловому контексті
Спільнота та підтримка	Сильна підтримка наукової спільноти	Це велика спільнота з великою кількістю ресурсів

Як говорять в дослідницьких колах: Pytorch використовують для дослідження моделей через свою гнучкість та універсальність, а Tenserflow використовують для кінцевого продукту. Тому в цій роботі було обрано саме Pytorch.

2.2 Вибірка для розпізнавання образів

База даних MNIST, що містить зображення рукописних цифр включає навчальний набір із 60 000 прикладів і тестовий набір із 10 000 прикладів. Це підмножина більшого набору даних, доступного в NIST. Цифри були нормалізовані за розміром і центровані в зображенні фіксованого розміру. Ця база даних добре підходить для тих, хто хоче випробувати методи навчання та розпізнавання образів на реальних даних, витрачаючи мінімум зусиль на передобробку та форматування [7].

База даних MNIST була створена на основі спеціальних баз даних NIST – Special Database 3 (SD-3) і Special Database 1 (SD-1), які містять двійкові зображення рукописних цифр. Спочатку NIST призначив SD-3 як навчальний набір, а SD-1 як тестовий набір. Однак SD-3 значно чистіший і легший для розпізнавання, ніж SD-1. Це пов'язано з тим, що SD-3 був зібраний серед співробітників Бюро перепису населення, тоді як SD-1 збирався серед старшокласників [7].

SD-1 містить 58 527 зображень цифр, написаних 500 різними авторами. На відміну від SD-3, де дані кожного автора розташовані послідовно, у SD-1 дані перемішані. Ідентифікаційні дані авторів SD-1 доступні, і ми використали цю інформацію для розділення даних за авторами. Потім SD-1 був поділений на дві частини: символи, написані першими 250 авторами, потрапили в новий навчальний набір. Решта 250 авторів були включені в тестовий набір. Таким чином, було отримано два набори приблизно по 30 000 прикладів кожен [7].

Для задачі тестування малих сіамських мереж ця вибірка підходить, також плюсом є те, що вибірка має 1 кольоровий шар, що зменшує навантаження на вимоги до НМ, а також має вирівняні зображення.

2.3 Вибір структур нейронних мереж

2.3.1 Згорткова мережа для класифікації

Щоб порівнювати якість роботи сіамських мереж було вирішено також створити згорткову мережу, шари якої представлено у табл. 2.2.

Таблиця 2.2 – Шари згорткової мережі для класифікації

Layer	Depth	Filter Height	Filter Width	Stride	Output Shape	Weights
Input	1				1×28×28	
Conv2d	32	5	5	1	32×28×28	832
MaxPool2d	32			2	32×14×14	
Conv2d	64	5	5	1	64×10×10	53248
MaxPool2d	64			2	64×5×5	
Flatten	1024				1024	
Linear	256				256	262400
Linear	256				256	65792
Linear	10				10	2570

Використання згорткових шарів забезпечує автоматичне виділення суттєвих ознак зображення, таких як контури, текстур та форми. Повнозв'язні шари використовуються для класифікації. Останній шар з функцією активації Softmax перетворює вихід моделі в імовірності для кожного класу. Загальна кількість параметрів такої мережі: 384842.

На рис. 2.1 можна побачити візуалізовану схему такої мережі.

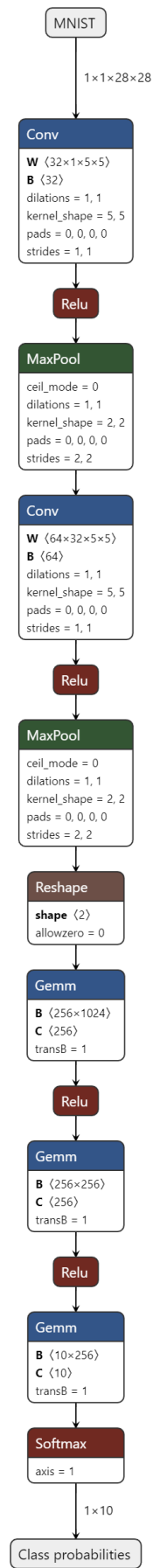


Рисунок 2.1 – Схема згорткової мережі для класифікації

2.3.2 Сіамська мережа з сигмоїдальною функцією активації

Оскільки вибірка не дуже складна, то також можна додати до порівняння сіамську мережу для бінарної класифікації, шари якої приведено у табл. 2.3.

Таблиця 2.3 – Шари сіамської мережі із сигмоїдальною функцією активації

Layer	Depth	Filter Height	Filter Width	Stride	Output Shape	Weights
Input	1				1×28×28	
Conv2d	32	5	5	1	32×28×28	832
MaxPool2d	32			2	32×14×14	
Conv2d	64	5	5	1	64×10×10	53248
MaxPool2d	64			2	64×5×5	
Flatten	1024				1024	
Cat	2048				2048	
Linear	256				256	524544
Linear	256				128	65792
Linear	128				1	256

На вхід цієї мережі подається два образи, які проходять через згорткові шари для виділення ембеддингів, а потім ці ембеддинги об'єднуються в один вектор (шар Cat) та проходять через декілька повнозв'язних шарів і в кінці активуються за допомогою сигмоїдальної функції для бінарної класифікації. Слід зазначити, що згорткові шари не навчаються окремо для першого та другого образу, а мають одні і ті ж самі ваги під час навчання. Загальна кількість параметрів такої мережі: 644672.

Схему такої мережі візуалізовано на рис. 2.2.

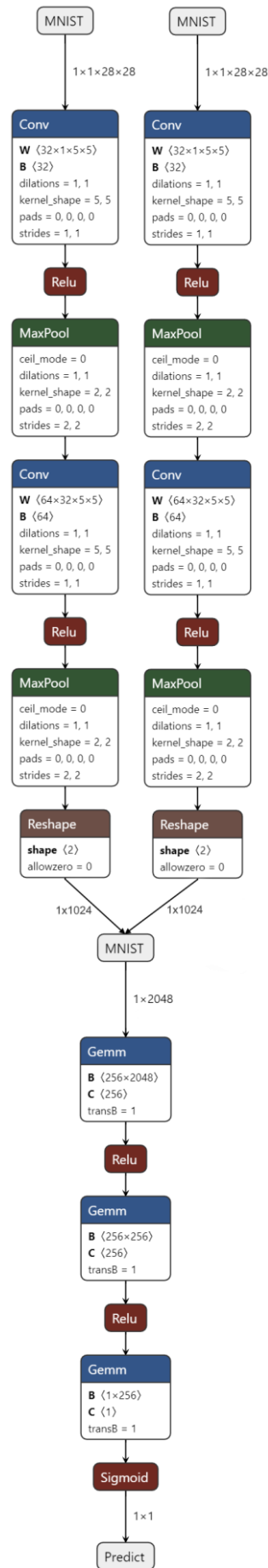


Рисунок 2.2 – Схема сіамської мережі з сигмоїдальною функцією активації

2.3.2 Згорткова мережа для ембеддингів

У цій роботі для задачі пошуку ембеддингів обрана згорткова нейронна мережа, шари якої приведено у табл. 2.4, що складається з двох основних компонентів: згорткової нейронної мережі та повнозв’язних шарів.

Таблиця 2.4 – Шари згорткової мережі для ембеддингів

Layer	Depth	Filter Height	Filter Width	Stride	Output Shape	Weights
Input	1				1×28×28	
Conv2d	32	5	5	1	32×28×28	832
MaxPool2d	32			2	32×14×14	
Conv2d	64	5	5	1	64×10×10	53248
MaxPool2d	64			2	64×5×5	
Flatten	1024				1024	
Linear	256				256	262400
Linear	256				128	65792
Linear	128				128	16394

Модель складається з двох згорткових шарів та трьох лінійних (повнозв’язних) шарів, що дозволяє ефективно зменшувати розмірність даних та отримувати компактні ембеддинги. Пулінгові шари дозволяють зменшити розмірність зображення, зберігаючи важливу інформацію для подальшого аналізу. Обрана модель дає можливість реалізувати як контрастну так і триплетну функції втрат. Загальна кількість параметрів складає: 398666.

Схему такої мережі візуалізовано на рис. 2.3.

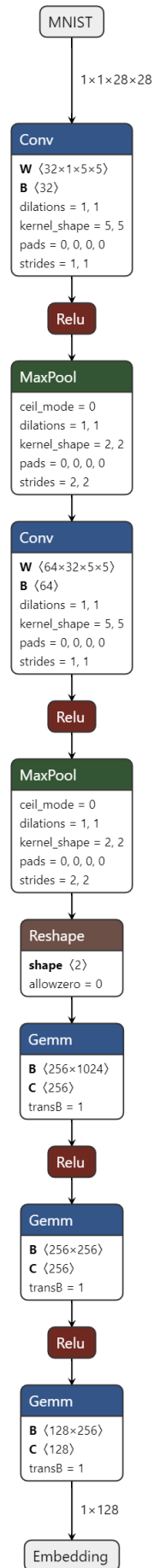


Рисунок 2.3 – Схема згорткової мережі для ембедингів

2.4 Висновок

У даному розділі було розглянуто два популярних фреймворки для побудови нейронних мереж: TensorFlow та Pytorch. Для дослідження сіамських НМ було фреймворк PyTorch. PyTorch є одним з найпопулярніших і найбільш гнучких фреймворків для роботи з глибоким навчанням, завдяки своїй зручності, можливості динамічного визначення графу обчислень і потужній підтримці GPU.

Як вибірку для проведення експериментів було обрано MNIST – одну з найбільш відомих і широко використовуваних наборів даних для задач класифікації зображень. MNIST складається з 60 000 зображень для навчання та 10 000 для тестування, що є ідеальним вибором для початкових експериментів і тестування моделей.

Для вирішення задачі пошуку ембеддингів з подальшою контрастною чи триплетною функцією втрат було обрано модель згорткової нейронної мережі. А також обрано модель мережі для класифікації та з сигмоїдальною функцією активації. Згорткові мережі є стандартом для задач комп'ютерного зору і ефективно справляються з задачами класифікації та пошуку подібності, оскільки вони можуть автоматично виділяти важливі ознаки зображень. Архітектура вибраної мережі включає згорткові шари для екстракції ознак з зображень, а також повнозв'язні шари для генерації ембеддингів, які використовуються для порівняння подібності між зображеннями.

3 ОПИС ПРОГРАМИ

3.1 Формування вибірок

Для формування вибірок було створено 4 класи, ієрархію яких можна побачити на рис. 3.1.

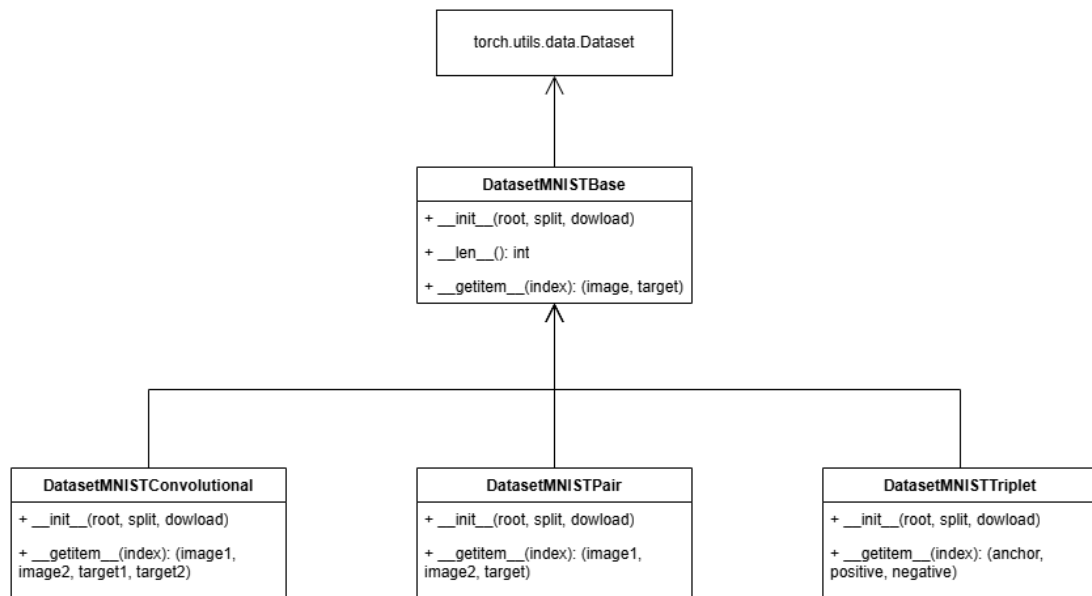


Рисунок 3.1 – Діаграма класів для формування вибірок

Код для обробки зразків даних може стати брудним і його важко підтримувати. В ідеалі, щоб код набору даних був відокремлений від навчального коду моделі для кращої читабельності та модульності. PyTorch надає два примітиви даних: `torch.utils.data.DataLoader` і `torch.utils.data.Dataset` які дозволяють використовувати попередньо завантажені набори даних, а також власні дані. `Dataset` зберігає зразки та їхні відповідні мітки та `DataLoader` обгортає iterable навколо `Dataset` щоб забезпечити легкий доступ до зразків [8].

Бібліотеки домену PyTorch надають ряд попередньо завантажених наборів даних (наприклад, `FashionMNIST`), які створюють підкласи `torch.utils.data.Dataset` та реалізують функції, специфічні для конкретних

даних. Їх можна використовувати для створення прототипу та тестування вашої моделі [8].

Від класу `Dataset` унаслідкується базовий клас `DatasetMNISTBase`. Цей клас приймає три параметри:

- `root`. Приймає рядок до директорії в яку буде завантажено вибірку MNIST;
- `split`. Приймає рядок «train» або «test», який відповідно завантажує вибірку для тренування або тестування;
- `download`. Приймає булеве значення, яке вказує чи слід завантажувати вибірку з мережі.

В базовому класі також реалізовано метод `__len__`, який повертає розмір відповідної вибірки. Для навчальної вибірки повертає 3000. Для тестової 10000. А також метод `__getitem__(index)`, який повертає зображення з вибірки та мітку класу.

Від класу `DatasetMNISTBase` унаслідкується три класи:

- `DatasetMNISTConvolutional`. Клас, який випадковим чином бере бере два зображення та повертає їх мітки класів;
- `DatasetMNISTPair`. Клас, який випадковим чином бере два зображення та повертає мітку схожості;
- `DatasetMNISTTriplet`. Клас, який повертає випадковим три зображення: випадкове, випадкове з того класу що і перше, випадкове з іншого класу.

3.2 Формування функцій втрат

Функції втрат які були описані у першій главі було створено у вигляді класів, які унаслідуються від `torch.nn.Module`. Це дозволяє функціям втрат безпосередньо використовувати всі можливості фреймворку PyTorch, такі як підтримка автоматичного обчислення градієнтів. Це спрощує процес

оптимізації, оскільки PyTorch автоматично обчислює похідні для параметрів моделі. Функції втрат показані на діаграмі класів (рис. 3.2).

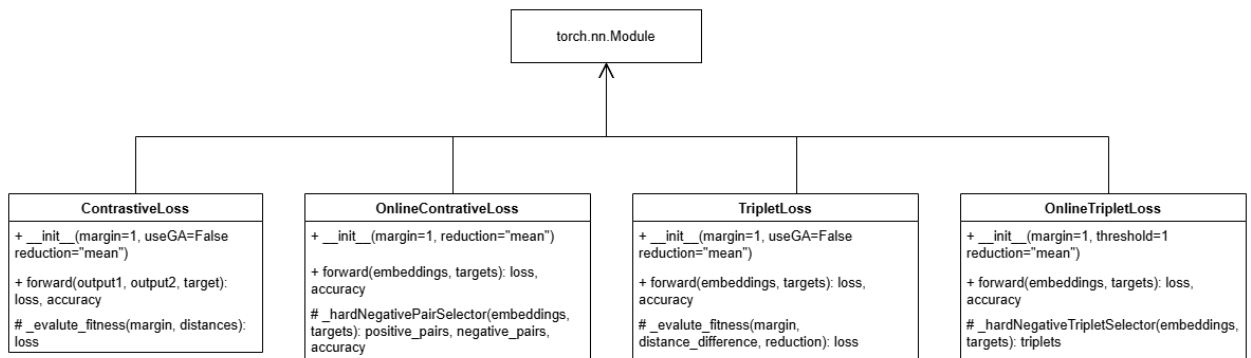


Рисунок 3.2 – Діаграма класів функцій втрат

Клас `ContrastiveLoss` приймає у конструкторі такі параметри:

- `margin=1`. Задає максимальну дозволена відстань між позитивним і негативним ембеддингом;
- `useGA=False`. Визначає чи використовувати генетичний алгоритм для оптимізації відступу у кожному пакеті;
- `reduction="mean"`. Визначає, як агрегуються значення втрат для мінібатчу, щоб отримати кінцеве значення втрат. Може рахувати середнє значення помилки або суму.

Функція `forward` реалізує вже сам алгоритм контрастної функції втрат, формула 1.6. Її псевдокод приведено у ліст. 3.1.

Лістинг 3.1 «Псевдокод роботи контрастної функції втрат»

```

Function forward(output1, output2, target):
    1. Compute the squared distances between output2 and output1:
        distances = sum((output2 - output1)^2 along dimension 1)

    2. Compute the loss for each pair:
        a. For positive pairs (target == 1):
            loss = 0.5 * distances
        b. For negative pairs (target == 0):
            loss = 0.5 * (ReLU(margin - sqrt(distances + epsilon))^2)

    3. Combine the losses:
        losses = 0.5 * (target * distances + (1 - target) *
negative_loss)
  
```

4. Aggregate the losses based on the reduction method:
 - a. If reduction is "mean":


```
return mean(losses)
```
 - b. Else if reduction is "sum":


```
return sum(losses)
```

Клас `OnlineContrastiveLoss` приймає у конструкторі такі параметри:

– `margin = 1`. Задає мінімальну різницю між "позитивною відстанню" (між анкором і позитивним прикладом) та "негативною відстанню" (між анкором і негативним прикладом), яка вважається прийнятною;

– `reduction="mean"`. Визначає, як агрегуються значення втрат для мінібатчу, щоб отримати кінцеве значення втрат. Може рахувати середнє значення помилки або суму.

Функція `forward` реалізує вже сам алгоритм онлайн контрастної функції втрат, формула 1.6. Її псевдокод приведено у ліст. 3.2 .

Лістинг 3.2 «Псевдокод роботи онлайн контрастної функції втрат»

```
Function forward(embeddings, targets):
  1. Select pairs of positive and negative examples:
    Call _hardNegativePairSelector(embeddings, targets) to obtain:
    positive_pairs: indices of pairs from the same class
    negative_pairs: indices of pairs from different classes
    accuracy: computed metric for evaluation (e.g., correct
classifications)

  2. Compute positive loss:
    For each positive pair (i, j) in positive_pairs:
      Compute the squared Euclidean distance between embeddings[i]
and embeddings[j]
      Store these values in positive_loss

  3. Compute negative loss:
    For each negative pair (i, j) in negative_pairs:
      Compute the Euclidean distance between embeddings[i] and
embeddings[j]
      Subtract this distance from the margin
      Apply the ReLU function to ensure non-negative values
      Square the result and store in negative_loss

  4. Combine the losses:
    Concatenate positive_loss and negative_loss into a single
tensor

  5. Aggregate the combined loss:
    If reduction == "mean":
      Compute the average of all losses
    Else (reduction == "sum"):
```

Compute the sum of all losses

6. Return:
The final loss value
The computed accuracy metric

За замовчуванням ця функція викликає іншу `_hardNegativePairSelector`, яка обирає триплети згідно стратегії Batch Hard та приведена у формулі 1.14. У лістингу 3.3 приведено її псевдокод.

Лістинг 3.3 «Псевдокод роботи онлайн контрастної функції втрат із стратегією Batch Hard»

```
Function _hardNegativePairSelector(embeddings, targets):
    1. Compute pairwise distances:
        Use `torch.cdist` to calculate the Euclidean distance between
        all pairs of embeddings.
        Store the distances in a matrix `distance_matrix`.

    2. Generate all possible pairs:
        Create `all_pairs`, which contains all combinations of indices
        (i, j) for the batch of embeddings.

    3. Select positive and negative pairs:
        `positive_pairs`: Select pairs (i, j) where targets[i] ==
        targets[j] (same class).
        `negative_pairs`: Select pairs (i, j) where targets[i] !=
        targets[j] (different classes).

    4. Compute distances for negative pairs:
        Extract distances for all `negative_pairs` from
        `distance_matrix` and store in `negative_distances`.

    5. Identify hard negatives:
        Use `torch.topk` to find the indices of the
        `len(positive_pairs)` closest negative pairs.
        Store these pairs as `top_negative_pairs`.

    6. Calculate accuracy:
        For all `positive_pairs`, check if their distances in
        `distance_matrix` are below the threshold.
        Compute the ratio of correct predictions to the total number of
        positive pairs and store as `accuracy`.

    7. Return:
        `positive_pairs`: Indices of pairs belonging to the same class.
        `top_negative_pairs`: Indices of the closest negative pairs.
        `accuracy`: Computed metric for evaluation.
```

Клас `TripletLoss` приймає у конструкторі такі параметри:

- `margin=1`. Задає максимальну дозволenu відстань між позитивним і негативним ембеддингом;
- `useGA=False`. Визначає чи використовувати генетичний алгоритм для оптимізації відступу у кожному пакеті;
- `reduction="mean"`. Визначає, як агрегуються значення втрат для мінібатчу, щоб отримати кінцеве значення втрат. Може рахувати середнє значення помилки або суму.

Функція `forward` реалізує вже сам алгоритм триплетної функції втрат, формула 1.10. Її псевдокод приведено у ліст. 3.4.

Лістинг 3.4 «Псевдокод роботи триплетної функції втрат»

```
Function forward(type_fit, anchor, positive, negative):
    Step 1: Calculate the squared distances between pairs
    distance_ap = Sum of squares of differences between anchor and
positive
    distance_an = Sum of squares of differences between anchor and
negative

    Step 2: Calculate the loss values using the triplet margin
formula
    loss_values = ReLU(distance_ap - distance_an + margin)

    Step 3: Aggregate the losses
    If reduction is "mean":
        loss = Mean of loss_values
    Else:
        loss = Sum of loss_values

    Step 4: Calculate accuracy
    accuracy = Count of cases where distance_ap < distance_an divided
by the total number of examples

    Step 5: Return the calculated loss and accuracy
    Return loss, accuracy
```

Клас `OnlineTripletLoss` приймає у конструкторі такі параметри:

- `margin=1`. визначає мінімальну різницю між «позитивною відстанню» та «негативною відстанню» яка вважається допустимою.
- `reduction="mean"`. Визначає, як агрегуються значення втрат для мінібатчу, щоб отримати кінцеве значення втрат. Може рахувати середнє значення помилки або суму.

Функція `forward` реалізує вже сам алгоритм онлайн триплетної функції втрат, формула 1.11. Її псевдокод приведено у ліст. 3.5.

Лістинг 3.5 «Псевдокод роботи онлайн триплетної функції втрат»

```
Function forward(embeddings, targets):
    1. Select triplets:
        Call _hardNegativeTripletSelector(embeddings, targets) to
obtain:
        triplets: indices of anchor, positive, and negative
embeddings.

    2. Compute the squared Euclidean distances for anchor-positive and
anchor-negative pairs:
        For each triplet (anchor, positive, negative) in `triplets`:
            Compute distance_ap = (embedding(anchor) -
embedding(positive))^2
            Compute distance_an = (embedding(anchor) -
embedding(negative))^2

    3. Compute the loss for each triplet:
        loss = ReLU(distance_ap - distance_an + margin)
        `ReLU` ensures that only triplets where the anchor-negative
pair is closer than the anchor-positive pair contribute to the loss.

    4. Aggregate the loss:
        If `reduction == "mean"`:
            Compute the mean loss across all triplets.
        Otherwise, if `reduction == "sum"`:
            Compute the sum of the losses across all triplets.

    5. Compute the accuracy:
        For each triplet, check if distance_ap < distance_an (i.e.,
anchor-positive is closer than anchor-negative).
        Compute the ratio of correct triplets (where anchor-positive is
closer than anchor-negative) to the total number of triplets.

    6. Return:
        loss: Computed loss for the batch.
        accuracy: The accuracy based on the comparison of distances.
```

За замовчуванням ця функція викликає іншу `_hardNegativeTripletSelector`, яка обирає триплети згідно стратегії Batch Hard та приведена у формулі 1.14. У ліст. 3.6 приведено її псевдокод.

Лістинг 3.6 «Псевдокод роботи онлайн триплетної функції втрат із стратегії Batch Hard»

```
Function _hardNegativeTripletSelector(embeddings, targets):
```

```

1. Compute pairwise distances between all embeddings:
   Use `torch.cdist` to calculate the Euclidean distance between
   embeddings in the batch and store it in `distance_matrix`.

2. Initialize an empty list for triplets:
   `triplets = []`

3. Loop over each unique class in `targets`:
   For each unique class label in `targets`:
       Create a mask `label_mask` to find all indices where the
       target is equal to the current class label.
       Get the indices of all samples belonging to this class:
       `label_indices = torch.where(label_mask)[0]`.

       If there are fewer than two samples of this class, skip to
       the next class.

       Find indices of negative samples (samples from different
       classes): `negative_indices = torch.where(~label_mask)[0]`.

       Generate all possible anchor-positive pairs using
       `torch.combinations` from `label_indices`.

4. For each anchor-positive pair:
   For each pair `(anchor_positive)` from the generated anchor-
   positive combinations:
       Compute the Euclidean distance between the anchor and the
       positive sample: `ap_distance`.
       Compute the distances from the anchor to all negative
       samples: `negative_distances`.

5. Calculate the loss values:
   For each negative sample, calculate the loss as `loss_values =
   ap_distance - negative_distances + margin`.
   Find the index of the hardest negative sample using
   `torch.argmax(loss_values)`.

6. Check if the loss for the hardest negative is positive:
   If `loss_values[hard_negative_idx] > 0`:
       Add the corresponding anchor-positive-negative triplet to the
       `triplets` list:
       `triplets.append([anchor_positive[0], anchor_positive[1],
       hard_negative])`.

7. If no triplets are found (empty list), and there are still at
   least two label indices and one negative index, add a default triplet:
   `triplets.append([label_indices[0], label_indices[1],
   negative_indices[0]])`.

8. Return the triplets as a tensor:
   Convert the `triplets` list to a tensor and return it: `return
   torch.tensor(triplets, dtype=torch.long)`.

```

3.3 Формування нейронних мереж

Згідно описаних у розділі 2.3 структур нейронних мереж, було розроблено відповідні класи, діаграма яких приведена на рис. 3.3.

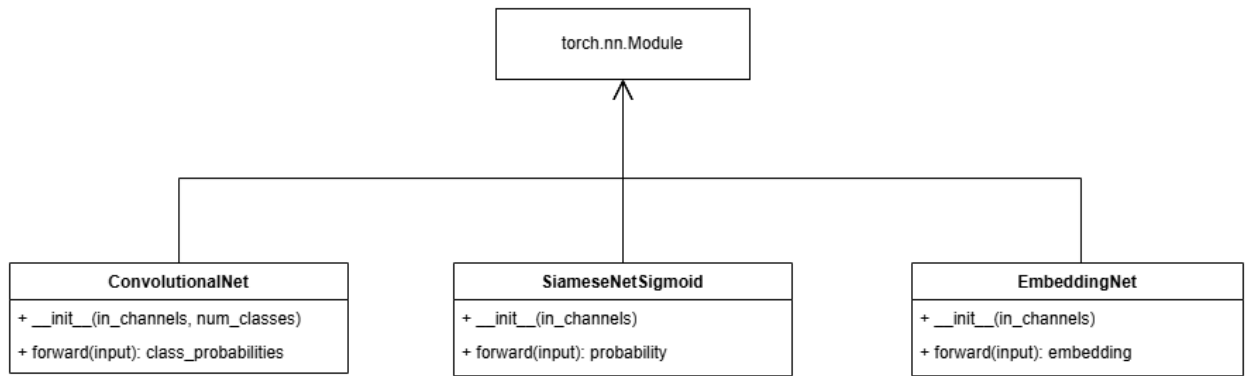


Рисунок 3.3 – Діаграма класів нейронних мереж

Оскільки схеми мереж відповідають тим, що описані у розділі 3.2 не бачу сенсу повторюватися та заново відображати їх структури. Зроблю коротке пояснення.

ConvolutionalNet. Ця модель є типовою згортковою нейронною мережею (CNN), розробленою для задач класифікації зображень. Вона прогнозує мітки класів для вхідних зображень.

SiameseNetSigmoid. Це Сіамська мережа, розроблена для порівняння двох вхідних зображень і визначення, чи схожі вони. Вона зазвичай використовується для задач, таких як навчання з одним прикладом або перевірка.

EmbeddingNet. Ця модель розроблена для генерації ембеддингів з вхідних зображень. Вона зазвичай використовується в задачах, таких як розпізнавання облич або задачах, пов'язаних з вимірюванням схожості.

Як і функції втрат, ці мережі унаслідуються від `torch.nn.Module` для роботи у середовищі `torch`.

3.4 Етап навчання на тестування

Процес навчання та тестування мереж був зведений до єдиної функції для уніфікації роботи з мережами. Псевдокод цієї функції приведено у ліст. 3.7.

Лістинг 3.7 «Псевдокод навчання та тестування нейронних мереж»

```

function fit(train_dataset, test_dataset, model, train_type_model,
test_type_model, train_loss_fn, test_loss_fn, optimizer, scheduler, epochs,
seed, train_batch_size, test_batch_size, model_name,
balanced_sampler=False, train_accuracy_fn=None, test_accuracy_fn=None):
    set seed

    if CUDA is available:
        set device to CUDA
    else:
        set device to CPU

    set train_kwargs with batch_size=train_batch_size
    set test_kwargs with batch_size=test_batch_size
    if CUDA is available:
        update train_kwargs and test_kwargs with CUDA configurations

    if balanced_sampler is True:
        update train_kwargs with balanced sampler for training dataset

    create data loaders for train_loader and test_loader using
train_kwargs and test_kwargs

    initialize metrics dictionary for tracking losses and accuracies

    move model to the appropriate device (CUDA/CPU)

    for each epoch in range(1, epochs + 1):
        train_loss, train_accuracy = call train() with the appropriate
arguments
        test_loss, test_accuracy = call test() with the appropriate
arguments

        store the train and test losses and accuracies in the metrics
dictionary

        update the learning rate scheduler with scheduler.step()

    save model weights to disk (model_name.model)
    save metrics dictionary to disk (model_name.metrics)
    call saveGraphs() with model_name, epochs, and metrics

```

Функція `fit` отримує багато параметрів які налаштовують роботу переданої нейронної мережі. Спочатку в ній ініціалізуються налаштування середовища запуску, створюються `dataloaders`, які формують пакети образів, ініціалізуються метрики роботи мереж. Потім для кожної епохи викликається функція для навчання та тестування. Та результати роботи мережі зберігаються.

Оскільки структури нейронних мереж різні, то була створена певна логіка роботи з конкретною мережею. У ліст. 3.8 приведено псевдокод цієї логіки.

Лістинг 3.8 «Псевдокод логіки роботи з конкретною нейронною мережею»

```
function networks_model(model, type_model, data, loss_fn,
accuracy_fn=None, stage=None):
    match type_model:
        case TypeModel.Conv:
            extract input1, input2, target1, target2 from data
            get output1 and output2 from model
            calculate loss using loss_fn and output1, target1
            calculate accuracy using accuracy_fn and output1, output2,
target1, target2

        case TypeModel.Pairs:
            extract input1, input2, target from data
            get output1 and output2 from model
            calculate loss using loss_fn and output1, output2, target
            calculate accuracy using accuracy_fn and output1, output2,
target

        case TypeModel.Triplet:
            extract input1, input2, input3 from data
            get output1, output2, output3 from model
            calculate loss using loss_fn and output1, output2, output3
            calculate accuracy using accuracy_fn and output1, output2,
output3

        case TypeModel.Sigma:
            extract input1, input2, target from data
            get output from model
            calculate loss using loss_fn and output, target
            calculate accuracy using accuracy_fn and output, target

        case TypeModel.Online:
            extract input, target from data
            get output from model
            calculate loss and accuracy using loss_fn and output,
target

    return loss, accuracy
```

Основна відмінність між роботою різних мереж – це кількість параметрів які повертають самі мережі та функції втрат.

У ліст. 3.9 приведено псевдокод для навчання нейронної мережі.

Лістинг 3.9 «Псевдокод навчання нейронної мережі»

```

function train(model, type_model, device, train_loader, optimizer,
loss_fn, epoch, accuracy_fn=None):
    set model to training mode
    initialize train_loss and train_accuracy to 0

    for each batch in train_loader:
        move data to device (CUDA/CPU)
        reset gradients (optimizer.zero_grad())
        get loss and accuracy by calling loss_model() with data
        accumulate train_loss and train_accuracy
        compute gradients (loss.backward())
        update model parameters (optimizer.step())
        print current epoch, loss, and accuracy

    return average train_loss, average train_accuracy

```

Функція `train` переводить модель в режим навчання та проходиться по пакетах з вибірки, викликає функції втрат, вираховує градієнти, підраховую метрики.

У ліст. 3.10 приведено псевдокод для тестування нейронної мережі.

Лістинг 3.10 «Псевдокод тестування нейронної мережі»

```

function test(model, type_model, device, test_loader, loss_fn,
accuracy_fn=None):
    set model to evaluation mode
    initialize test_loss and test_accuracy to 0

    for each batch in test_loader:
        move data to device (CUDA/CPU)
        get loss and accuracy by calling loss_model() with data
        accumulate test_loss and test_accuracy

    calculate final average test_loss and test_accuracy
    print test results

    return test_loss, test_accuracy

```

Функція `test` переводить модель в оціночний режим та проходиться по пакетах з вибірки, викликає функції втрат для підрахунку метрик.

3.4 Генетичний алгоритм для оптимізації відступів

У розділі 1.6 було запропоновано використання генетичного алгоритму для оптимізації відступу у контрастних та триплетних функціях втрат. Для його реалізації використався популярний фреймворк DEAP. DEAP – це нова еволюційна обчислювальна структура для швидкого створення прототипів і тестування ідей. Він прагне зробити алгоритми явними, а структури даних прозорими. Він працює в ідеальній гармонії з механізмом розпаралелювання, таким як багатопроцесорність і SCOOP [11].

Псевдокод генетичного алгоритму приведено у ліст. 3.11.

Лістинг 3.11 «Псевдокод генетичного алгоритму»

```
Function InitializeGA(opt, fitness_function, kwargs):
    Define range for attributes (low = 0.1, up = 1)
    Create Fitness Class ("FitnessMax") to maximize the fitness value
    Create Individual Class ("Individ") that includes fitness
information
    Initialize toolbox:
        Register "attribute" as a random value in range [low, up]
        Register "individualCreator" to create an individual with 1
attribute
        Register "populationCreator" to create a population of
individuals
        Register "evaluate" as the fitness_function
        Register "mate" for crossover using Simulated Binary Bounded
(SBB) operator
        Register "mutate" for mutation using Polynomial Bounded (PB)
operator
        Register "select" for selection using Tournament selection
with size = 3

Function RunGA(iterations):
    Initialize Hall of Fame (hof) to store the best individual
    Create initial population of size 20 using "populationCreator"
    For each generation in range(iterations):
        Evaluate fitness for all individuals in the population
        Perform selection to choose parents for reproduction
        Apply crossover (cxpb = 0.5 probability) to generate offspring
        Apply mutation (mutpb = 0.1 probability) to offspring
        Replace least fit individuals with new offspring
        Add the best individual to Hall of Fame (hof)
    Return the best individual from Hall of Fame (hof[0][0])

main:
    Define optimization direction (opt) as maximize or minimize
    Pass fitness_function to evaluate the performance of individuals
```

```
Initialize GA with parameters
optimal_margin = RunGA(iterations=50)
```

3.5 Висновок

У цьому розділі були розглянуті основні етапи побудови та навчання моделей на основі нейронних мереж, зокрема, створення вибірок, функцій втрат, архітектур мереж і процесу тренування.

Важливим кроком є підготовка та правильне формування виборки для тренування і тестування. Для цього використовуються вибірка MNIST, а також розробляються додаткові з налаштуванням для певних типів вхідних даних.

Функція втрат є критично важливою для правильного налаштування моделі. Вона визначає, як модель повинна коригувати свої параметри, щоб мінімізувати помилку. Різні типи функцій втрат, такі як контрастні втрати, триплетні втрати та інші, використовуються в залежності від типу задачі та архітектури моделі.

Для різних завдань використовуються різні архітектури нейронних мереж. Згідно обраних архітектур з розділу 2.3 створено згорткові мережі, а також сіамські мережі, які використовуються для порівняння пар або триплетів.

Процес навчання включає визначення функції втрат, оптимізатора та методів для оцінки результатів. Під час тренування мережа отримує дані з виборки, розраховує втрати, коригує свої параметри, використовуючи зворотний зв'язок (градієнтний спуск). Оцінка ефективності навчання проводиться через точність та значення втрат на тестовій вибірці.

4 ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМ

Слід зазначити що в ході тестування алгоритмів використовувався планувальник швидкості оптимізатору `torch.optim.lr_scheduler.LRScheduler`, який надає кілька методів налаштування швидкості навчання на основі кількості епох. Більшість планувальників швидкості навчання можна назвати ланцюжковими планувальниками. Результатом є те, що кожен планувальник застосовується один за одним та впливає швидкість навчання [10].

4.1 Тестування згорткової мережі

Параметри запуску згорткової нейронної мережі для класифікації вибірки MNIST приведено у табл. 4.1. Схему мережі приведено на рис. 2.1.

Таблиця 4.1 – Параметри запуску згорткової нейронної мережі для класифікації

Параметр	Значення
Кількість епох	20
Модель	ConvolutionalNet(1, 10)
Кількість ваг	384842
Вибірка	DatasetMNISTConvolutional
Кількість образів для навчання	60000
Кількість образів для тестування	20000
Балансування вибірки	Hi
Розмір пакету для навчання	64
Розмір пакету для тестування	64
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Функція втрат	CrossEntropyLoss()

Результати роботи мережі приведено на рис. 4.1-4.2.

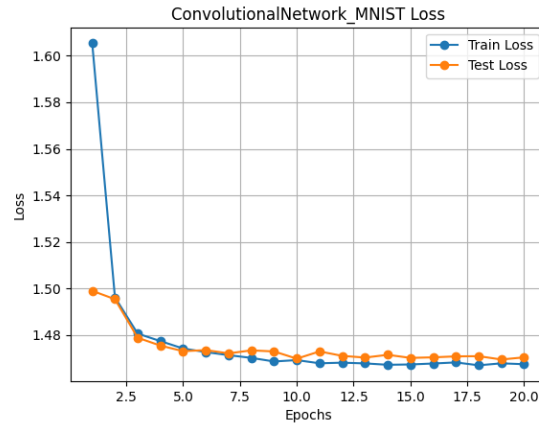


Рисунок 4.1 – Втрати під час навчання згорткової НМ

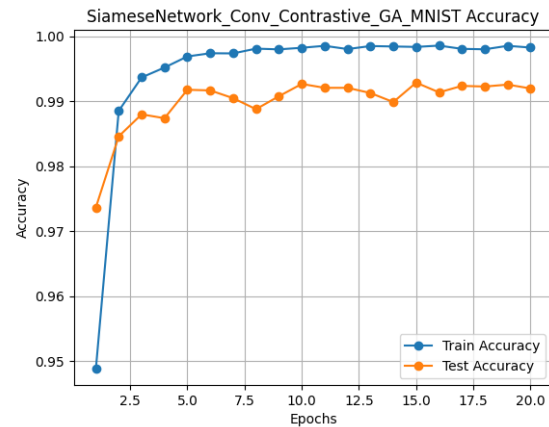


Рисунок 4.2 – Точність класифікації під час навчання згорткової НМ

Значення на останній епосі:

- Навчання. Втрати – 1.4675. Точність – 0.9889;
- Тестування. Втрати – 1.4704. Точність – 0.9826.

Модель показала досить велику точність класифікації ще на 10 епосі, а потім знаходилася на одному рівні. Це дає базові значення метрик для порівняння із сіамськими мережами.

4.2 Тестування сіамської нейронної мережі із сигмоїдальною функцією активації

Параметри запуску сіамської нейронної мережі із сигмоїдальною функцією активації для класифікації вибірки MNIST приведено у табл. 4.2. Схему мережі приведено на рис. 2.2.

Таблиця 4.2 – Параметри запуску сіамської нейронної мережі із сигмоїдальною функцією активації

Параметр	Значення
Кількість епох	20
Модель	SiameseNetSigmoid(1)
Кількість ваг	644672
Вибірка	DatasetMNISTPair
Кількість образів для навчання	60000
Кількість образів для тестування	20000
Балансування вибірки	Hi
Розмір пакету для навчання	64
Розмір пакету для тестування	64
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Функція втрат	BCELoss()

Результати роботи мережі приведено на рис. 4.3-4.4

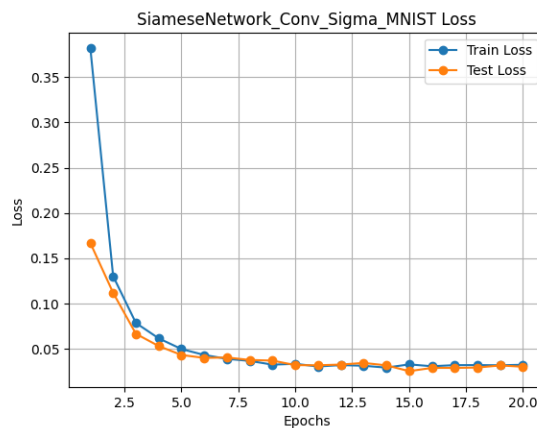


Рисунок 4.3 – Втрати під час навчання сіамської НМ із сигмоїдальною функцією активації

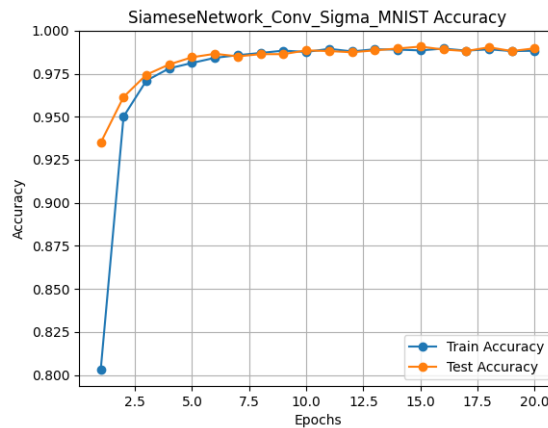


Рисунок 4.4 – Точність класифікації під час навчання сіамської НМ із сигмоїдальною функцією активації

Значення на останній епосі:

– Навчання. Втрати – 0.0326. Точність – 0.9883;

– Тестування. Втрати – 0.0305. Точність – 0.9897.

Модель показала майже таку ж точність, що і згорткова мережа, проте сходимість під час навчання значно краще.

4.3 Тестування сіамських нейронних мереж з контрастними функціями втрат

Параметри запуску сіамської нейронної мережі з контрастною функцією втрат для класифікації вибірки MNIST приведено у табл. 4.3. Схему мережі приведено на рис. 2.3.

Таблиця 4.3 – Параметри запуску сіамської нейронної мережі з контрастною функцією втрат

Параметр	Значення
Кількість епох	20
Модель	EmbeddingNet(1)
Кількість ваг	398666
Вибірка	DatasetMNISTPair
Кількість образів для навчання	60000

Продовження таблиці 4.3

Параметр	Значення
Кількість образів для тестування	20000
Балансування вибірки	Ні
Розмір пакету для навчання	64
Розмір пакету для тестування	64
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Функція втрат	ContrastiveLoss(reduction="sum")

Результати роботи мережі приведено на рис. 4.5-4.6.

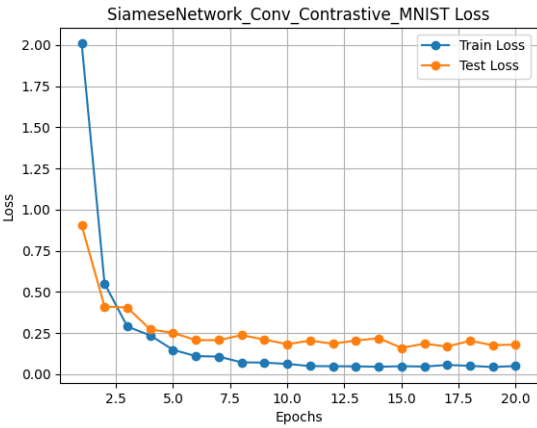


Рисунок 4.5 – Втрати під час навчання сіамської НМ з контрастною функцією втрат

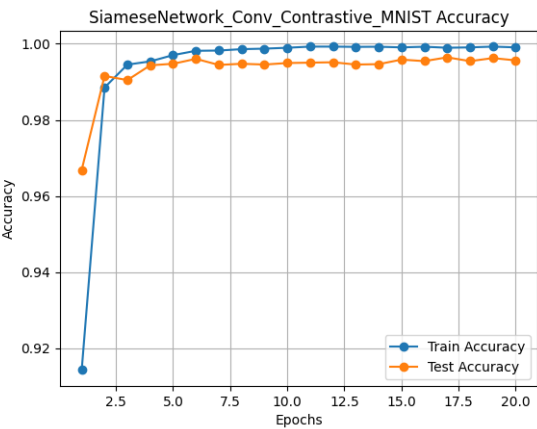


Рисунок 4.6 – Точність класифікації під час навчання сіамської НМ з контрастною функцією втрат

Значення на останній епосі:

- Навчання. Втрати – 0.0499. Точність – 0.9962;
- Тестування. Втрати – 0.1800. Точність – 0.9956.

Параметри запуску сіамської нейронної мережі з контрастною функцією втрат та адаптивним відступом для класифікації вибірки MNIST приведено у табл. 4.4 та. Схему мережі приведено на рис. 2.4.

Таблиця 4.4 – Параметри запуску сіамської нейронної мережі з контрастною функцією втрат та адаптивним відступом

Параметр	Значення
Кількість епох	20
Модель	EmbeddingNet(1)
Кількість ваг	398666
Вибірка	DatasetMNISTPair
Кількість образів для навчання	60000
Кількість образів для тестування	20000
Балансування вибірки	Hi
Розмір пакету для навчання	64
Розмір пакету для тестування	64
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Функція втрат	ContrastiveLoss(useGA=True, reduction="sum")

Результати роботи моделі приведено на рис. 4.7-4.8.

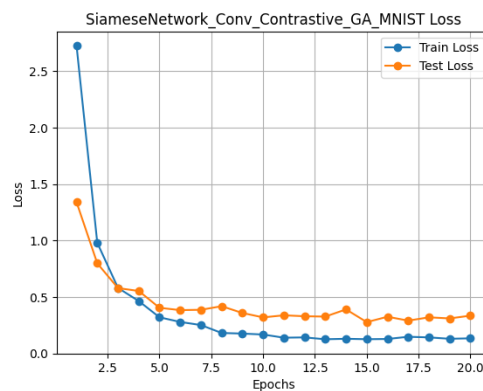


Рисунок 4.7 – Втрати під час навчання сіамської НМ з контрастною функцією втрат та адаптивним відступом

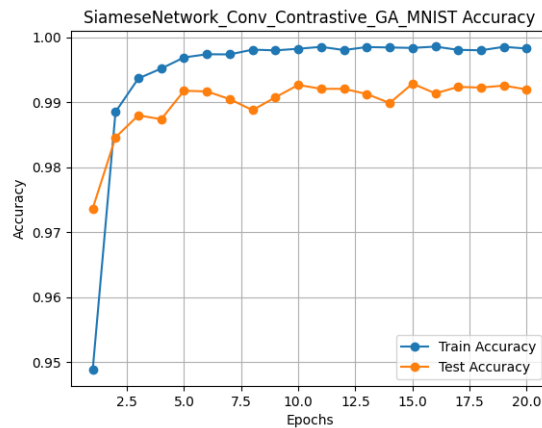


Рисунок 4.8 – Точність класифікації під час навчання сіамської НМ з контрастною функцією втрат та адаптивним відступом

Значення на останній епосі:

- Навчання. Втрати – 0.1349. Точність – 0.9982;
- Тестування. Втрати – 0.3356. Точність – 0.9919.

Параметри запуску сіамської нейронної мережі з онлайн контрастною функцією втрат та стратегією Batch Hard для класифікації вибірки MNIST приведено у табл. 4.5. Схему мережі приведено на рис. 2.3. Для запуску цієї мережі також використовувалася балансування вибірок для рівномірного розподілу класів по пакетах.

Таблиця 4.5 – Параметри запуску сіамської нейронної мережі з онлайн контрастною функцією втрат та стратегією Batch Hard

Параметр	Значення
Кількість епох	20
Модель	EmbeddingNet(1)
Кількість ваг	398666
Вибірка для навчання	DatasetMNISTBase
Кількість образів для навчання	60000
Балансування вибірки для навчання	Так
Розмір пакету для навчання	64
Функція втрат для навчання	OnlineContrastiveLoss(threshold=0.7, reduction="sum")
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)

Продовження таблиці 4.5

Параметр	Значення
Вибірка для тестування	DatasetMNISTPair
Кількість образів для тестування	20000
Балансування вибірки для тестування	Ні
Розмір пакету для тестування	64
Функція втрат для тестування	ContrastiveLoss(reduction=“sum”)

Результати роботи мережі приведено на рис. 4.9-4.10.

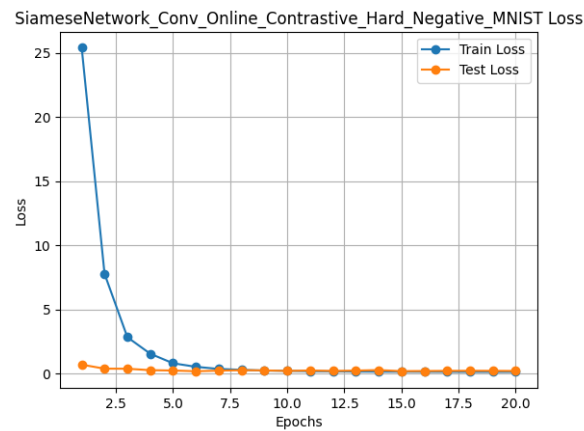


Рисунок 4.9 – Втрати під час навчання сіамської НМ з онлайн контрастною функцією втрат та стратегією Batch Hard

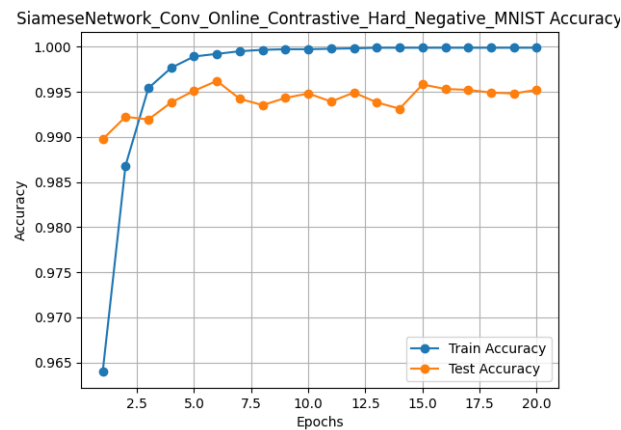


Рисунок 4.10 – Точність класифікації під час навчання сіамської НМ з онлайн контрастною функцією втрат та стратегією Batch Hard

Значення на останній епосі:

- Навчання. Втрати – 0.1240. Точність – 0.9999;
- Тестування. Втрати – 0.1941. Точність – 0.9952.

Порівняння точностей розглянутих моделей з контрастною функцією втрат на тестових вибірках показано на рис. 4.11.

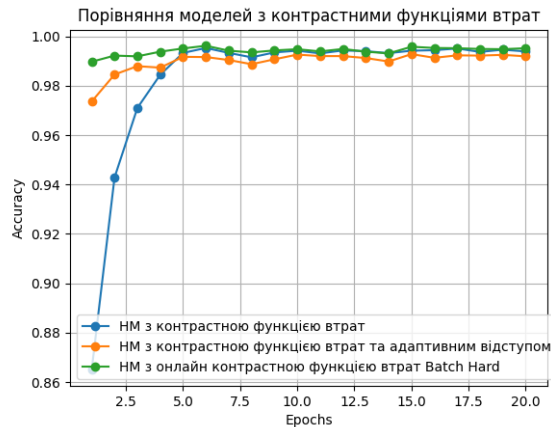


Рисунок 4.11 – Точність класифікації під час тестування сіамських НМ з контрастними функціями втрат

Модель з адаптивним відступом за допомогою генетичного алгоритму на ранніх епохах відпрацювала краще, ніж із статичним відступом, найкраще відпрацювала онлайн функція втрат.

4.4 Тестування сіамських нейронних мереж з триплетними функціями втрат

Параметри запуску сіамської нейронної мережі з триплетною функцією втрат для класифікації вибірки MNIST приведено у табл. 4.6. Схему мережі приведено на рис. 2.3.

Таблиця 4.6 – Параметри запуску сіамської нейронної мережі з триплетною функцією втрат

Параметр	Значення
Кількість епох	20
Модель	EmbeddingNet(1)
Кількість ваг	398666
Вибірка	DatasetMNISTTriplet
Кількість образів для навчання	90000
Кількість образів для тестування	20000
Балансування вибірки	Hi
Розмір пакету для навчання	64
Розмір пакету для тестування	64
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Функція втрат	torch.nn.TripletMarginWithDistanceLoss(reduction="sum")

Результати роботи мережі приведено на рис. 4.12-4.13.

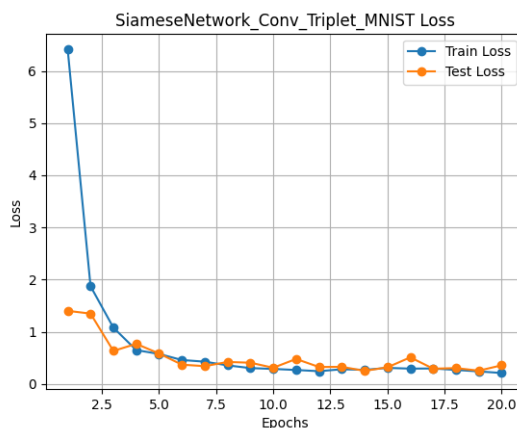


Рисунок 4.12 – Втрати під час навчання сіамської НМ з триплетною функцією втрат

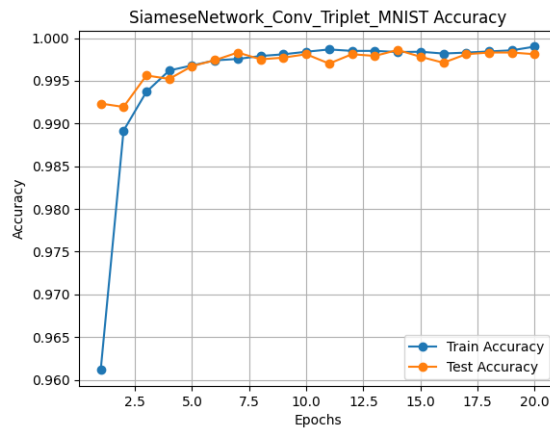


Рисунок 4.13 – Точність класифікації під час навчання сіамської НМ з триплетною функцією втрат

Значення на останній епосі:

- Навчання. Втрати – 0.2082. Точність – 0.9990;
- Тестування. Втрати – 0.3559. Точність – 0.9981.

Модель показала таку саму точність та сходимість як і онлайн контрастна функція втрат із стратегією Batch Hard.

Параметри запуску сіамської нейронної мережі з триплетною функцією втрат та адаптивним відступом для класифікації вибірки MNIST приведено у табл. 4.7. Схему мережі приведено на рис. 2.3.

Таблиця 4.7 – Параметри запуску сіамської нейронної мережі з триплетною функцією втрат та адаптивним відступом

Параметр	Значення
Кількість епох	20
Модель	EmbeddingNet(1)
Кількість ваг	398666
Вибірка	DatasetMNISTTriplet
Кількість образів для навчання	90000
Кількість образів для тестування	20000
Балансування вибірки	Ні
Розмір пакету для навчання	64

Продовження таблиці 4.7

Параметр	Значення
Розмір пакету для тестування	64
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Функція втрат	torch.nn.TripletMarginWithDistanceLoss(useGA=True, reduction="sum")

Результати роботи приведено на рис. 4.14-4.15.

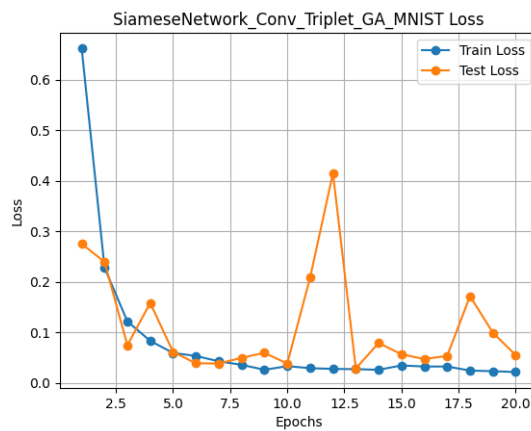


Рисунок 4.14 – Втрати під час навчання сіамської НМ з триплетною функцією втрат та адаптивним відступом

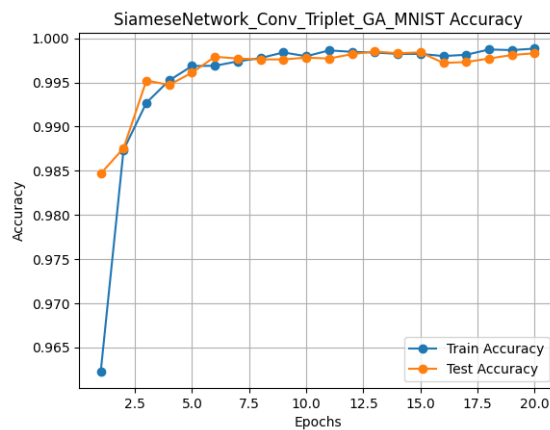


Рисунок 4.15 – Точність класифікації під час навчання сіамської НМ з триплетною функцією втрат та адаптивним відступом

Значення на останній епосі:

- Навчання. Втрати – 0.0220. Точність – 0.9988;
- Тестування. Втрати – 0.0554. Точність – 0.9983

Параметри запуску сіамської нейронної мережі з онлайн триплетною функцією втрат та стратегією Batch Hard для класифікації вибірки MNIST приведено у табл. 4.7. Схему мережі приведено на рис. 2.3. Для запуску цієї мережі також використовувалася балансування вибірок для рівномірного розподілу класів по пакетах.

Таблиця 4.6 – Параметри запуску сіамської нейронної мережі з онлайн триплетною функцією втрат та стратегією Batch Hard

Параметр	Значення
Кількість епох	20
Модель	EmbeddingNet(1)
Кількість ваг	398666
Вибірка для навчання	DatasetMNISTTriplet
Кількість образів для навчання	90000
Балансування вибірки для навчання	Так
Розмір пакету для навчання	64
Функція втрат для навчання	OnlineTripletLoss(margin=1, reduction="sum")
Оптимізатор	Adam(lr=1e-3)
Планувальник	StepLR(step_size=2, gamma=0.5)
Вибірка для тестування	DatasetMNISTTriplet
Кількість образів для тестування	20000
Балансування вибірки для тестування	Hi
Розмір пакету для тестування	64
Функція втрат для тестування	torch.nn.TripletMarginWithDistanceLoss(reduction="sum")

Результати роботи мережі приведено на рис. 4.16-4.17.

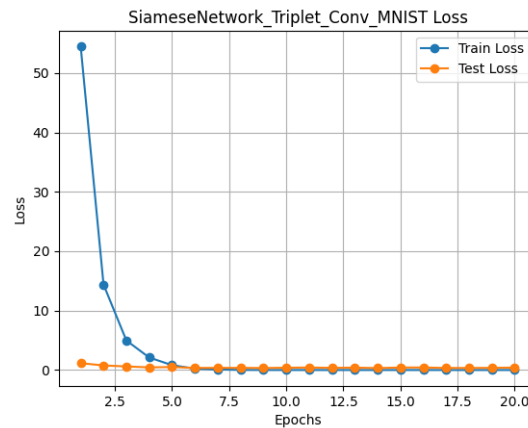


Рисунок 4.16 – Втрати під час навчання сіамської НМ з онлайн триплетною функцією втрат та стратегією Batch Hard

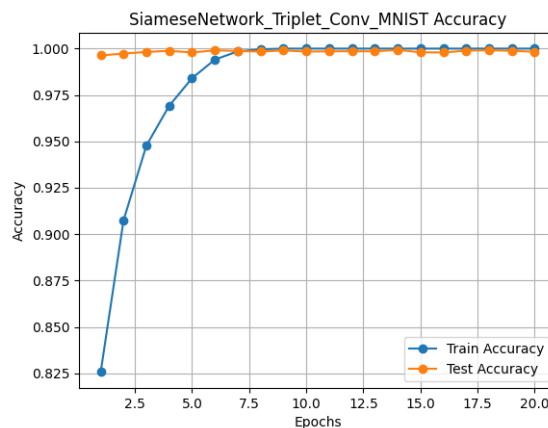


Рисунок 4.17 – Точність класифікації під час навчання сіамської НМ з онлайн триплетною функцією втрат та стратегією Batch Hard

Значення на останній епосі:

- Навчання. Втрати – 0.0000. Точність – 1.0000;
- Тестування. Втрати – 0.4109. Точність – 0.9983.

Порівняння точностей на тестових вибірках показано на рис. 4.18. Можна бачити, що онлайн функція втрат класифікує краще ніж звичайна. А функція втрат з адаптивним відступом на перших епохах працює трохи гірше ніж проста триплетна функція втрат.

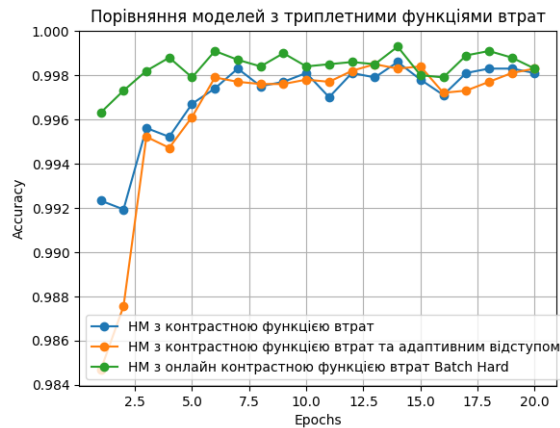


Рисунок 4.18 – Точність класифікації під час тестування сіамських НМ з тріплетними функціями втрат

4.5 Порівняння результатів

На рис. 4.19 приведено порівняння роботи всіх раніше розглянутих мереж за метрикою – точність класифікації навчальної вибірки.

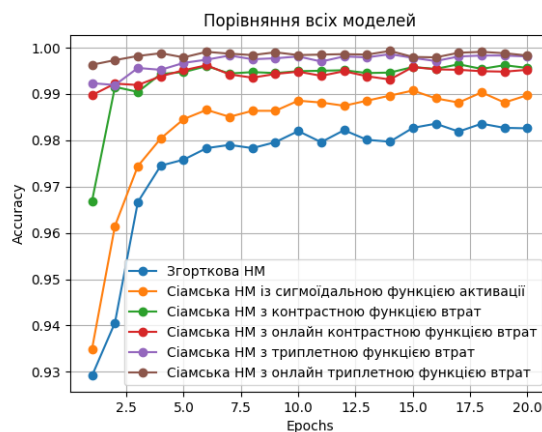


Рисунок 4.19 – Точність класифікації всіх мереж

Як було описано раніше, чим складніше модель та стратегія вибору образів, тим краще класифікація моделі. Безпосередньо найкращу якість показала сіамська НМ з онлайн тріплетною функцією втрат та стратегією Batch All. А також моделі з онлайн стратегіями вибірки образів показали кращі результати ніж випадкові вибірки образів.

ВИСНОВКИ

В процесі виконання кваліфікованої роботи магістра було досліджено та програмно реалізовано розпізнавання об'єктів на зображенні на основі сіамських нейронних мереж. Розглянуто та реалізовано різні функції втрат для сіамських НМ – контрастну та триплетну. Вони забезпечують навчання моделей для задач класифікації та пошуку схожостей між об'єктами.

Наукова новизна роботи полягає у використанні генетичного алгоритму для оптимізації параметра відступу в контрастних та триплетних функціях втрат. Традиційно параметр відступу задається вручну або визначається емпірично, що може призводити до недосягнення оптимального значення і, як наслідок, погіршення якості навчання моделі. Запропонований підхід використовує генетичний алгоритм для динамічного налаштування цього параметра, дозволяючи враховувати специфіку кожного пакета даних під час тренування. Це забезпечує адаптивність моделі, покращує її здатність до узагальнення та підвищує точність роботи, особливо у складних задачах, таких як пошук схожих об'єктів чи класифікація з високою варіативністю даних.

Було розглянуто та використано стратегії Batch Hard, Batch All та Lifted Embedding, які дозволяють ефективно вибирати позитивні та негативні пари чи триплети в рамках одного пакету для оптимального навчання.

PyTorch було використано як основний інструмент для реалізації, завдяки його гнучкості, простоті та широкій підтримці інструментів для роботи з нейронними мережами.

Використано вибірку MNIST, яка є стандартним набором даних для навчання моделей обробки зображень.

Було обрано відповідні архітектури згорткових нейронних мереж для вирішення поставлених завдань. Враховано баланс між складністю моделі та її здатністю до узагальнення.

Створено класи для роботи з різними структурами нейронних мереж, які забезпечують навчання, тестування та збереження результатів у стандартизованому вигляді.

Проведено експериментальні дослідження, що дозволили оцінити ефективність обраних архітектур та стратегій навчання. Чим складніше модель та стратегія вибору образів, тим краще класифікація моделі. Безпосередньо найкращу якість показала сіамська НМ з онлайн триплетною функцією втрат та стратегією Batch All.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Siamese Neural Networks: An overview [Electronic resource]. – Access mode: https://link.springer.com/protocol/10.1007/978-1-0716-0826-5_3.
2. Dimensionality Reduction by Learning an Invariant Mapping [Electronic resource]. – Access mode: <https://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf>.
3. FaceNet: A Unified Embedding for Face Recognition and Clustering [Electronic resource]. – Access mode: <https://arxiv.org/pdf/1503.03832>.
4. In Defense of the Triplet Loss for Person Re-identification [Electronic resource]. – Access mode: <https://arxiv.org/pdf/1703.07737>.
5. Keras vs Tensorflow vs Pytorch: Key Differences Among Deep Learning [Electronic resource]. – Access mode: <https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article>.
6. PyTorch vs Tensorflow vs Keras [Electronic resource]. – Access mode: <https://www.datacamp.com/tutorial/pytorch-vs-tensorflow-vs-keras>.
7. THE MNIST DATABASE [Electronic resource]. – Access mode: <https://yann.lecun.com/exdb/mnist>.
8. Datasets & DataLoaders [Electronic resource]. – Access mode: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.
9. TripletMarginWithDistanceLoss [Electronic resource]. – Access mode: <https://pytorch.org/docs/stable/generated/torch.nn.TripletMarginWithDistanceLoss.html>.
10. How to adjust learning rate [Electronic resource]. – Access mode: <https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate>.
11. DEAP documentation [Electronic resource]. – Access mode: <https://deap.readthedocs.io/en/master/>.

ДОДАТОК А
Текст програми

A.1 Фрагмент файла datasets.py

```

import numpy.random
from torch.utils.data import Dataset
from torchvision.transforms import v2 as T
from torchvision import datasets
import torch, numpy as np

class DatasetMNISTBase(Dataset):
    def __init__(self, root, split, download=False):
        super().__init__()
        self.split = split
        self.transform = T.ToDtype(torch.float, scale=True)
        self.dataset = datasets.MNIST(root, train=(split == "train"),
download=download)
        self.data = self.dataset.data.unsqueeze(1).clone()
        np_arr = np.array(self.dataset.targets.clone())
        self.list_classes = np.unique(np_arr)
        self.grouped_examples = {i: np.where(np_arr == i)[0] for i in
self.list_classes}

    def __len__(self):
        return int(len(self.dataset) / 2) if self.split == "train"
else len(self.dataset)

    def __getitem__(self, index):
        image = self.transform(self.data[index].clone())
        return image, self.dataset[index][1]

class DatasetMNISTPair(DatasetMNISTBase):
    def __init__(self, root, split, download=False):
        super().__init__(root, split, download)

    def __getitem__(self, index):
        class1 = numpy.random.choice(self.list_classes)
        index1 = numpy.random.choice(self.grouped_examples[class1])
        image_1 = self.transform(self.data[index1].clone())
        # same class
        if index % 2 == 0:
            index2 =
numpy.random.choice(self.grouped_examples[class1])
            while index2 == index1:
                index2 =
numpy.random.choice(self.grouped_examples[class1])
            image_2 = self.transform(self.data[index2].clone())
            target = torch.tensor(1, dtype=torch.float)
        # different class
        else:
            class2 = numpy.random.choice(self.list_classes)
            while class2 == class1:
                class2 = numpy.random.choice(self.list_classes)
            index2 =
numpy.random.choice(self.grouped_examples[class2])
            image_2 = self.transform(self.data[index2].clone())
            target = torch.tensor(0, dtype=torch.float)
        return image_1, image_2, target

class DatasetMNISTTriplet(DatasetMNISTBase):
    def __init__(self, root, split, download=False):

```

```

        super().__init__(root, split, download)

    def __getitem__(self, index):
        class1 = numpy.random.choice(self.list_classes)
        index1 = numpy.random.choice(self.grouped_examples[class1])
        anchor = self.transform(self.data[index1].clone())
        # same class
        index2 = numpy.random.choice(self.grouped_examples[class1])
        while index2 == index1:
            index2 =
numpy.random.choice(self.grouped_examples[class1])
        positive = self.transform(self.data[index2].clone())
        # different class
        class2 = numpy.random.choice(self.list_classes)
        while class2 == class1:
            class2 = numpy.random.choice(self.list_classes)
        index3 = numpy.random.choice(self.grouped_examples[class2])
        negative = self.transform(self.data[index3].clone())
        return anchor, positive, negative

class DatasetMNISTConvolutional(DatasetMNISTBase):
    def __init__(self, root, split, download=False):
        super().__init__(root, split, download)

    def __getitem__(self, index):
        class1 = numpy.random.choice(self.list_classes)
        index1 = numpy.random.choice(self.grouped_examples[class1])
        image_1 = self.transform(self.data[index1].clone())
        # same class
        if index % 2 == 0:
            index2 =
numpy.random.choice(self.grouped_examples[class1])
            while index2 == index1:
                index2 =
numpy.random.choice(self.grouped_examples[class1])
            image_2 = self.transform(self.data[index2].clone())
            class2 = class1
        # different class
        else:
            class2 = numpy.random.choice(self.list_classes)
            while class2 == class1:
                class2 = numpy.random.choice(self.list_classes)
            index2 =
numpy.random.choice(self.grouped_examples[class2])
            image_2 = self.transform(self.data[index2].clone())
        return image_1, image_2, class1, class2

```

A.2 Фрагмент файла losses.py

```

import torch, random
import torch.nn as nn
import torch.nn.functional as F
from deap import base, creator, tools, algorithms
import numpy as np

```

```

class GA:
    def __init__(self, opt, fitness_func, **kwargs):
        low = 0.1
        up = 1
        creator.create("FitnessMax", base.Fitness, weights=(opt,))
        creator.create("Individ", list, fitness=creator.FitnessMax)
        self.toolbox = base.Toolbox()
        self.toolbox.register("attribute", random.uniform, low, up)
        self.toolbox.register("individualCreator", tools.initRepeat,
creator.Individ, self.toolbox.attribute, n=1)
        self.toolbox.register("populationCreator", tools.initRepeat,
list, self.toolbox.individualCreator)
        self.toolbox.register("evaluate", fitness_func, **kwargs)
        self.toolbox.register("mate", tools.cxSimulatedBinaryBounded,
low=low, up=up, eta=20)
        self.toolbox.register("mutate", tools.mutPolynomialBounded,
low=low, up=up, eta=20, indpb=1)
        self.toolbox.register("select", tools.selTournament,
tournsize=3)

    def run(self, iter):
        hof = tools.HallOfFame(1)
        population = self.toolbox.populationCreator(n=20)
        algorithms.eaSimple(population, self.toolbox, cxpb=0.5,
mutpb=0.1, ngen=iter, halloffame=hof, verbose=False)
        return hof[0][0]

class ContrastiveLoss(nn.Module):
    def __init__(self, margin=1, threshold=0.7, useGA=False,
reduction="mean"):
        super(ContrastiveLoss, self).__init__()
        self.useGA = useGA
        self.reduction = reduction
        self.eps = 1e-9
        self.thresholds = []
        self.reset_train = False
        self.margin = margin
        self.threshold = threshold

    def forward(self, type_fit, output1, output2, targets):
        distances = torch.norm(output1 - output2, dim=1)
        if type_fit == "train":

```

```

        if self.reset_train:
            self.thresholds = []
            self.reset_train = False
        if self.useGA:
            ga = GA(opt=1.0, fitness_func=self._evaluate_fitness,
distances=distances, targets=targets)
            self.threshold = ga.run(50)
            self.thresholds.append(self.threshold)
        if type_fit == "test":
            if self.useGA and self.reset_train == False:
                self.threshold = np.mean(self.thresholds)
                self.reset_train = True
                print(f"Threshold: {self.threshold:.4f}")
            losses = 0.5 * (targets.float() * distances + (1 -
targets).float() * F.relu(self.margin - (distances +
self.eps).sqrt()).pow(2))
            loss = losses.mean() if self.reduction == "mean" else
losses.sum()

            predict = torch.where(distances < self.threshold, 1, 0)
            accuracy = (predict == targets).sum().item() / len(targets)
            return loss, accuracy

    def _evaluate_fitness(self, margin, distances, targets):
        predict = torch.where(distances < margin[0], 1, 0)
        accuracy = (predict == targets).sum().item() / len(targets)
        return accuracy,

class OnlineContrastiveLoss(nn.Module):
    def __init__(self, margin=1, threshold=0.7, reduction="mean"):
        super(OnlineContrastiveLoss, self).__init__()
        self.margin = margin
        self.reduction = reduction
        self.threshold = threshold

    def forward(self, embeddings, targets):
        positive_pairs, negative_pairs, accuracy =
self._hardNegativePairSelector(embeddings, targets)
        positive_loss = (embeddings[positive_pairs[:, 0]] -
embeddings[positive_pairs[:, 1]]).pow(2).sum(1)
        negative_loss = F.relu(self.margin -
(embeddings[negative_pairs[:, 0]] - embeddings[negative_pairs[:,
1]]).pow(2).sum(1).sqrt()).pow(2)

```

```

        loss = torch.cat([positive_loss, negative_loss], dim=0)
        loss = loss.mean() if self.reduction == "mean" else loss.sum()
        return loss, accuracy

    def _hardNegativePairSelector(self, embeddings, targets):
        distance_matrix = torch.cdist(embeddings, embeddings,
p=2).to(embeddings.device)
        batch_size = targets.size(0)
        all_pairs = torch.combinations(torch.arange(batch_size),
r=2).to(embeddings.device)
        positive_pairs = all_pairs[targets[all_pairs[:, 0]] ==
targets[all_pairs[:, 1]]]
        negative_pairs = all_pairs[targets[all_pairs[:, 0]] !=
targets[all_pairs[:, 1]]]
        negative_distances = distance_matrix[negative_pairs[:, 0],
negative_pairs[:, 1]]
        top_negative_distances, top_indices =
torch.topk(negative_distances, k=len(positive_pairs), largest=False)
        top_negative_pairs = negative_pairs[top_indices]

        positive_predictions = (distance_matrix[positive_pairs[:, 0],
positive_pairs[:, 1]] < self.threshold)
        accuracy = positive_predictions.sum().float() /
len(positive_predictions)
        return positive_pairs, top_negative_pairs, accuracy.to("cpu")

class TripletLoss(nn.Module):
    def __init__(self, margin=1, useGA=False, reduction="mean"):
        super(TripletLoss, self).__init__()
        self.margin = margin
        self.margins = []
        self.reset_train = False
        self.reduction = reduction
        self.useGA = useGA

    def forward(self, type_fit, anchor, positive, negative):
        distance_ap = (anchor - positive).pow(2).sum(1)
        distance_an = (anchor - negative).pow(2).sum(1)
        distance_difference = distance_ap - distance_an

        if type_fit == "train":
            if self.reset_train:

```

```

        self.margins = []
        self.reset_train = False
    if self.useGA:
        ga = GA(opt=-1.0, fitness_func=self._evaluate_fitness,
distance_difference=distance_difference, reduction=self.reduction)
        self.margin = ga.run(50)
        self.margins.append(self.margin)
    if type_fit == "test":
        if self.useGA and self.reset_train == False:
            self.margin = np.mean(self.margin)
            self.reset_train = True
            print(f"Margin: {self.margin:.4f}")

    losses = F.relu(distance_difference + self.margin)
    loss = losses.mean() if self.reduction == "mean" else
losses.sum()
    accuracy = ((distance_ap < distance_an).sum().item() /
len(distance_ap))
    return loss, accuracy

def _evaluate_fitness(self, margin, distance_difference,
reduction):
    losses = F.relu(distance_difference + margin[0])
    loss = losses.mean() if reduction == "mean" else losses.sum()
    return loss.item(),

class OnlineTripletLoss(nn.Module):
    def __init__(self, margin=1, reduction="mean"):
        super(OnlineTripletLoss, self).__init__()
        self.margin = margin
        self.reduction = reduction

    def forward(self, embeddings, targets):
        triplets = self._hardNegativeTripletSelector(embeddings,
targets)
        distance_ap = (embeddings[triplets[:, 0]] -
embeddings[triplets[:, 1]]).pow(2).sum(1)
        distance_an = (embeddings[triplets[:, 0]] -
embeddings[triplets[:, 2]]).pow(2).sum(1)
        losses = F.relu(distance_ap - distance_an + self.margin)
        loss = losses.mean() if self.reduction == "mean" else
losses.sum()

```

```

        accuracy = ((distance_ap < distance_an).sum().item() /
len(distance_ap))
        return loss, accuracy

def _hardNegativeTripletSelector(self, embeddings, targets):
    distance_matrix = torch.cdist(embeddings, embeddings, p=2)
    triplets = []
    for label in torch.unique(targets):
        label_mask = (targets == label)
        label_indices = torch.where(label_mask)[0]
        if len(label_indices) < 2:
            continue
        negative_indices = torch.where(~label_mask)[0]
        anchor_positives = torch.combinations(label_indices, r=2)
        for anchor_positive in anchor_positives:
            ap_distance = distance_matrix[anchor_positive[0],
anchor_positive[1]]
            negative_distances =
distance_matrix[anchor_positive[0], negative_indices]
            loss_values = ap_distance - negative_distances +
self.margin
            hard_negative_idx = torch.argmax(loss_values)
            if loss_values[hard_negative_idx] > 0:
                hard_negative =
negative_indices[hard_negative_idx]
                triplets.append([anchor_positive[0].item(),
anchor_positive[1].item(), hard_negative.item()])
            if len(triplets) == 0 and len(label_indices) > 1 and
len(negative_indices) > 0:
                triplets.append([label_indices[0].item(),
label_indices[1].item(), negative_indices[0].item()])
        return torch.tensor(triplets, dtype=torch.long)

def AccuracyEmbeddingPair(output1, output2, targets, threshold):
    distances = torch.norm(output1 - output2, dim=1)
    predict = torch.where(distances <= threshold, 1, 0)
    return (predict == targets).sum().item() / len(targets)

def AccuracyEmbeddingTriplet(output1, output2, output3):
    distances_ap = torch.norm(output1 - output2, dim=1)
    distances_an = torch.norm(output1 - output3, dim=1)

```

```

predict = torch.where(distances_ap < distances_an, 1, 0)
return predict.sum().item() / len(output1)

def AccuracySigma(output, target):
    predict = torch.where(output > 0.5, 1, 0)
    return (predict == target).sum().item() / len(target)

def AccuracyConv(output1, output2, target1, target2):
    predict = (torch.argmax(output1, dim=1) == target1) &
(torch.argmax(output2, dim=1) == target2)
    return predict.sum().item() / len(predict)

```

A.3 Фрагмент файла networks.py

```

import torch.nn as nn
import torch

class EmbeddingNet(nn.Module):
    def __init__(self, in_channels):
        super(EmbeddingNet, self).__init__()
        self.convnet = nn.Sequential(nn.Conv2d(in_channels, 32, 5),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(2, stride=2),
                                      nn.Conv2d(32, 64, 5),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(2, stride=2))

        self.fc = nn.Sequential(nn.Linear(1024, 256),
                                  nn.ReLU(inplace=True),
                                  nn.Linear(256, 256),
                                  nn.ReLU(inplace=True),
                                  nn.Linear(256, 128)
                                  )

    def forward(self, input1):
        output = self.convnet(input1)
        output = output.view(output.size()[0], -1)
        output = self.fc(output)
        return output

class ConvolutionalNet(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(ConvolutionalNet, self).__init__()
        self.convnet = nn.Sequential(
            nn.Conv2d(in_channels, 32, 5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, stride=2),
            nn.Conv2d(32, 64, 5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, stride=2)

```

```

    )
    self.fc = nn.Sequential(
        nn.Linear(1024, 256),
        nn.ReLU(inplace=True),
        nn.Linear(256, 256),
        nn.ReLU(inplace=True),
        nn.Linear(256, num_classes),
        nn.Softmax(dim=1)
    )

    def forward(self, input1):
        output = self.convnet(input1)
        output = output.view(output.size(0), -1)
        output = self.fc(output)
        return output

class SiameseNetSigmoid(nn.Module):
    def __init__(self, in_channels):
        super(SiameseNetSigmoid, self).__init__()
        self.convnet = nn.Sequential(nn.Conv2d(in_channels, 32, 5),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(2, stride=2),
                                      nn.Conv2d(32, 64, 5),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(2, stride=2))

        self.fc = nn.Sequential(
            nn.Linear(1024 * 2, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, input1, input2):
        output1 = self.convnet(input1)
        output1 = output1.view(output1.size()[0], -1)
        output2 = self.convnet(input2)
        output2 = output2.view(output2.size()[0], -1)
        output = torch.cat((output1, output2), 1)
        output = self.fc(output)
        return output

```

A.4 Фрагмент файла utils.py

```

import torch, os, random
from sampler import BalancedBatchSampler
from tqdm import tqdm
from enum import Enum, auto
from matplotlib import pyplot as plt
import numpy as np

```

```

def fit(train_dataset, test_dataset, model, train_type_model,
test_type_model, train_loss_fn, test_loss_fn, optimizer, scheduler, epochs,
seed, train_batch_size, test_batch_size,
        model_name, balanced_sampler=False, train_accuracy_fn=None,
test_accuracy_fn=None):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    np.random.seed(seed)
    random.seed(seed)
    use_cuda = torch.cuda.is_available()
    if use_cuda:
        device = torch.device("cuda")
    else:
        device = torch.device("cpu")
    train_kwargs = {"batch_size": train_batch_size}
    test_kwargs = {"batch_size": test_batch_size}
    if use_cuda:
        cuda_kwargs = {"num_workers": 1, "pin_memory": True, }
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)
    if balanced_sampler:
        train_kwargs.update({"sampler":
BalancedBatchSampler(train_dataset, train_dataset.dataset.targets)})

    train_loader = torch.utils.data.DataLoader(train_dataset,
**train_kwargs)
    test_loader = torch.utils.data.DataLoader(test_dataset,
**test_kwargs)
    metrics = {"train_losses": [], "test_losses": [],
"train_accuracies": [], "test_accuracies": []}
    model.to(device)
    for epoch in range(1, epochs + 1):
        train_loss, train_accuracy = train(model, train_type_model,
device, train_loader, optimizer, train_loss_fn, epoch, train_accuracy_fn)
        test_loss, test_accuracy = test(model, test_type_model,
device, test_loader, test_loss_fn, test_accuracy_fn)
        metrics["train_losses"].append(train_loss)
        metrics["train_accuracies"].append(train_accuracy)
        metrics["test_losses"].append(test_loss)
        metrics["test_accuracies"].append(test_accuracy)
    scheduler.step()

```

```

        torch.save(model.state_dict(),                os.path.join("models",
f"{model_name}.model"))
        torch.save(metrics,                            os.path.join("models",
f"{model_name}.metrics"))
        saveGraphs(model_name, epochs, metrics)

class TypeModel(Enum):
    Conv = auto()
    Pairs = auto()
    Triplet = auto()
    Sigma = auto()
    Online = auto()

def networks_model(model, type_model, type_fit, data, loss_fn,
accuracy_fn=None, stage=None):
    match type_model:
        case TypeModel.Conv:
            input1, input2, target1, target2 = data[0], data[1],
data[2], data[3]
            output1 = model(input1)
            output2 = model(input2)
            loss = loss_fn(output1, target1)
            accuracy = accuracy_fn(output1, output2, target1, target2)
        case TypeModel.Pairs:
            input1, input2, target = data[0], data[1], data[2]
            output1 = model(input1)
            output2 = model(input2)
            loss, accuracy = loss_fn(type_fit, output1, output2,
target)
        case TypeModel.Triplet:
            input1, input2, input3 = data[0], data[1], data[2]
            output1 = model(input1)
            output2 = model(input2)
            output3 = model(input3)
            loss, accuracy = loss_fn(type_fit, output1, output2,
output3)
        case TypeModel.Sigma:
            input1, input2, target = data[0], data[1], data[2]
            output = model(input1, input2).squeeze(1)
            loss = loss_fn(output, target)

```

```

        accuracy = accuracy_fn(output, target)
    case TypeModel.Online:
        input, target = data[0], data[1]
        output = model(input)
        loss, accuracy = loss_fn(output, target)

    return loss, accuracy

def train(model, type_model, device, train_loader, optimizer, loss_fn,
epoch, accuracy_fn=None):
    model.train()
    train_tqdm = tqdm(train_loader, leave=True)
    train_loss = 0
    train_accuracy = 0
    l = len(train_loader)
    for data in train_tqdm:
        data = [x.to(device) for x in data]
        optimizer.zero_grad()
        loss, accuracy = networks_model(model, type_model, "train",
data, loss_fn, accuracy_fn=accuracy_fn)
        train_accuracy += accuracy
        train_loss += loss.item()
        loss.backward()
        optimizer.step()
        train_tqdm.set_description(f"[Epoch:      {epoch},      Loss:
{train_loss / l:.4f}, Accuracy: {train_accuracy / l:.4f}]")
    return train_loss / l, train_accuracy / l

def test(model, type_model, device, test_loader, loss_fn,
accuracy_fn=None):
    model.eval()
    test_loss = 0
    test_accuracy = 0
    l = len(test_loader)
    with torch.no_grad():
        for data in test_loader:
            data = [x.to(device) for x in data]
            loss, accuracy = networks_model(model, type_model, "test",
data, loss_fn, accuracy_fn=accuracy_fn)
            test_accuracy += accuracy
            test_loss += loss.item()

```

```

        test_accuracy = test_accuracy / 1
        test_loss = test_loss / 1
        print(f"\nTest      set:      Loss:      {test_loss:.4f},      Accuracy:
{test_accuracy:.4f})")
        return test_loss, test_accuracy

def saveGraphs(model_name, epochs, metrics):
    plt.plot(range(1, epochs + 1), metrics["train_losses"],
label="Train Loss", marker="o")
    plt.plot(range(1, epochs + 1), metrics["test_losses"], label="Test
Loss", marker="o")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title(f"{model_name} Loss")
    plt.legend()
    plt.grid()
    plt.savefig(os.path.join("graphs", f"{model_name}_Loss.png"))
    plt.clf()

    plt.plot(range(1, epochs + 1), metrics["train_accuracies"],
label="Train Accuracy", marker="o")
    plt.plot(range(1, epochs + 1), metrics["test_accuracies"],
label="Test Accuracy", marker="o")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title(f"{model_name} Accuracy")
    plt.legend()
    plt.grid()
    plt.savefig(os.path.join("graphs", f"{model_name}_Accuracy.png"))
    plt.clf()

```

ДОДАТОК Б
Слайди презентації

Національний університет «Запорізька політехніка»

Кафедра програмних засобів

Дипломна кваліфікаційна робота магістра

Дослідження та програмна реалізація розпізнавання об'єктів на зображенні на основі сіамських нейронних мереж

Виконав

ст. гр. КНТ-223м

Владислав ПРИМАКОВ

Керівник

к.т.н., доцент

Олександр СТЕПАНЕНКО

Рисунок Б.1 – Слайд № 1

ОБ'ЄКТ, ПРЕДМЕТ, МЕТА ДОСЛІДЖЕННЯ

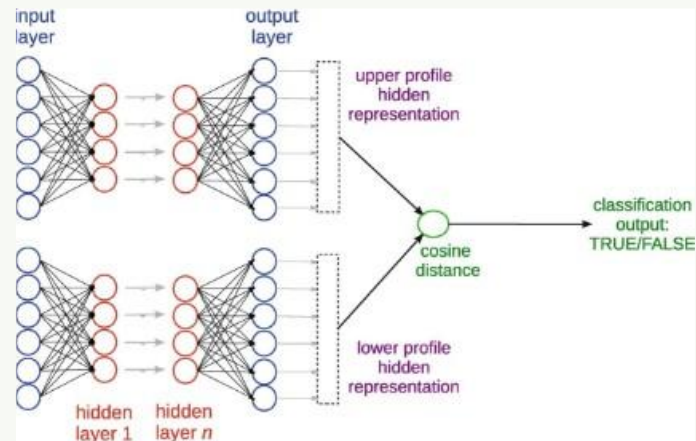
Об'єктом дослідження цієї роботи є нейронні мережі для зіставлення об'єктів на основі їхніх представлень (ембеддингів), зокрема сіамські нейронні мережі, які використовуються для задач класифікації, пошуку схожих об'єктів та зменшення розмірності даних.

Предметом дослідження є методи, архітектури та стратегії навчання сіамських нейронних мереж для задач зіставлення об'єктів на основі їхніх образів.

Метою роботи є розробка, реалізація та аналіз ефективності сіамських нейронних мереж для задач зіставлення об'єктів на основі їхніх ембеддингів.

Рисунок Б.2 – Слайд № 2

СІАМСЬКІ НЕЙРОННІ МЕРЕЖІ



Представлення структури моделі сіамської нейронної мережі

3

Рисунок Б.3 – Слайд № 3

ЗМЕНШЕННЯ РОЗМІРНОСТІ ДАНИХ ШЛЯХОМ НАВЧАННЯ ІНВАРІАНТНОГО ВІДОБРАЖЕННЯ (DRLIM)

$$I = \{X_1, \dots, X_n\}, X_i \in R^D, \forall i = 1, \dots, n, \quad (1)$$

де n – кількість екземплярів;

D – розмірність екземпляру.

$$G_W: R^D \rightarrow R^d, d \ll D. \quad (2)$$

Яка має такі властивості:

- прості міри відстані у вихідному просторі (наприклад, евклідова відстань) повинні апроксимувати відносини сусідства у вхідному просторі;
- відображення не повинно обмежуватися реалізацією простих мір відстані у вхідному просторі і повинно мати можливість вивчати інваріанти до складних перетворень;
- функція має бути вірна навіть для зразків, для яких невідомі відносини між сусідніми зразками.

4

Рисунок Б.4 – Слайд № 4

КОНТРАСТНА ФУНКЦІЯ ВТРАТ

$$X_1, X_2 \in I$$

$$Y = f(x) = \begin{cases} 0, \text{ якщо } X_1 \text{ і } X_2 \text{ є подібними} \\ 1, \text{ якщо } X_1 \text{ і } X_2 \text{ є відмінними} \end{cases}$$

$$D_W(X_1, X_2) = \|G_W(X_1) - G_W(X_2)\|_2, \quad (3)$$

$$L(W) = \sum_{i=1}^P L(W, (Y, X_1, X_2)^i), \quad (4)$$

$$L(W, (Y, X_1, X_2)^i) = (1 - Y)L_S(D_W^i) + YL_D(D_W^i), \quad (5)$$

де $(Y, X_1, X_2)^i$ – i -та помічена пара екземплярів

L_S – функція часткових втрат для пари схожих екземплярів

L_D – функція часткових втрат для пари відмінних екземплярів

P – кількість навчальних пар.

$$L(W, Y, X_1, X_2) = (1 - Y) \frac{1}{2} (D_W)^2 + Y \frac{1}{2}, \quad m > 0, \quad (6)$$

де m – відступ який визначає радіус навкол $G_W(X)$.

5

Рисунок Б.5 – Слайд № 5

ТРИПЛЕТНА ФУНКЦІЯ ВТРАТ



Триплетна функція втрат

$$L_{LMNN}(W) = (1 - \mu)L_{pull}(W) + \mu L_{push}(W), \quad (7)$$

$$L_{pull}(W) = \sum_{i,j \in T(i)} D_{i,j}, \quad (8)$$

$$L_{push}(W) = \sum_{\substack{a,n \\ y_a \neq y_n}} [m + D_{a,T(a)} - D_{a,n}]_+, \quad (9)$$

$$L_{tri}(W) = \sum_{\substack{a,p,n \\ y_a = y_p \neq y_n}} [m + D_{a,p} - D_{a,n}]_+. \quad (10)$$

6

Рисунок Б.6 – Слайд № 6

ГЕНЕРАЦІЯ ПАР/ТРИПЛЕТІВ ОНЛАЙН

$$L_{BH}(W; X) = \overbrace{\sum_{i=1}^P \sum_{a=1}^K}^{all\ anchors} \left[m + \overbrace{\max_{p=1 \dots K} D(G_W(X_a^i), G_W(X_p^i))}^{hardest\ positive} - \overbrace{\min_{\substack{j=1 \dots P \\ n=1 \dots K \\ j \neq i}} D(G_W(X_a^i), G_W(X_n^j))}^{hardest\ negative} \right]_+, \quad (11)$$

де X – партія екземплярів у пакеті;

x_i^j – відповідає j -тому екземпляру з i -того класу.

$$L_{BA}(W; X) = \overbrace{\sum_{i=1}^P \sum_{a=1}^K}^{all\ anchors} \overbrace{\sum_{p=1}^K \sum_{j=1}^P \sum_{n=1}^K}^{all\ pos\ all\ negatives} \left[m + D(G_W(X_a^i), G_W(X_p^i)) - D(G_W(X_a^i), G_W(X_n^j)) \right]_+. \quad (12)$$

$$L_{LE}(W; X) = \overbrace{\sum_{i=1}^P \sum_{a=1}^K}^{all\ anchors} \left[m + \overbrace{\log \sum_{p=1}^K e^{D(G_W(X_a^i), G_W(X_p^i))}}^{all\ positive} - \overbrace{\log \sum_{j=1}^P \sum_{n=1}^K e^{m - D(G_W(X_a^i), G_W(X_n^j))}}^{all\ negative} \right]_+. \quad (13)$$

7

Рисунок Б.7 – Слайд № 7

ВИБІР ФРЕЙМВОРКУ

Критерій	PyTorch	TensorFlow
Ключові відмінності	Підходить для тестування та дослідження	Готовий до розгортання та виробництва
Архітектура	Динамічний обчислювальний графік, який можна змінити в будь-який час	Граф статичних обчислень, який визначається лише один раз і використовується знову
Простота використання	Інтуїтивно зрозумілий інтерфейс користувача на основі Python	Швидше навчання, але більше функцій, ніж раніше
Гнучкість і філософія дизайну	Простота та адаптивність	Продуктивність і масштабованість
Вплив на практичну побудову моделі	Швидкі ітерації та налагодження моделі	Функції надійного обслуговування TensorFlow і розгортання TensorFlow Lite
Швидкість і ефективність	Для невеликих моделей і розробок це, як правило, швидше	Він призначений для масштабних моделей
Масштабованість	Ідеально підходить для малих і середніх програм і досліджень	Висока масштабованість для широкого розосередженого навчання.
Популярність	Стає все більш популярним у наукових і наукових колах	Він широко використовується в бізнесі та промисловому контексті
Спільнота та підтримка	Сильна підтримка наукової спільноти	Це велика спільнота з великою кількістю ресурсів

8

Рисунок Б.8 – Слайд № 8

ВИКОРИСТАННЯ ГЕНЕТИЧНОГО АЛГОРИТМУ ДЛЯ ОПТИМІЗАЦІЇ ВІДСТУПУ У ФУНКЦІЯ ВТРАТ



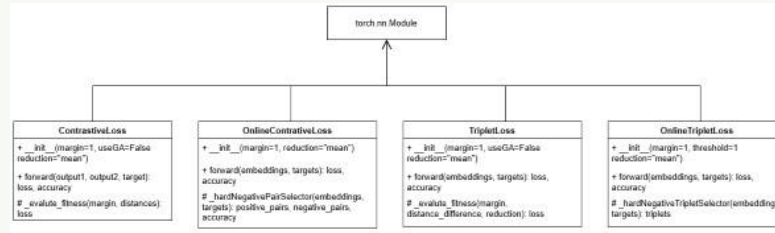
Рисунок Б.9 – Слайд № 9

СТРУКТУРИ НЕЙРОННИХ МЕРЕЖ

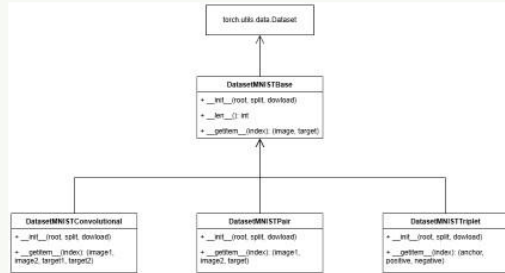


Рисунок Б.10 – Слайд № 10

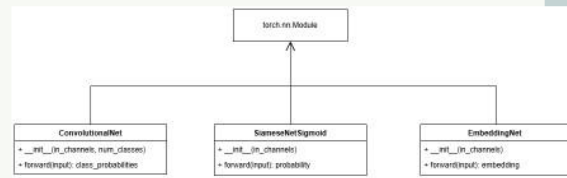
ОПИС ПРОГРАМИ



Діаграма класів функцій втрат



Діаграма класів вибірок даних

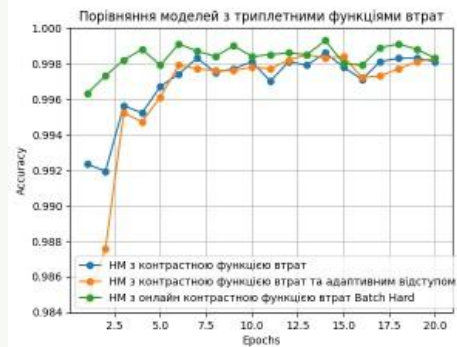
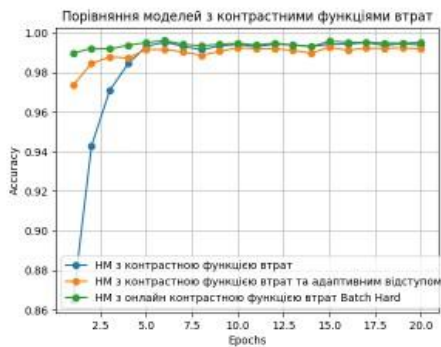


Діаграма класів нейронних мереж

11

Рисунок Б.11 – Слайд № 11

ТЕСТУВАННЯ

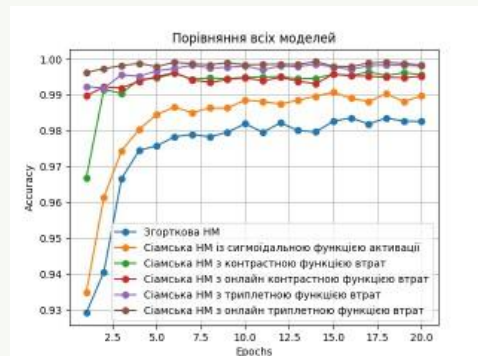


12

Рисунок Б.12 – Слайд № 12

ТЕСТУВАННЯ

Мережа	Точність на навчальній вибірці	Точність на тестовій вибірці
Згортоква мережа	0.9889	0.9826
Сіамська НМ із сигмоїдальною функцією активації	0.9883	0.9897
Сіамська НМ з контрастною функцією втрат	0.9962	0.9956
Сіамська НМ з онлайн контрастною функцією втрат (Batch Hard)	0.9999	0.9952
Сіамська НМ з триплетною функцією втрат	0.9990	0.9981
Сіамська НМ з онлайн триплетною функцією втрат	1.0000	0.9983



Порівняння роботи мереж на тестовій вибірці

13

Рисунок Б.13 – Слайд № 13

ВИСНОВОК

В процесі виконання кваліфікованої роботи магістра було досліджено та програмно реалізовано розпізнавання об'єктів на зображенні на основі сіамських нейронних мереж. Розглянуто та реалізовано різні функції втрат для сіамських НМ – контрастну та триплетну. Вони забезпечують навчання моделей для задач класифікації та пошуку схожостей між об'єктами.

Наукова новизна роботи полягає у використанні генетичного алгоритму для динамічної оптимізації параметра відступу в контрастних і триплетних функціях втрат. Це дозволяє враховувати специфіку кожного пакета даних, підвищуючи адаптивність моделі, її здатність до узагальнення та точність роботи в задачах пошуку схожих об'єктів чи класифікації.

PyTorch обрано за гнучкість і простоту, а MNIST — як стандартну вибірку для навчання моделей обробки зображень. Вибрано архітектури згорткових мереж із балансом між складністю та здатністю до узагальнення. Створено класи для навчання, тестування та збереження результатів, проведено експериментальні дослідження ефективності архітектур і стратегій навчання.

14

Рисунок Б.14 – Слайд № 14