Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Розробка інструментів для дослідження методів підвищення результатів тесту ARC-AGI (Abstraction and Reasoning Corpus for Artificial General Intelligence)
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-21-2
Єлизавета Котляр
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)
Керівник ас. Ірина Малєєва
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Олег ЗОЛОТУХІН
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Котляр Єлизаветі Сергіївні
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка інструментів для дослідження методів підвищення результатів тесту ARC-AGI (Abstraction and Reasoning Corpus for Artificial General Intelligence)

затверджена наказом університету від 19 травня 20 25 р. № 378 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 19 червня 20 25 р.

3. Вихідні дані до роботи Науково-технічні публікації, дані Інтернет-джерел та відомих наукових проектів, Python documentation, набір даних для тренування та тестування системи, платформа Kaggle

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі та постановка задачі
2) Практична реалізація створення системи ШІ
3) Результати досліджень

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	19.05.2025	виконано
2	Аналіз предметної галузі	19.05.2025 – 22.05.2025	виконано
3	Постановка задачі	22.05.2025 – 23.05.2025	виконано
4	Проведення теоретичних досліджень	23.05.2025 – 29.05.2025	виконано
5	Вибір технологій та засобів розробки	29.05.2025 – 30.05.2025	виконано
6	Реалізація проекту	30.05.2025 – 10.06.2025	виконано
7	Оформлення пояснювальної записки	10.06.2025 – 14.06.2025	виконано
8	Надання роботи на рецензування	14.06.2025	виконано
9	Перевірка пояснювальної записки на плагіат	15.06.2025	виконано
10	Попередній захист роботи з керівником	16.06.2025	виконано
11	Захист перед ЕК	19.06.2025	

Дата видачі завдання 19 травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Ірина Малєєва
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 84 с., 14 рис., 3 дод., 21 джерел.

АУТОЕНКОДЕР, ГЛИБИННА НЕЙРОННА МЕРЕЖА, ЗГОРТКОВА НЕЙРОННА МЕРЕЖА, КОРПУС АБСТРАКЦІЙ І МІРКУВАНЬ ДЛЯ ЗАГАЛЬНОГО ШТУЧНОГО ІНТЕЛЕКТУ, ABSTRACTION AND REASONING CORPUS FOR ARTIFICIAL GENERAL INTELLIGENCE ARC-AGI.

Об’єкт дослідження – ШІ (штучний інтелект), як система, що виконує завдання абстрагування та міркування, зокрема в умовах, подібних до задач тесту ARC (Abstraction and Reasoning Corpus), що імітують здатності загального інтелекту (AGI (Artificial General Intelligence)).

Предмет дослідження – використання аутоенкодера на згортковій нейронній мережі в якості трансдуктивної моделі при застосуванні оригінального варіанту TTT (тонкого налаштування під час тестування), з пошуком у прихованому (латентному) просторі моделі.

Мета роботи – створення системи ШІ здатної ефективно набувати нових навичок незалежно від даних навчання, шляхом створення архітектури нейронної мережі, її навчання, та подальшого використання отриманих результатів.

Методи дослідження – теоретичний (збір та структуризація теоретичного матеріалу), експериментальний (програмна реалізація нейронної мережі та її навчання). Методи розробки базуються на технологіях Python з фреймворком tensorflow.

Результатом роботи є досягнута точність співпадіння прогнозів перетворення вхідних зразків мережі з оригіналами міток понад 90% при використанні впродовж навчання трансдуктивної моделі в якості тренувальних даних не більше 50 зразків.

ABSTRACT

Bachelor’s thesis contains: 84 pp., 14 fig., 3 ann., 21 references.

ABSTRACTION AND REASONING CORPUS FOR ARTIFICIAL GENERAL INTELLIGENCE, ARC-AGI, AUTOENCODER, CONVOLUTIONAL NEURAL NETWORK, DEEP NEURAL NETWORK.

The object of research is AI (artificial intelligence), as a system that performs abstraction and reasoning tasks, in particular in conditions similar to the tasks of the ARC (Abstraction and Reasoning Corpus) test, which simulate the abilities of general intelligence (AGI (Artificial General Intelligence)).

The subject of research is the use of an autoencoder on a convolutional neural network as a transductive model when applying the original version of TTT (fine tuning during testing), with a search in the hidden (latent) space of the model.

The purpose of the work is to create an AI system capable of effectively acquiring new skills regardless of the training data, by creating the architecture of a neural network, its training, and further use of the obtained results.

Research methods are theoretical (collection and structuring of theoretical material), experimental (software implementation of a neural network and its training). Development methods are based on Python technologies with the tensorflow framework.

The result of the work is the achieved accuracy of matching the predictions of the transformation of the input samples of the network with the original labels of over 90% when using no more than 50 samples as training data during the training of the transductive model.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Аутоенкодер – нейронна мережа, яка копіює вхідні дані на вихід. Аутоенкодери стискають вхідні дані для представлення їх у latent-space (прихований простір), а потім відновлюють з цього представлення вихідні дані;

Декодер – частина аутоенкодера, що відновлює (розтискає) дані з прихованого (латентного) простору у вихідні дані;

ДНК – дезоксирибонуклеїнова кислота;

Енкодер – частина аутоенкодера, що стискає вхідні дані для представлення їх у latent-space (прихований простір);

Трансдуктивна модель – це модель, яка намагається безпосередньо передбачити вихідні дані з урахуванням тестових вхідних даних та специфікації задачі, замість того, щоб спочатку намагатися написати програму, яка відповідає завданню;

ШІ – штучний інтелект;

AGI – Artificial General Intelligence – штучний інтелект загального призначення;

ARC-AGI – Abstraction and Reasoning Corpus for Artificial General Intelligence – Корпус абстракцій і міркувань для загального штучного інтелекту. Спеціально розроблений тест;

GPT-3 – третє покоління алгоритму обробки природної мови від компанії OpenAI;

IDE – Integrated Development Environment – Інтегроване середовище розробки, також єдине середовище розробки, ECP – комплекс програмних засобів, що використовується програмістами для розробки програмного забезпечення;

LLM – Large Language Model – велика мовна модель.

ВСТУП

Поточні системи ШІ не можуть узагальнювати нові проблеми за межами їх навчальних даних, незважаючи на широке навчання на великих наборах даних. LLM вивели ШІ на масовий рівень для великої кількості відомих завдань. Проте прогрес у напрямі штучного інтелекту загального призначення AGI зупинився. Поліпшення в AGI можуть дозволити системам ШІ думати та винаходити разом із людьми.

У 2019 році Франсуа Шолле, творець Keras, бібліотеки глибокого навчання з відкритим вихідним кодом, прийнятої більш ніж 2,5 млн розробників, опублікував впливову статтю «Про вимір інтелекту», в якій представив еталонний тест «Корпус абстракцій і міркувань для загального штучного інтелекту при вирішенні невідомих завдань (ARC-AGI) [1].

Щоб зробити усвідомлений прогрес у напрямі розумніших і людиноподібних систем, нам слід дотримуватися відповідного сигналу зворотного зв'язку. Нам потрібно визначити та оцінити інтелект.

Ці визначення та оцінки стають орієнтирами, які використовуються для вимірювання прогресу у створенні систем, здатних мислити та винаходити разом з нами.

Загальноприйняте визначення AGI – «система, здатна автоматизувати більшу частину економічно цінної роботи» – хоч і є корисною метою, але є неправильним виміром інтелекту.

Вимірювання навичок, пов'язаних із виконанням конкретних завдань, не є добрим показником інтелекту.

Навичка сильно залежить від попередніх знань та досвіду. Необмежені апріорні дані або необмежені дані навчання дозволяють розробникам купувати рівні навичок для системи. Це маскує свою силу узагальнення системи.

Інтелект полягає у широких чи універсальних здібностях; він характеризується набуттям навичок та узагальненням, а не самими навичками.

AGI – це система, здатна ефективно набувати нових навичок незалежно від даних навчання.

Формальніше: Інтелект системи – це міра ефективності набуття нею навичок у рамках певного кола завдань з урахуванням апріорних даних, досвіду та складності узагальнення [1].

Це означає, що система здатна адаптуватися до нових проблем, з якими раніше не стикалася і які її творці (розробники) не передбачали.

ARC-AGI – єдиний еталон ШІ, який вимірює прогрес людства на шляху до загального рівня інтелекту.

ARC-AGI складається з унікальних завдань навчання та оцінки. Кожне завдання містить приклади введення–виводу. Входи і виходи, схожі на головоломки (рисунок В.1), є сіткою, де кожен квадрат може бути одного з десяти кольорів. Сітка може бути будь-якої висоти або ширини від 1x1 до 30x30 і більше.

Для успішного вирішення завдання тестований повинен створити точну сітку виводу для оцінки. Це включає вибір правильних розмірів сітки виведення.

ARC-AGI спеціально розроблено для порівняння штучного інтелекту з людським. Для цього ARC-AGI явно перераховує попередні знання, які мають людина, щоб забезпечити справедливу основу для порівняння. Ці «основні попередні знання» – це ті, які люди мають від природи, навіть у дитинстві:

– об'єктність. Об'єкти зберігаються і не можуть з'являтись або зникати без причини. Об'єкти можуть взаємодіяти чи ні залежно обставин;

– цілеспрямованість. Об'єкти можуть бути одухотвореними та неживими. Деякі об'єкти є «агентами» – вони мають наміри, і вони мають на меті;

– числа та рахунок. Предмети можна підраховувати або сортувати за формою, зовнішнім виглядом або рухом, використовуючи базові математичні дії, такі як додавання, віднімання та порівняння;

– базова геометрія та топологія. Об'єкти можуть мати форму прямокутника, трикутника та кола, які можна дзеркально відбивати, обертати, переносити, деформувати, комбінувати, повторювати тощо. Можна виявити різницю у відстанях.

ARC-AGI уникає опори на будь-яку інформацію, яка не є частиною цих апріорних знань, наприклад, набутих або культурних знань, таких як мова.

Вирішення проблеми ARC-AGI є реальним кроком на шляху до AGI.

Як мінімум, рішення ARC-AGI призведе до нової парадигми програмування. Це дозволить будь-якій людині, навіть без знань програмування, створювати програми, просто надаючи кілька прикладів введення-виведення того, що вона хоче.

Це значно розширить коло тих, хто може використовувати програмне забезпечення та автоматизацію. Програми могли б автоматично вдосконалюватися при отриманні нових даних, подібно до того, як навчаються люди.

Якщо рішення ARC-AGI буде знайдено, воно матиме більший вплив, ніж відкриття архітектури Transformer (один з видів архітектури глибоких нейронних мереж). Рішення відкриє нову галузь технологій.

Ця робота зосереджена на пошуку інструментів дослідження ідеї за межами LLM, які значною мірою залежать від великих наборів даних та стикаються з новими проблемами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис предметної галузі

Коротко ШІ можна визначити так: спроба автоматизації інтелектуальних завдань, які зазвичай виконують люди. Відповідно, ШІ – це область, що охоплює машинне та глибоке навчання, а також включає багато підходів, з навчанням не пов'язаних.

Довгий час багато експертів вважали, що штучний інтелект рівня людини можна створити, якщо надати програмісту достатній набір явних правил для маніпулювання знаннями. Цей підхід, відомий як символічний ШІ.

Символічний ШІ чудово справлявся із чітко визначеними логічними завданнями (такими як гра у шахи). Але, як виявилося, це не працювало для складніших і менш чітких випадків (наприклад, для класифікації зображень, розпізнавання мови та перекладу іншими мовами), адже для їх вирішення задати суворі правила неможливо. Тому на зміну символічному ШІ прийшов новий підхід: машинне навчання.

У машинному навчанні система навчається, а не програмується явно. Їй передаються численні приклади, що стосуються даної задачі, а вона знаходить там статистичну структуру, яка дозволяє виробити відповідні правила для вирішення цього завдання.

Глибоке навчання – особливий розділ машинного навчання, новий підхід до пошуку представлення даних, наголошує на вивченні послідовних верств (чи рівнів) дедалі значних уявлень. Число шарів, на які ділиться модель даних, називають глибиною моделі.

У глибокому навчанні такі багатошарові уявлення розглядаються (майже завжди) з використанням нейронних мереж – моделей, структурованих як шари, накладені друг на друга [2].

Машинне навчання та глибоке навчання – області скоріше емпіричні, ніж теоретичні. Ви експериментуєте з багатьма моделями і налаштовуєте їх, поки не знайдете моделі, які найкраще підходять для ваших застосунків [3].

Глибоке навчання добре працює при великих обсягах даних, але може бути ефективним при менших обсягах даних у поєднанні з такими методами, як перенесення навчання та збільшення даних [4], [5]. Тим не менш, у загальному випадку чим більше у вас даних, тим краще ви зможете навчити модель глибокого навчання [3]. Останнє складно назвати інтелектом: воно не мислить у звичному для нас значенні слова, а обробляє великі масиви даних [6].

Франсуа Шолле пропонує новий підхід до створення та оцінки ШІ.

По-перше, інтелект потрібно вимірювати не тим, яких цілей він досяг, а тим, наскільки ефективно він набуває навичок. При навчанні нейромережі вражаючі результати виходять не за рахунок «розуму» програми, а за рахунок величезної кількості даних.

По-друге, Шолле у 2019 році розробив кілька варіантів тесту ARC-AGI, що може оцінювати інтелектуальність як ШІ, так і людини.

По-третє, Шолле пропонує створити нові інструменти для порівняння інтелектів, враховуючи здатність до вирішення нових та нестандартних завдань та вміння логічно мислити. Нині ці якості повною мірою притаманні лише людям [7].

Виходячи з вищесказаного, рай на Землі або повстання машин доведеться трохи відкласти. Принаймні, допоки ШІ не стане справжнім інтелектом.

1.2 Порівняльна характеристика аналогів

У наступні роки після початкового випуску ARC-AGI чисті підходи до глибокого навчання показали погані результати на ARC-AGI, оскільки класична парадигма глибокого навчання працює, пов'язуючи нові ситуації з

ситуаціями, що спостерігаються під час навчання, без адаптації або підлаштовування знань під час тестування, що унеможливлює для таких моделей вирішення завдань ARC-AGI.

Впродовж 2020 року жоден підхід, заснований на глибокому навчанні, не набрав більше ніж 1% співпадінь прогнозів з оригіналами. Оригінальна модель GPT-3 від компанії OpenAI набрала 0% за результатами публічної оцінки за допомогою прямих підказок.

Ось чому, незважаючи на те, що ARC-AGI був створений до появи LLM, він чинить опір зростанню числа LLM у період 2022–2024 років.

Перше змагання по вирішенню ARC-AGI відбулося у 2020 році з найвищим балом 20%. Через чотири роки найвищий бал збільшився лише до 33%. Цю відсутність прогресу в ARC-AGI можна пояснити відсутністю прогресу у напрямі AGI. З 2020 до початку 2024 року в галузі досліджень ШІ домінувало масштабування систем глибокого навчання, які підвищували навички, пов'язані з конкретними завданнями, але не покращували здатність вирішувати завдання без доступних навчальних даних під час навчання (тобто штучний інтелект загального призначення). Прогрес у напрямку AGI зупинився в цей період – системи ШІ ставали більшими і запам'ятовували все більше навчальних даних, але спільність у передових системах ШІ була приблизно статичною [8].

Проте прогрес знову прискорився у 2024 році (рисунок 1.1), чому сприяли три основні категорії підходів:

– синтез програм, керований глибоким навчанням: використання моделей глибокого навчання, зокрема спеціалізованих кодів LLM, для створення програм вирішення завдань або управління процесом пошуку програм за межами сліпих методів перебору;

– тестове навчання (далі TTT) для трансдуктивних моделей: точне налаштування LLM під час навчання за заданою специфікацією задачі ARC-AGI з метою підлаштування попередніх знань LLM у нову модель, адаптовану до поставленого завдання;

– об'єднання програмного синтезу з трансдуктивними моделями: об'єднання двох підходів, описаних вище, в один суперпідхід, що ґрунтується на спостереженні, що кожен підхід має тенденцію вирішувати різні види завдань.

Слід зауважити, що команда ARChitects, що досягла максимального співпадіння прогнозів з оригіналами у 2024 році використала TTT, щоб набрати 53,5%, а наступний за ними, Екін Акюрек та команда, використали TTT, щоб набрати 47,5%.

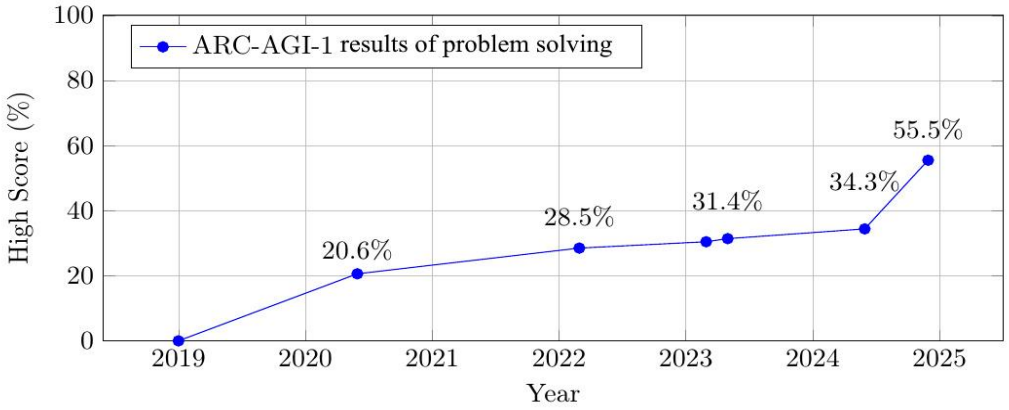

Рисунок 1.1 – Прогрес рішення ARC-AGI

Сьогодні всі провідні підходи до трансдукції на основі LLM для ARC-AGI використовують TTT, і не існує жодного рішення щодо трансдукції в стилі класичного перебору великої кількості зразків, яке набрало б понад 11%. Цей різкий розрив наголошує на нездатності класичної парадигми глибокого навчання узагальнюватися на нові завдання [8].

Новий варіант TTT, який було застосовано, в одному з можливих видів, у даній роботі, включає пошук у прихованому (латентному) просторі моделі. Цей підхід використовує випадковий пошук та градієнтний спуск для пошуку кращих уявлень програм у прихованому просторі моделі на рівні стислих даних у енкодері – новий підхід до адаптації під час тестування, який не є тонким налаштуванням, ні дискретним пошуком [8].

1.3 Методи створення системи ШІ, яка виконує задачі ARC–AGI

На початку руху до досягнення заявленої мети було випробувано різноманітні методики. Оскільки вхідні дані ARC-AGI перетворюються у вихідні нелінійно, спершу було спробувано використання нейронної мережі з повнозв’язковими шарами типу Dense та архітектурою для вирішення задачі регресії.

Проте випробування мереж подібного типу показало, що для досягнення, хоч якого прогресу у вирішенні ARC-AGI, необхідно використовувати значну кількість шарів у мережі, що тягне за собою велику кількість змінних параметрів, себто, ваг і як наслідок, потребує багато оперативної пам’яті та часу на навчання моделі. Нажаль, подібні випробування не дали точності перетворення даних більш ніж 5%, що, звісно, незадовільно.

Окрім сказаного, обмеження кількості вхідних зразків даних, яке було прийняте у кількості не більше 80 зразків (яких для навчання будь-якої моделі зовсім недостатньо), теж вимагало додаткових заходів для штучного збільшення вхідних даних, щоб раннє перенавчання не заважало роботі.

В цей час виникла ідея спробувати застосувати аналогію для пошуку шляху вирішення проблеми, як свого часу, вчинили на зорі створення нейронних мереж.

Рішенням ARC-AGI [9] є знаходження узагальненого алгоритму вирішення раніше майже не знайомого завдання. Для ознайомлення з незнайомим завданням представлені кілька подібних готових рішень, які включають деякі вхідні дані та відповідні вихідні дані. Досліджуючи готові рішення, необхідно знайти узагальнений алгоритм і нейронну мережу, здатну відтворити шукане рішення, тобто відсутні вихідні дані.

Використовуючи аналогію з людиною, завдання можна сформулювати інакше. Людина має якусь потребу, що вона хоче задовольнити. Потреба описується готовими рішеннями, які показують що

людина бажає отримати в результаті перетворень вхідного стану, представленого вхідними даними. Іншими словами, є вхідний стан (вхідні дані), але хочеться отримати щось краще і більше, щось корисніше, бажане (вихідні дані, що шукаються).

Це щось бажане описується, пояснюється, уточнюється за допомогою кількох готових рішень (є вихідні стани, тобто, є отримані бажані вихідні дані). Процес задоволення потреби полягає у застосуванні певної шуканої функціональної залежності вихідних даних (бажаного стану, якого прагне людина у процесі задоволення своєї потреби) від вхідних даних (вхідного стану, що викликав потребу).

Пошук шуканої функціональної залежності здійснюється за допомогою кількох готових рішень тренувального набору (як пояснень того, що хочеться досягти) та нейронної мережі (ШІ, який вирішує це нове, незнайоме раніше для нього завдання).

Оскільки вихідні дані є наслідком вхідних даних, логічно припустити, що між ними існує певна функціональна залежність, за допомогою якої, знаючи вхідні дані, можна знайти вихідні.

Слід врахувати, що в живій природі немає чітких абсолютних залежностей, а всі залежності імовірнісні і при повторі одного і того ж рішення отримуємо відповідь трохи відмінну від попередньої, але з високим ступенем ймовірності дорівнюючу йому. Те саме видно і в нейронних мережах, де ваги ініціалізуються випадковим чином і, отже, через відмінні початкові умови рішення щоразу будуть декілька (на дуже малу частку) відмінні одне від одного.

Отже, вхідні дані за аналогією були прийняті як дитина, а вихідні дані як доросла людина, стосовно людей (те ж саме можна сказати і про інших представників тваринного світу).

Зараз ні для кого не секрет, що по молекулі ДНК можна однозначно визначити споріднений зв'язок між батьками та їхніми дітьми, але з певним ступенем ймовірності. Якщо досліджувати закономірності зростання

людини всі апріорно дані їй властивості, навички, які збережуться і, навіть, можуть розвинутися у процесі життя, щодо його ДНК, можна знайти якусь функціональну залежність між дорослою людиною і нею ж у дитинстві зі знання його ДНК. У ДНК людини записані закономірності зростання, що підпорядковуються тій функціональної залежності, про яку говориться.

У всіх людей, які виросли в однакових зовнішніх умовах, ця узагальнена функціональна залежність властивостей дорослої людини від властивостей дитини однакова з високою ймовірністю. Таким чином, якщо знайти цю шукану функціональну залежність в однієї групи людей, то з високим ступенем ймовірності можна сказати, що в іншої групи подібних людей, що виросла в подібних зовнішніх умовах, ця функціональна залежність, що шукається нами, буде такою ж.

Тобто, повертаючись до конкретної задачі кваліфікаційної роботи, якщо вдасться знайти функціональну залежність між тренувальними наборами вхідних та відповідних їм вихідних даних, то застосувавши її до контрольного значення вхідних даних, буде можливо знайти вихідні дані, що шукаються, з високим ступенем ймовірності.

Відомо, що згорткові нейронні мережі заглиблюючись з кожним шаром, знаходять все більш загальні властивості досліджуваних даних, водночас зменшуючи розмір останніх. Таким чином, згорткова нейронна мережа стискає досліджувані дані, над якими в кожному шарі проводиться операція згортки, і на виході у прихованому (латентному) просторі отримує найбільш загальні властивості, які можна назвати як головні або основні. За прийнятою аналогією, ці загальні, головні чи основні властивості є ні що інше, як ДНК. Як знайти залежність між властивостями дитини і дорослого, знаючи ДНК? По зворотній аналогії, необхідно декодувати загальні, основні характеристики (ДНК). Тут головне в тому, що брати як вхідні та вихідні дані при навчанні нейронної мережі.

Якщо брати вхідні дані, то вийде, так званий аутоенкодер (тип нейронної мережі), що складається з енкодера, що виводить на виході ДНК,

як сказано за аналогією, і декодера, який повертає на виході початкові, тобто вхідні дані, але з високим ступенем ймовірності, так як в процесі стиснення в енкодері і декомпресії в декодері з’являться певні неточності [10][11].

А якщо взяти в якості вхідних даних вхідні дані тренувальних екземплярів, а в якості вихідних даних вихідні дані тренувальних екземплярів і навчити на них модель, то можливо отримати навчену нейронну мережу, що приховує у своїх внутрішніх прихованих (латентних) шарах шукану в кваліфікаційній роботі функціональну залежність між вхідними і вихідними даними або за зворотною аналогією залежність між дитиною та дорослою людиною, виходячи з її ДНК.

1.4 Постановка задачі

Метою роботи – є створення системи ШІ, здатної ефективно набувати нових навичок незалежно від даних навчання, шляхом створення архітектури нейронної мережі, її навчання, та подальшого використання отриманих результатів для, як мінімум, створювання програм, просто надаючи кілька прикладів введення-виведення бажаного.

Успішне втілення поставленої задачі передбачає виконання наступних завдань:

– аналіз предметної галузі;

– дослідження моделей нейронних мереж для отримання вихідних даних з мінімальною кількістю вхідних;

– проектування архітектури нейронної мережі;

– вибір мови програмування та технологій для реалізації;

– тестування коректності роботи програми.

Отже, задачею цієї програми є реалізація ефективного та точного алгоритму та архітектури нейронної мережі для отримання можливості останньою набувати нових навичок незалежно від даних навчання.

2 ПРАКТИЧНА РЕАЛІЗАЦІЯ СТВОРЕННЯ СИСТЕМИ ШІ

2.1 Умови для роботи системи ШІ

Для забезпечення надійної та результативної роботи нейронної мережі необхідно попередньо вжити певні заходи.

«Поняття «достатня кількість зразків» досить відносно – насамперед щодо розміру та глибини моделі, що навчається. Не можна навчити згорткову нейронну мережу розв'язанню складного завдання на кількох десятках зразків, а от кількох сотень цілком може вистачити, якщо модель невелика і добре регуляризована, а вирішуване завдання просте» [2].

В нашому випадку кількість вхідних навчальних тренувальних даних ARC-AGI, що складаються з 4–5 десятків зразків, для одного контрольного зразка треба штучно збільшити, щоб досягти вищезазначеної «достатньої кількості зразків».

Для цього спочатку визначимо максимально можливий розмір матриці в тренувальному наборі, а згодом збільшимо її розмір, з яким навчатимемо мережу на стільки, наскільки буде можливо в умовах обмежень на використання оперативної пам’яті комп’ютера користувача або обмежень на використання інтернет ресурсів, накшталт, Kaggle, та обмеження на час роботи під час навчання (не більше 3 діб на одне навчання на комп’ютері користувача). Ці межі визначаються експериментально крім максимально можливого розміру матриці в тренувальному наборі.

Розроблено функцію, що створює згорткові мережі з різними початковими розмірами матриць вхідних даних і різними ступенями стиснення (тобто з різними кінцевими розмірами матриці вхідних даних в енкодері), та декодує дані до початкового розміру матриці вхідних даних. Більш докладний опис цієї функції буде далі.

Далі подбали про зменшення оперативної пам'яті, що використовується. Для цього значення змінних, що зберігають

дані (матриці), вирішено зберігати відразу у файли типу *.h5. Потім за допомогою генератора з цього файлу пакетами завантажувались дані під час навчання моделі, скоротивши тим самим використання оперативної пам'яті. Однак збільшиться час роботи програми.

Ще декілька функцій потрібні для зручної роботи мережі при навчанні і головним чином стосуються збереження у вихідних даних (матриць) потрібних розмірів.

2.2 Методика проведення досліджень

2.2.1 Обробка даних

У ARC-AGI, що використовується в роботі, зразки даних у вигляді матриць (2D масивів), що мають у комірках значення, які є цілими числами від 0 до 9, що для зручності візуалізації відображаються певним кольором, та знаходяться у файлах типу *.json (наприклад 0d3d703.json).

Вони розподілені на групи (ключі, кожен ключ це окремий файл), в яких знаходяться зразки даних, які в свою чергу, розподілені на контрольні (test) та тренувальні (train), а ті в свою чергу на вхідні (input) та вихідні (output тільки для тренувальних даних).

Розподіл виконано у форматі словників, де ключами є назви ключів (груп) (кожна назва довжиною 7 символів) та назви контрольних, тренувальних, вхідних та вихідних матриць у наборах:

```
"0057622":{"test":[{"input":[...]}],
"train":[{"input":[...]  "output":[...]},  {"input":[...]
"output":[...]}]}
```

Спочатку встановлено значення змінних, що задають кількість даних з ARC-AGI, що будуть використані для дослідження та співвідношення між ними (лістинг 2.1).

Лістинг 2.1 – Попередні установки даних

```
    # перевірочна доля для варіанту навчального та перевірочного
    tt_size = 0.3
    # тренування на навчальному та перевірочному наборах
    train_to_comp = False
    # кількість контрольних тестових матриць
    count_matrix_test_contr = 1
    # кількість тестових матриць для навчання УСЬОГО
    count_matrix_test = 80
    # кількість тестових матриць для навчання ПЕРВИННОГО
    count_matrix_test_f = 50
    # кількість тестових матриць для навчання ВТОРИННОГО
    count_matrix_test_s = count_matrix_test - count_matrix_test_f
    # кількість тестових вторинних матриць, які будуть використані у навчанні
    count_matrix_test_s_vyk = 10
    # вибір навчання ПЕРВИННОГО чи ВТОРИННОГО
    first_train = True
    # діапазон даних моделі(0-1, 0-9)
    diapazon_z_one = False
    if diapazon_z_one:
        diapazon = 'z_one'
    else:
        diapazon = 'z_nine'
    # стартовий ключ тренування завдання
    start_ind = 7
    # кількість ключів для тренування по замовчанню
    num_keys = 1
    # розмір ядра крайніх шарів згорткової мережі (3, 5)
    kernel_size_extreme = 5
    # Коефіцієнт мозаїки
    k_mozaik = 1
    # розмір пакетів при навчанні
```

Продовження лістингу 2.1

```
batch_size = 64
# кількість епох навчання
count_epoch = 16
```

Після зчитування даних з файлу у список, останній обов’язково перемішується:

```
list_dict = list(json.load(fp))
# Перемішування елементів у списку тренувальних матриць
random.shuffle(list_dict)
```

Розподілено дані на первинні та вторинні, тобто дані, що будуть використані для попереднього навчання мережі (ПЕРВИННІ) та дані (ВТОРИННІ), що будуть використані для наступного навчання мережі, коли попередньо навчена мережа буде донавчатися.

Окрім цього, всі вхідні та вихідні дані було розподілено на тренувальні набори матриць (train) та тестові, контрольні набори матриць (test). Тренувальні набори використовувались для навчання мережі, де вихідні дані були в якості справжніх зразків перетворених матриць, а тестові контрольні набори використовувались для виконання мережею прогнозів після навчання. При цьому вихідні дані тестових контрольних наборів (в нашому випадку їх використовували, на відміну від оригінального тестового набору) використовувались для остаточного повного порівняння з прогнозом мережі.

Збагачення даних було виконано шляхом поворотів та віддзеркалювання матриць (лістинг 2.2). Окрім цього, оригінальні матриці даних збільшувались у розмірі з заповненням пустих комірок 0. Це давало змогу переміщувати трансформовані оригінальні матриці всередині збільшеної по вертикалі та по горизонталі матриці та за рахунок цього додатково збагачувати дані. Також було враховано можливість мозаїчного перетворення матриць, в якості одного з видів збагачення даних.

Лістинг 2.2 – Трансформація оригінальної матриці

```
def generate_transformations(matrix_in, matrix_out):
    """
    Генерує всі унікальні трансформації (повороти та відображення)
    для квадратної матриці.
    :param matrix: вхідна квадратна матриця (numpy array)
    :return: список унікальних матриць
    """
    transformations_in = []
    transformations_out = []
    # Додаємо повороти на 0°, 90°, 180°, 270°
    for i in range(4):
        rotated = np.rot90(matrix_in, k=i)
        transformations_in.append(rotated)
        rotated = np.rot90(matrix_out, k=i)
        transformations_out.append(rotated)
    # Додаємо відображення та їх повороти
    flipped_in = np.flipud(matrix_in)  # Відображення по горизонталі
    flipped_out = np.flipud(matrix_out)  # Відображення по горизонталі
    for i in range(4):
        transformations_in.append(np.rot90(flipped_in, k=i))
        transformations_out.append(np.rot90(flipped_out, k=i))
    # Додаємо відображення та їх повороти
    flipped_in = np.fliplr(matrix_in)  # Відображення по вертикалі
    flipped_out = np.fliplr(matrix_out)  # Відображення по вертикалі
    for i in range(4):
        transformations_in.append(np.rot90(flipped_in, k=i))
```

Продовження лістингу 2.2

```
            transformations_out.append(np.rot90(flipped_out,
k=i))
        # Фільтруємо лише унікальні матриці відносно вхідних а
вихідні, беремо за індексами вхідних унікальних
        unique_transformations_in = []
        unique_transformations_out = []
        unique_ind = []
        for ind, t in enumerate(transformations_in):
            if not any(np.array_equal(t, u) for u in
unique_transformations_in):
                unique_transformations_in.append(t)
                unique_ind.append(ind)
        for ind in unique_ind:

unique_transformations_out.append(transformations_out[ind])
        return                      unique_transformations_in,
unique_transformations_out
```

Для зменшення оперативної пам’яті, що використовується, збагачені дані потрібно зберегти на локальному диску у файл типу .h5 лістинг 2.3.

Лістинг 2.3 – Створення файлу типу .h5

```
with h5py.File(filedata, "w") as f:
        dataset_x    =    f.create_dataset("x",    shape=(0,
in_shape[0],  in_shape[1],  1),  maxshape=(None,  in_shape[0],
in_shape[1], 1), dtype='float16')
        dataset_y    =    f.create_dataset("y",    shape=(0,
in_shape[0],  in_shape[1],  1),  maxshape=(None,  in_shape[0],
in_shape[1], 1), dtype='float16')
        print(f'Створено пустий файл даних {filedata}\n')
```

Далі, під час збагачення даних, цей файл буде заповнюватися даними, а після заповнення дані у файлі будуть перемішані та розділені на

навчальний та на перевірочний набір, що буде застосовано при роботі з мережею. З цих навчального та перевірочного наборів за допомогою функцій генераторів дані будуть подаватися пакетами безпосередньо у нейронну мережу підчас навчання.

2.2.2 Створення та навчання мережі для оптимізації гіперпараметрів

Головним завданням цієї роботи було знаходження оптимальних гіперпараметрів нейронної мережі, при яких досягаються результати перетворення матриць даних з максимально можливою точністю.

Після підготовки даних настає черга створення нейронної мережі та навчання її на цих даних.

У кінці кожного циклу навчання мережі зберігалися основні кількісні та якісні параметри, що описують результат перетворення даних та на їх основі приймалося рішення про зміну гіперпараметрів мережі в ту чи іншу сторону або відмову від деяких з них.

Такі цикли навчання проводились до тих пір, поки дозволяли спроможності комп’ютерної техніки та інтернет ресурсів (Kaggle), що використовувалася.

При досягненні максимальних значень параметрів точності перетворення даних мережею, гіперпараметри фіксувалися, а файл мережі зберігався на локальному диску.

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕНЬ

3.1 Оптимізація результатів дослідження системи ШІ

Пошук гіперпараметрів, що оптимізують результати роботи нейронної мережі більш емпіричний ніж теоретичний шлях. Гіперпараметри це параметри архітектури моделі, які не навчаються під час зворотнього розповсюдження помилки [12].

На практиці у якості гіперпараметрів, що досліджуються були взяті наступні:

– розмір пакету даних;

– кількість епох навчання;

– швидкість навчання;

– розмір ядра крайніх шарів згорткової мережі;

– тип ініціалізації ваг шарів;

– оптимізатор;

– функція втрат;

– метрика;

– розмір матриці даних;

– типи шарів аутоенкодера;

– кількість шарів аутоенкодера;

– кількість фільтрів у кожному шарі аутоенкодера.

Окрім гіперпараметрів для оптимізації аутоенкодера важливі й інші дані, що впливають на кількість вхідних даних: кількість зразків обраного ключа вхідних даних, кількість варіантів збагачення даних.

3.2 Результати навчання аутоенкодера

Для перевірки ефективності вторинного навчання моделі, що вже навчилася на більшій кількості даних, використовувалась «заморозка»

шарів згорткової частини навченої моделі, тобто ці шари на навчались при вторинному навчанні моделі, а навчались тільки останні додані шари (лістинг 3.1).

Лістинг 3.1 – Заморозка і розморозка деяких шарів моделі

```
# заморозка згорткової частини та розморозка останніх шарів, що навчаються
autoencoder_in_to_out.trainable = True
for layer in autoencoder_in_to_out.layers[:-1]:
    layer.trainable = False
```

Спершу архітектуру аутоенкодера було прийнято у такому вигляді наведеному у лістингу 3.2 з різницею у кількості шарів, в залежності від вибраного розміру матриці даних [13]

Лістинг 3.2 – Перший вигляд аутоенкодера (для матриці 50x50)

```
x = layers.Conv2D(32, (kernel_size_extreme, kernel_size_extreme), activation='relu', padding='same')(encoder_input)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
encoded = layers.Conv2D(256, (3, 3), activation='relu', name='encoder')(x)

x = layers.Conv2D(256, (3, 3), activation='relu', padding='same', name='input_decoder')(encoded)
x = layers.UpSampling2D((2, 2))(x)
```

Продовження лістингу 3.2

```
    x = layers.Conv2DTranspose(128, (3, 3), activation='relu')(x)
    x = layers.UpSampling2D((2, 2))(x)
    x = layers.Conv2DTranspose(64, (3, 3), activation='relu')(x)
    x = layers.UpSampling2D((2, 2))(x)
    x = layers.Conv2D(32, (3, 3), activation='relu')(x)
    decoder_output = layers.Conv2D(10, (kernel_size_extreme, kernel_size_extreme), padding="same", activation="softmax")(x)
```

Для побудови такої мережі були прийняті наступні принципи:

– кількість шарів енкодера і декодера, по можливості, мають бути рівними, щоб ступінь компресії та декомпресії даних у мережі були, по можливості, рівними, щоб зменшити спотворення даних;

– перед збагаченням дані нормалізувались шляхом розподілу значення у кожній комірці на 9 таким чином, щоб вони були не вище 1, що зазвичай зручно для нейронної мережі;

– намагались кожні Conv2D та Conv2DTranspose шари чергувати з шарами MaxPooling2D та UpSampling2D відповідно, щоб запобігти ранньому перенавчанню;

– у якості основного активатора був прийнятий ‘relu’, оскільки його характеристика не вносить спотворення в дані з величинами від 0 до 9;

– ступінь стиску не повинна бути більше ніж до розміру матриці даних 5x5, щоб запобігти внесення значної додаткової неточності результатів за рахунок стиску;

– останній шар декодера був прийнятий з 10 фільтрами та активатором ‘softmax’, щоб передбачити ймовірності для 10 класів (значення комірок матриць від 0 до 9) [13].

Під час компіляції моделі лістинг 3.3.

Лістинг 3.3 Компіляція моделі

```
autoencoder_in_to_out.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
optimizer=keras.optimizers.Adam(learning_rate=1e-4), metrics=['sparse_categorical_accuracy'])
```

Функція втрат «sparse_categorical_crossentropy» була прийнята тому що класи вихідного шару аутоенкодера це 10 цілих чисел, а у якості метрики ['sparse_categorical_accuracy'] з тих же причин. Параметр from_logits=True прийнятий таким, позаяк активатор останнього шару мережі ‘softmax’.

У якості оптимізатора взято Adam, як такий, що швидше оптимізує зворотнє розповсюдження помилки, а швидкість навчання вибрана для уточнення при дослідженнях, але не дуже висока, щоб уникнути «викидів».

Окрім цього застосовано декілька функцій зворотнього виклику лістинг 3.4

Лістинг 3.4 – Функції зворотнього виклику

```
# Створення callbacks для поліпшення навчання
best_mod = keras.callbacks.ModelCheckpoint(filepath=model_path_save, save_best_only=True, monitor='val_loss')
early_stopping = keras.callbacks.EarlyStopping(monitor='val_sparse_categorical_accuracy', patience=5, mode='max')
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='val_loss', patience=2, factor=0.5, mode='min', min_lr=0.0000001)
```

Таким чином найкращі екземпляри моделі під час навчання зберігалися на підставі значення втрат при перевірці в зазначене місце на диску. Навчання припинялось, якщо на протязі 5 епох точність моделі не

зростала і не зменшувалася. Швидкість навчання зменшувалась на підставі значень втрат при перевірці, якщо їх величина не зменшувався на протязі двох епох, кроками по 0,5, але не менш ніж до min_lr. Ці заходи забезпечували оптимальний процес навчання моделі.

Під час навчання згаданої вище моделі використовувалося на різних етапах навчання 40, 50 зразків перетворення даних (матриць), по 64, 128 та 256 матриць у пакеті та оптимізовані розміри матриць, що подавались у модель 50x50, 60x60 у різних комбінаціях для виявлення найкращої, з кращими показниками точності та втрат.

Спочатку навчання проводилось для даних з різних ключів (файлів). Більш ніж у п’ятдесяти випробуваннях покоміркове співпадіння матриць прогнозів з оригінальними вихідними матрицями не піднялося в усередненому вигляді вище за 57% відсотків (додаток Б).

Після перших двох десятків навчань аутоенкодера стало ясно:

– кількість зразків обраного ключа вхідних даних має бути менш ніж 80, позаяк час навчання зростає вище ніж 3 доби;

– розмір ядра крайніх шарів аутоенкодера має бути 5, а не 3, позаяк точність перетворення даних при 5 зростає на 3%;

– розмір матриці даних має бути не більше ніж 80, позаяк час навчання зростає вище ніж 3 доби;

– застосування вторинного навчання на малій кількості зразків для попередньо навченої моделі на великій кількості зразків не дає позитивного результату, так як точність зменшується на 40%;

– нормалізація даних в межах від 0 до 1 не потрібна, бо зменшує точність за рахунок додаткових перерахувань на 2%;

– застосування збагачення даних за допомогою їх мозаїчного представлення дало негативний результат на зменшення точності мережі на 14% – 20%.

Результати навчання аутоенкодера, які впливали на ті чи інші дії відображені у таблицях (додаток Б, додаток В).

Для підвищення співпадіння прогнозів з оригінальними вихідними матрицями далі були задіяні декілька наступних заходів.

Більшість згорткових мереж мають пірамідальну структуру (ієрархію ознак). Глибокі ієрархії хороші тим, що сприяють повторному використанню ознак і, отже, абстрагуванню. Загалом глибокий стек шарів меншої розмірності працює краще, ніж дрібний стек шарів великої розмірності. Однак кількість шарів не може збільшуватися нескінченно через проблему згасання градієнта. Для її подолання використовується важливий архітектурний шаблон – залишкові зв'язки.

Залишкові зв'язки діють як короткі шляхи для поширення інформації в обхід деструктивних блоків або блоків, що вносять суттєві спотворення (таких як блоки з небажаними активаціями або шарами проріджування), дозволяючи інформації градієнта помилок проходити по глибокій мережі без спотворень [14] лістинг 3.5.

Лістинг 3.5 – Функції застосування залишкового зв’язку

```
def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.SeparableConv2D(filters, 3, activation="relu", padding="same", depthwise_initializer='he_normal', pointwise_initializer='he_normal')(x)
    x = layers.SeparableConv2D(filters, 3, padding="same", depthwise_initializer='he_normal', pointwise_initializer='he_normal')(x)
    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.SeparableConv2D(filters, 1, strides=2, depthwise_initializer='he_normal', pointwise_initializer='he_normal')(residual)
    elif filters != residual.shape[-1]:
```

Продовження лістингу 3.5

```
            residual = layers.SeparableConv2D(filters, 1,
depthwise_initializer='he_normal',
pointwise_initializer='he_normal')(residual)
        x = layers.add([x, residual])
        x = layers.Activation('relu')(x)
        return x
```

В функції для частини енкодера моделі спочатку у змінній residual робиться збереження для залишкового зв'язку для коду цього осередку, так як на початку форма буде як форма вхідного тензора x.shape і кількість фільтрів як у вхідного тензора x.shape[-1] (у списку по порядку ідуть висота, ширина, кількість фільтрів), тобто перший праворуч елемент списку кількість фільтрів. Наступні 2 шари SeparableConv2D зберігають форму, але активується тільки перший шар, тому що наступна активація буде після додавання залишкового зв’язку.

У випадку, якщо застосовується шар MaxPooling2D, наступний шар згортки має шаг stride=2, щоб забезпечити для залишкового зв’язку зменшений у 2 рази шаром MaxPooling2D розмір форми.

Якщо кількість фільтрів filters, що задається, не дорівнює кількості фільтрів вхідного тензора x.shape [-1], то далі застосовується згортковий шар з ядром розміром 1. Таким чином при проходженні скрізь цей шар форма не міняється, позаяк немає активації шару, тобто, просто «підганяється» форма.

Далі складається залишковий зв’язок з виходом попередніх шарів, та додається шар активації.

У функції residual_block_up для частини декодера моделі виконуються ті ж задачі, що і у попередній з відмінністю, пов’язаною з тим, що кількість фільтрів з глибиною зменшується, а не зростає, як у енкодері, а форма зростає, тому використовуються потрібні для цього шари Conv2DTranspose.

Для вирівнювання форми входу та виходу залишаються попередні шари для частини енкодера позаяк розмір ядра у них 1.

З залишковими зв'язками можна конструювати мережі довільної глибини, не переймаючись згасанням градієнтів.

Для того, щоб зробити модель легшою (з меншою кількістю вагових параметрів, що навчаються) і швидкою (з меншою кількістю операцій з числами), а також підвищити якість вирішення завдання на кілька відсотків, використовувались шари SeparableConv2D замість Conv2D [15], [16].

Цей шар виконує просторову згортку кожного каналу у вхідних даних окремо перед змішуванням вихідних каналів за допомогою крапкової згортки (згортки 1×1). Це еквівалентно роздільному виділенню просторових та канальних ознак. Подібно до того, як згортка заснована на припущенні про відсутність зв'язку закономірностей у зображеннях з певними місцезнаходженнями, роздільна згортка по глибині ґрунтується на припущенні сильної кореляції просторових місць у проміжних активаціях і практично повної незалежності різних каналів. Оскільки це припущення зазвичай правильне для зображень, що вивчаються глибокими нейронними мережами, воно служить корисною попередньою умовою, яка допомагає моделі ефективніше використовувати навчальні дані.

В результаті виходять моделі меншого розміру, які сходяться швидше і менш схильні до перенавчання. Ці переваги є особливо важливими при навчанні невеликих моделей з нуля на обмеженому наборі даних, тобто, як раз це справедливо для нашого випадку [16].

Після застосування вищенаведених нововведень до першого варіанту моделі, отримали наступний код (лістинг 3.6), наприклад для матриці даних з формою (60, 60).

У даному коді моделі видно, що всі шари, окрім перших та останніх у енкодері та декодері із застосуванням залишкового зв’язку, що дозволяє впевнено використовувати модель з великою кількістю шарів з причин, названих вище.

Лістинг 3.6 – Модель із застосуванням залишкових зв’язків

```
    # Encoder
            x           =           layers.SeparableConv2D(32,
(kernel_size_extreme, kernel_size_extreme), activation='relu',
padding='same',              depthwise_initializer='he_normal',
pointwise_initializer='he_normal')(encoder_input)
            x = residual_block(x, filters=32, pooling=True)
            x = residual_block(x, filters=64, pooling=True)
            x = residual_block(x, filters=128, pooling=True)
            encoded   =   layers.SeparableConv2D(256,   (3,   3),
activation='relu',                                name='encoder',
depthwise_initializer='he_normal',
pointwise_initializer='he_normal')(x)

            # Decoder
            x    =    layers.SeparableConv2D(256,    (3,    3),
activation='relu',                          name='input_decoder',
depthwise_initializer='he_normal',
pointwise_initializer='he_normal')(encoded)
            x = residual_block_up(x, filters=256, pooling=True)
            x = residual_block_up(x, filters=128, pooling=True)
            x = residual_block_up(x, filters=64, pooling=True)
            x = residual_block_up(x, filters=32, pooling=True)
            decoder_output     =     layers.SeparableConv2D(10,
(kernel_size_extreme, kernel_size_extreme))(x)
```

Треба сказати, що для ініціалізації ваг примусово використовувалася ініціалізація типу не нормального розподілу 'he_normal' це спеціалізований підхід для нейронних мереж із функціями активації ReLU.

Ця ініціалізація враховує, що ReLU «відсікає» половину активацій (від'ємні значення), тому вона компенсує це, збільшуючи дисперсію ваг у порівнянні з іншими типами, допомагає уникнути проблеми зникаючих градієнтів на ранніх етапах, оскільки зберігає дисперсію активацій між шарами, мережі з нею часто досягають нижчих значень

функції втрат вже на початкових епохах в порівнянні з іншими типами ініціалізації.

3.3 Отримання та підготовка даних для роботи з мережею

3.3.1 Загрузка даних з файлів ARC-AGI

Бажано, щоб усі входи та виходи (цілі) в нейронній мережі були тензорами чисел з плаваючою точкою (або в особливих випадках тензорами цілих чисел). Які б дані не було б потрібно обробити – звук, зображення, текст, – їх спочатку потрібно перетворити на тензори. Цей крок називається векторизацією даних. Векторизацію виконано з перетворенням даних у тип floar16 для економії пам’яті комп’ютера, виходячи з принципу «необхідно та достатньо».

Як вже вказувалося раніше, нормалізація даних в межах від 0 до 1 не потрібна, бо зменшує точність за рахунок додаткових перерахувань на 2%. Але такий стан справ підходить, тому що функція активації ReLU, що використовується, добре працює як з даними в межах 0-1, так і в межах 0-9, завдяки своїй формі. Таким чином отримані дані було зроблено однорідними, що бажано для роботи в нейронній мережі.

Зазвичай, при визначенні методів конструювання зразків та виборі архітектури моделі, які з великою ймовірністю підійдуть для вирішення завдання перед мережею, ставлять дві гіпотези:

– гіпотезу про те, що вихідні дані можна передбачити за вхідними даними (в нашому випадку це не гіпотеза, а аксіома);

– гіпотезу у тому, що доступні дані досить інформативні для вивчення відносин між вхідними і вихідними даними.

Цілком можливо, що ця друга гіпотеза помилкова, і тоді доведеться виконати певні дії з даними, щоб досягти необхідної їх інформативності.

Отже, початок роботи з даними це початок роботи програми (додаток А) після блоку імпортів та ініціалізації глобальних параметрів, який коментований як «# отримання та перетворення даних».

В залежності від наявності файлу даних, створюється новий пустий файл, або використовується наявний.

Інформація з файлів ARC-AGI [17], [18] завантажується в список, далі перемішується та розподіляється на дані для первинного та вторинного навчання. В залежності від кількості контрольних зразків (матриць), створюються списки для тренування та контролю навченої моделі [19]. Потім ці списки додаються в словники по ключам, які зберігаються у файл, стосовно первинного навчання, на диск.

3.3.2 Розрахунок розміру матриць (даних) для моделі

Оскільки розміри матриць (даних), навіть, у одному файлі (ключі) можуть бути різними, а для роботи мережі потрібно, щоб вхідна матриця була квадратною, необхідно дослідити розміри всіх наявних матриць та визначити максимальний розмір, щоб потім матриці, саме з таким розміром, використовувалися у навчанні нейронної мережі.

Наступний блок коду з коментарем «# розрахунок розміру матриць для моделі» саме і займається вирішенням цієї задачі.

Якщо файл даних поточного ключа відсутній, то в залежності від файлу (ключа), з якого взяті ARC-AGI дані, вони розподіляються на навчальні (train), перевірочні, або контрольні (test), а ті в свою чергу на вхідні (input) та вихідні (output) [20].

В циклах по ключам та по матрицям даних одного ключа перебираються вхідні та вихідні матриці (вихідні дані можуть бути різного розміру з вхідними) тільки навчальні, тому що контрольні мають однакові розміри з навчальними. Далі максимальні розміри цих матриць по ключам,

а також взагалі, пакуються у кортежі, а ті в словник, який записується у файл на диску.

У випадку ж наявності файлу даних ключа, просто завантажується збережений файл зі словником максимальних розмірів та його словник та кортежі розпаковуються для встановлення потрібних змінних у програму.

Ще під час завантаження даних з файлів ARC-AGI, був створений список всіх ключів (файлів). Тепер перебір по всім ключам дозволяє вибирати перший елемент кортежу, тобто максимальний розмір (форму) даних для поточного ключа і брати з цього розміру (теж кортежу) потрібний елемент 0, якщо це розмір по вертикалі, або 1, якщо це розмір по горизонталі, і, відповідно, знаходити різницю між прийнятим розміром матриць (даних) у мережі та одним з перелічених розмірів, щоб розуміти скільки кроків зміщення для кожного ключа буде у вкладеної в більшу матрицю для мережі.

Таким чином, для кожного ключа ми розраховуємо ступінь збагачення за рахунок зсуву по вертикалі та по горизонталі максимально розмірної матриці поточного ключа, що вкладена в матрицю мережі (яка має розміри рівні, або більші, ніж вкладені матриці). Таку матрицю мережі на кожному крокові зсувів можна вважати як окремий зразок навчальної матриці для мережі.

Далі в консолі виводиться інформація про проведені розрахунки.

3.3.3 Формування даних та запис у файл *.h5 після збагачення

Оскільки для економії пам’яті вирішено зберігати дані для мережі у файлі типу *.h5, то необхідно формувати ці дані за допомогою пакетів, що формуються у буферах даних, а по заповненні останніх, записуються у файл.

Ці дії виконує блок коду програми, що закоментований як «# формування та збагачення даних».

Цей код перебирає дані по ключам, по значенням яких відкривається відповідний файл з структурованими даними, під час завантаження даних з файлів ARC-AGI відповідно до підрозділу 3.3.1, раніше завантажений на диск.

Інформація із цього файлу розпаковується у словники та списки, які завдяки ітераціям досліджуються та формуються у 2D масиви (матриці).

Далі матриці можуть бути для збагачення перетворені у мозаїку, завдяки присвоєнню змінній k_mozaik певного значення 2, 3 і т. ін.. Цю операцію виконує функція на лістингу 3.7.

Лістинг 3.7 – Функція створення мозаїки з матриці

```
def mozaik(matrix, koef):
    """
    Генерує мозаїку з матриці, що задана з
    коефіцієнтом множення мозаїки
    """
    for h in range(koef):
        if h == 0:
            h_kort_h = matrix
        if h > 0:
            h_kort_h = np.hstack((h_kort_h, matrix))
    for v in range(koef):
        if v == 0:
            v_kort_v = h_kort_h
        if v > 0:
            v_kort_v = np.vstack((v_kort_v, h_kort_h))
    return v_kort_v
```

Як видно, завдяки бібліотеці Numpy, мозаїка складається в стик по горизонталі та по вертикалі з блоків тотожних матриці, що обробляється. Функція повертає мозаїку вхідної матриці.

Далі вхідні та вихідні матриці вкладаються у матрицю, з якою буде працювати мережа, тобто у більшу матрицю (де після вкладання значення

не зайнятих комірок будуть дорівняні 0), всередині якої, по її вісям, вкладені матриці будуть переміщуватися кроками циклу при збагаченні, створюючи на кожному кроці зразок збагачених даних.

Наступним кроком є збагачення даних за рахунок поворотів та віддзеркалення матриць (лістинг 3.8), яке можливо, позаяк у згорткових мережах завдяки принципу інваріантності згорток, зчитувані елементи даних з регіонів матриць не залежать від їх розташування відповідно вісей матриці. Таким чином «перемішані» зчитані елементи однієї оригінальної матриці будуть однаково сприйняті згортковою мережею, але з точки зору мережі у різних матрицях, тобто у різних зразках вхідних даних.

Лістинг 3.8 – Функція трансформації матриць

```
def generate_transformations(matrix_in, matrix_out):
    """
    Генерує всі унікальні трансформації (повороти та відображення) для квадратної матриці.
    :param matrix: вхідна квадратна матриця (numpy array)
    :return: список унікальних матриць
    """
    transformations_in = []
    transformations_out = []
    # Додаємо повороти на 0°, 90°, 180°, 270°
    for i in range(4):
        rotated = np.rot90(matrix_in, k=i)
        transformations_in.append(rotated)
        rotated = np.rot90(matrix_out, k=i)
        transformations_out.append(rotated)

    # Додаємо відображення та їх повороти
    flipped_in = np.flipud(matrix_in)   # Відображення по горизонталі
    flipped_out = np.flipud(matrix_out)  # Відображення по горизонталі
```

Продовження лістингу 3.8

```
        for i in range(4):
            transformations_in.append(np.rot90(flipped_in,
k=i))
            transformations_out.append(np.rot90(flipped_out,
k=i))
        # Додаємо відображення та їх повороти
        flipped_in = np.fliplr(matrix_in)  # Відображення по
вертикалі
        flipped_out = np.fliplr(matrix_out)  # Відображення по
вертикалі
        for i in range(4):
            transformations_in.append(np.rot90(flipped_in,
k=i))
            transformations_out.append(np.rot90(flipped_out,
k=i))

        # Фільтруємо лише унікальні матриці відносно вхідних а
вихідні
        # беремо за індексами вхідних унікальних
        unique_transformations_in = []
        unique_transformations_out = []
        unique_ind = []

        for ind, t in enumerate(transformations_in):
            if not any(np.array_equal(t, u) for u in
unique_transformations_in):
                unique_transformations_in.append(t)
                unique_ind.append(ind)

        for ind in unique_ind:

unique_transformations_out.append(transformations_out[ind])
        return                    unique_transformations_in,
unique_transformations_out
```

Спочатку створюються списки трансформованих вхідних та вихідних матриць, адже їх потрібно синхронно трансформувати. Потім ці матриці повертаються по 90 градусів на 360 градусів з додаванням у створені списки.

Потім матриці віддзеркалюються в різні боки та знову обертаються на 360 градусів по 90 градусів і знов додаються до згаданих списків.

Коли списки готові, створюються нові списки для збереження унікальних індексів матриць, що трансформувалися, а по ним (по матрицям) у циклі з умовою унікальності вибираються відповідні індекси для вхідних матриць. Далі по цим індексам вибираються унікальні вихідні матриці.

Функція повертає списки унікальних синхронно трансформованих вхідних та вихідних матриць.

Нарешті готуються буфери для накопичення даних та запису їх з економією часу у файл даних, який відкривається, створюються dataset, як формат для роботи з мережею у середовищі keras, у циклах перебору даних по кількості трансформованих матриць, кількості переміщень по горизонталі та по вертикалі при заповненні буферів даними, виконується запис їх у файл.

Прогрес виконання цієї роботи відображається прогрес баром бібліотеки tqdm. Оновлення прогрес бару кратно 1000, тому що, особливо, у інтернет ресурсах Kaggle та у Google Colaboratory неможливе швидке оновлення цього елементу між клієнтом та сервером.

В кінці циклів тренувального набору звільняється пам’ять і далі всі описані вище в цьому підрозділі процедури виконуються з набором test, з винятком дій по збагаченню даних, тому що з цими даними ці дії не потрібні, позаяк контрольні дані потрібні лише для прогнозування, вже навченою моделлю.

Після умовного оператора, що забезпечує правильний вибір розмірів матриць, в кінці цього блоку коду звільняється пам’ять.

3.3.4 Формування даних у форматі dataset та запис у файл *.h5

У підрозділі 3.3.3 йшлося про запис даних у файл даних в датасети з найменуваннями «x» та «y», але ж для нейронної мережі в середовищі keras потрібно мати датасети 'x_train', 'y_train', 'x_val', 'y_val', позаяк мережа на одних датасетах ('x_train', 'y_train') даних вчиться, а потім на других('x_val', 'y_val') перевіряє своє навчання.

То ж далі є наступний блок коду, який має один з коментарів «# підготовка наборів даних для моделі», який переформатує дані та необхідні датасети запише у файл даних для роботи з мережею.

Якщо файла даних ще немає, тобто він ще не заповнений даними у необхідному форматі для роботи з мережею, то цей файл відкривається для запису даних. Для економії пам’яті узнаємо розміри датасетів без завантаження даних, генеруємо перемішані індекси даних, та встановлюємо роздільний індекс для навчального та перевірочного наборів даних.

Далі видаляємо старі датасети, при їх наявності, створюємо нові датасети для запису в них сформованих даних для мережі. Сортуємо індексі перед читанням даних із файлу та визначаємо розмір пакету даних для запису у файл попакетно, щоб пришвидшити процес та зменшити навантаження на пам’ять.

Далі в циклах пакетами записуємо навчальні та перевірочні дані згідно розділених раніше індексів, а потім видаляємо старі, непотрібні вже датасети, чистимо пам’ять, та встановлюємо флаг наявності файлу даних.

Наразі можна сказати, що дані завантажені, сформовані та підготовлені для подальшої роботи з нейронною мережею.

3.3.5 Процес створення, навчання моделі та отримання прогнозу

Наступний блок коду має коментар «# Безпосередня робота з моделлю». Цей код створює або завантажує готову модель нейронної

мережі, яка потім проходить процес навчання та зберігання. Навчена збережена модель використовується для прогнозування вихідної матриці для вхідної контрольної, або декількох таких, в залежності від початкових установок.

Спочатку задається шлях до збереженої моделі, тобто шлях де вона буде зберігатися. Далі згідно шаблону проводиться пошук існуючої моделі за допомогою спеціальної функції (лістинг 3.9).

Лістинг 3.9 Функція пошуку існуючої моделі

```
def search_model(patt):
    """
    Пошук файлів моделі згідно шаблону та їх розбір згідно регулярного виразу
    """

    # Повний шлях для пошуку згідно шаблону
    search_path = os.path.join(dir_save, patt)

    # Пошук файлів згідно шаблону
    matching_files = glob.glob(search_path)
    # немає ніяких файлів
    if len(matching_files) == 0:
        return [False, in_shape[0], count_matrix_train_first, batch_size, count_epoch]
    # Регулярний вираз для вилучення значень з назви файлу
    regex = re.compile(fr"Autoencod_test_in_to_out_(\d+)_(\d+)_(\d+)_(\d+)_{num_steps}_FIRST\.keras")
    # список кількостей епох
    arr_epochs = []
    # список параметрів моделей
    arr_files = []
    # Перебір знайдених файлів
    for file_path in matching_files:
```

Продовження лістингу 3.9

```
            # вилучення імені файлу з повного шляху
            file_name = os.path.basename(file_path)

            # вилучення значень з назви файлу
            match = regex.match(file_name)
            # якщо є файли
            if match:
                file_in_shape = int(match.group(1))
                file_count_matrix_train_first =
int(match.group(2))
                file_batch_size = int(match.group(3))
                file_count_epoch = int(match.group(4))
                # якщо є файли потрібні на даному
етапі(співпадають задані та наявні елементи)
                if file_in_shape == in_shape[0] and
file_count_matrix_train_first == count_matrix_train_first and
file_batch_size == batch_size:
                    # заповнення списків параметрів
                    arr_files.append([file_in_shape,
file_count_matrix_train_first, file_batch_size,
file_count_epoch])
                    arr_epochs.append(file_count_epoch)
        if len(arr_files) > 0:
            # індекс максимального значення
            index_max = np.argmax(arr_epochs)
            file_max = arr_files[index_max]
            # параметри для збереження існуючої моделі
            return [True, file_max[0], file_max[1],
file_max[2], file_max[3]]
        else:
            # параметри для створення нової моделі у разі
відсутності збереженої
            return [False, in_shape[0],
count_matrix_train_first, batch_size, count_epoch]
```

Ця функція спочатку задає повний шлях збереження моделі, щоб потім, згідно шаблону, знайти всі файли, що відповідають шаблону по цьому шляху.

Якщо за вказаним шляхом відсутні файли згідно шаблону, функція повертає список з 5 елементів, де перший False сповіщає про те, що шукана модель відсутня, другий елемент вказує головний параметр для створення моделі, тобто, розмір вхідної матриці моделі, третій елемент потрібен для влаштування у назву моделі для її однозначної ідентифікації, четвертий елемент вказує прийнятий розмір пакета даних навчання та п’ятий елемент вказує кількість епох навчання. Взагалі, якщо перший елемент списку, що повертає ця функція дорівнює False, то це говорить про те, що шукана модель відсутня.

А коли будуть знайдені файли згідно пошукового шаблону, створюється регулярний вираз для вилучення значень з назви файлу та пусті списки для зберігання кількостей епох, та елементів списків, що повертаються функцією, окрім першого. Далі у циклі перебираються всі знайдені файли та з їх назв за допомогою сформованого регулярного виразу зчитуються дані для заповнення створених попередньо списків.

Далі зі списку кількості епох береться індекс максимального значення, а, оскільки ці два списки створювалися та заповнювалися синхронно, то за індексом максимального значення у списку кількості епох, зчитуємо значення елементів, що повертаються функцією.

Отже функція повертає список, де першим елементом буде True (модель знайдена), далі елементи, з яких складається назва моделі, і головне, кількість епох. Останній елемент вказує на те, що остання, потрібна для поточної роботи з мережею, створена та навчена модель навчалась впродовж даної кількості епох, а це означає, що наступного разу модель буде навчатися не з 0, а з цієї вказаної кількості епох, до якої буде додаватися задана кількість епох наступного разу, тобто модель буде донавчатися.

На підставі інформації, що повернула згадана вище функція, умовний оператор вибирає ті чи інші інформаційні повідомлення користувачу, завантажує або створює модель та встановлює величини деяких змінних.

Якщо це вторинне навчання, то встановлюються шари, які будуть навчатися та в будь якому разі, встановлюються функції зворотнього виклику.

Після цього йдуть дві дуже схожі частини коду. В них функції генератори для створення періодичного потоку даних пакетами, розмір яких розраховується окремо, для подачі в модель при навчанні та при перевірці у вигляді потрібних форматів датасетів середовища Keras.

Далі йдуть інформаційні повідомлення про навчання, розміри даних, припустиму назву файлу для збереження лістингу консолі, саме навчання моделі з роздрукуванням у консолі короткої інформації про параметри навчання.

В кінці навчання, після перевірки моделі розраховуються мінімальна втрата та максимальна точність найкращої моделі, що буде збережена по команді функції зворотнього виклику, та завантажена для виконання прогнозів від контрольних вхідних матриць.

Остання частина коду з коментарем «# цикл по контрольним матрицям з прогнозуванням» виконує прогнозування контрольного виходу, або виходів, виконує за допомогою бібліотеки matplotlib графічну візуалізацію процесу навчання та виводить зображення вхідних, вихідних та прогнозованих матриць засобами бібліотеки Cub для більшої наглядності.

Оскільки контрольних вхідних матриць, взагалі, може бути декілька, то створюється цикл по ним, де виводиться прогноз.

Оскільки вихід моделі ймовірнісний, тобто, кожна комірка по 10 каналам має значення ймовірності, то треба визначити індекси каналів, де ймовірність максимальна.

Значення цих індексів і будуть оригінальними значеннями комірок матриці прогнозу.

```
    # переформатування зі значеннями індексів
    y_pred = tf.argmax(y_pred, axis=-1, output_type=tf.int32)
```

Далі прогноз переформатується у матрицю для зручної візуалізації і якщо матриця була мозаїкою, то треба повернути її в оригінальний вигляд вихідної матриці для моделі за допомогою спеціальної функції (лістинг 3.10).

Лістинг 3.10 – Функція перетворення мозаїки у оригінал

```
def transform_matrix(matrix, k_mozaik, kernel_size):
    """
    Копіює внутрішні кордони між елементами мозаїки та переставляє їх в зовнішні для лівої верхньої матриці елементу мозаїки, а потім виділяє ліву верхню матрицю з мозаїки
    """
    if k_mozaik != 1:
        a, b = matrix.shape[0] // k_mozaik, matrix.shape[1] // k_mozaik
        lenmerge = 1
        if kernel_size == 3:
            lenmerge = 1
        elif kernel_size == 5:
            lenmerge = 2
        # Копіюємо комірки стовпчиків
        copied_columns = matrix[a:k_mozaik*a-lenmerge, b:b+lenmerge].copy()
        matrix[:copied_columns.shape[0], :lenmerge] = copied_columns
        # Копіюємо комірки рядків
        copied_rows = matrix[a:a+lenmerge, b:k_mozaik*b-lenmerge].copy()
        matrix[:lenmerge, :copied_rows.shape[1]] = copied_rows
        # виділяємо верхню матрицю ліворуч у мозаїці
```

Продовження лістингу 3.10

```
            matrix   =   matrix[:int(matrix.shape[0]/k_mozaik),
:int(matrix.shape[1]/k_mozaik)]
        return matrix
```

Ця функція копіює внутрішні кордони між елементами мозаїки та переставляє їх в зовнішні для лівої верхньої матриці елементу мозаїки, а потім виділяє ліву верхню матрицю з мозаїки. Це робиться для того, щоб позбавитися межового ефекту під час згортки, коли ядро виходить за зовнішні межі матриць. Функція вибирає такі самі межи, але вже не зовнішні, що можливо в мозаїці, та копіює їх замість аналогічних, але зовнішніх, з межовим ефектом згортки.

В залежності від розміру ядра згортки вибирається розмір копії межі. Межі копіюються та вставляються замість певних зовнішніх, а потім виділяється оновлена матриця ліворуч верхня для подальшого перетворення в оригінал.

Матриця контрольного входу теж переформатується до 2D формату і виділяється з мозаїки за потреби.

Зазвичай матриці для роботи з моделлю більше розміром ніж оригінальні вхідні та вихідні матриці, тому далі їм повертаються оригінальні розміри за допомогою простої функції та заздалегідь встановлених змінних (лістинг 3.11).

Лістинг 3.11 Повернення оригінальної форми контрольних матриць

```
def matrix_to_submit(matrix_in, matrix_shape, out=True):
    """Виділення матриці оригінального розміру"""
    if out:
        matrix_in      =      matrix_in[:matrix_shape[0],
:matrix_shape[1]]

    return matrix_in
```

Як видно, по заданій формі виділяється оригінальна матриця шляхом зрізу по розмірам елементів форми. Отримана після зрізу матриця повертається функцією.

Нарешті виводяться зображення контрольної вхідної матриці, матриці прогнозу, та оригінальної матриці вихідної. Далі виконується розрахунок покоміркового співпадіння оригіналу вихідної матриці з матрицею прогнозом. Після чого звільняється пам’ять.

3.3.6 Отримання перетворених даних, їх аналіз, подальші рішення

Як вже було сказано в підрозділі 3.3.1, потрібно бути впевненим в справедливості гіпотези про те, що «доступні дані досить інформативні для вивчення відносин між вхідними і вихідними даними.

Цілком можливо, що ця гіпотеза помилкова, і тоді доведеться виконати певні дії з даними, щоб досягти необхідної їх інформативності» [21].

Після того, як програма дослідження мережі готова та, навіть, запущена і почала працювати, видаючи на виході певні дані, питання для вирішення поставленої задачі, що вирішується не закінчуються, а, навпаки, повстають нові, які потрібно вирішити в процесі навчання моделі та аналізу отриманих після цього даних.

Чи достатньо шарів і параметрів, щоб правильно змоделювати задачу?

Основна проблема машинного навчання – протиріччя між оптимізацією та узагальненням; ідеальною вважається модель, яка стоїть безпосередньо на межі між недонавчанням та перенавчанням, між недостатньою та надмірною ємністю. Де пролягає цей кордон? Щоб його знайти, його треба перетнути.

Для оцінки того, наскільки великою має бути модель, спочатку маємо сконструювати модель з ефектом перенавчання. Зроблено це було легко:

– додано шарів;

– вказано велику кількість параметрів у шарах;

– навчено модель на великій кількості епох.

Постійно вівся контроль за тим, як змінювався рівень втрат на етапах навчання та перевірки, а також як змінювалися будь-які інші показники на тих же етапах, які були цікаві. Погіршення якості моделі на перевірочних даних свідчило про досягнення ефекту перенавчання [2], [12].

Коли було досягнуто перенавчання моделі, основною метою стала максимізація узагальненості.

Цей етап зайняв найбільше за все часу: довелося багаторазово змінювати модель, навчати її, оцінювати якість на перевірочних даних (контрольні дані не брали тут ніякої участі), знову доводилося змінювати її та повторювати цикл, доки якість моделі не досягала бажаного рівня.

Ось дещо з того, що робилося:

– додавалося проріджування;

– випробувалися різні архітектури, додавалися та видалялися шари;

– випробувалися різні гіперпараметри (підрозділ 3.1), щоб знайти оптимальні налаштування.

Щоразу, коли використовувався зворотний зв'язок із процесу перевірки для налаштування моделі, відбувався витік інформації в модель. Цикл повторювався лише кілька разів, щоб це не призводило до перенавчання моделі на перевірочних даних (навіть при тому, що модель безпосередньо не отримувала їх). Так вчинялося, щоб не знижувати надійність процесу оцінки [2].

Після отримання задовільної конфігурації, було навчено остаточний варіант моделі на всіх доступних даних (навчальних і перевірочних) і оцінено її якість на контрольному наборі. Якщо якість моделі на контрольних даних виявлялася значно гіршою, ніж на перевірочних, це означало, що процедура перевірки була ненадійною або в процесі налаштування параметрів моделі був ефект перенавчання на перевірочних

даних. Процес налаштування в такому разі повторювався, але вже з початку, щоб уникнути зазначеного у попередньому абзаці витоку інформації в модель.

Тут не будуть представлені інформаційні повідомлення програми у консолі з відповідними даними візуалізації процесу навчання, перевірки та результатів прогнозу на контрольних даних для скорочення обсягу пояснювальної записки.

Варто зазначити, що програма, дещо, редагувалася у дослідницькому процесі, тому повідомлення у консолі можуть частково відрізнятися, в залежності від версії програми, що використовувалася.

Як вказувалося у підрозділах 2.1.1, 2.2.4, 3.3.3, дослідження проводились як з використанням IDE Spyder платформи Anaconda, так і з використанням інтернет ресурсів Kaggle та Google Colaboratory, тому деякі повідомлення та візуалізації взяті з консолей різних ресурсів, можуть дещо розрізнюватися зовні.

Оскільки дослідницьких даних дуже багато (додаток Б, додаток В), щоб не засмічувати роботу великою кількістю малоінформативного тексту, будуть надані деякі прикінцеві та кінцеві результати.

Результати кодуються наступним чином X_Y_Z_Q, де:

– X – це розмір вхідної матриці моделі;

– Y – це кількість зразків матриць для тренування поточного ключа;

– Z – це розмір пакету навчання;

– Q – це кількість епох навчання.

Цей код використовується як назва відповідних рисунків або лістингів.

Для економії часу навчання та перевірка моделі проводилась етапами по 16 епох, де після кожного етапу зберігалась модель і для наступного етапу завантажувалася збережена та донавчалась наступні 16 епох, щоб не починати навчання з 0.

Кінцеві та прикінцеві результати з візуалізацією (додаток В) – це результати, відповідно, з останньою та попередньою архітектурами моделі, описаними у підрозділі 3.2.

Повні результати дослідницької роботи не будуть представлені тут для скорочення обсягу пояснювальної записки, як вже було зазначено вище.

В цьому підрозділі розміщена тільки візуалізація вхідних, вихідних даних та відповідних прогнозів при застосуванні різних гіперпараметрів у досліджуваній мережі. Вхідні та вихідні дані рівномірно взяті з тренувального набору зразків у трьох екземплярах для усереднення. До рисунків додані основні параметри, що досліджувались.

В наведених результатах дослідження досягнуто покоміркову точність прогнозів з оригіналами більш ніж у 90 відсотків.

Тезис, викладений у абзаці 5 підрозділу 3.3.6 «Основна проблема машинного навчання – протиріччя між оптимізацією та узагальненням; ідеальною вважається модель, яка стоїть безпосередньо на межі між недонавчанням та перенавчанням, між недостатньою та надмірною ємністю. Де пролягає цей кордон? Щоб його знайти, його треба перетнути», наглядно виявився на вище наданих рисунках.

Зростання кількості зразків для навчання веде до підвищення точності (рисунки 3.2 та 3.4), але до певної межі, після якої зростання кількості зразків веде до перенавчання та зменшує точність (рисунки 3.8, 3.9 та 3.6, 3.7). Зростання кількості епох навчання теж підвищує точність моделі до певної міри (рисунки 3.4 та 3.5), але потім починається перенавчання і точність падає (рисунки 3.11 та 3.13). Таким чином і перетинається згаданий у попередньому абзаці кордон, щоб наблизити модель до ідеальної.

Лише зміна архітектури моделі, у будь якому випадку, позитивно впливає на точність (рисунки 3.1 та 3.2, 3.3 та 3.4 і т. ін. для усіх прикінцевих та кінцевих пар).

ВИСНОВКИ

Завдання на кваліфікаційну роботу виконано повністю. Завдяки застосуванню оригінального варіанту TTT (тонкого налаштування під час тестування), у даній роботі з пошуком у прихованому (латентному) просторі моделі завдяки використанню аутоенкодеру на базі згорткової мережі. Створено систему ШІ здатну ефективно набувати нових навичок незалежно від даних навчання, шляхом створення архітектури нейронної мережі та її навчання.

Було виконано випадковий пошук та градієнтний спуск для пошуку кращих уявлень програм у прихованому просторі моделі на рівні стислих даних у енкодері, що є новим підходом до адаптації під час тестування, який не є ні тонким налаштуванням, ні дискретним пошуком.

Досягнуто покоміркову точність співпадіння матриць прогнозів перетворення вхідних зразків з матрицями оригіналами понад 90% при використанні впродовж навчання трансдуктивної моделі в якості тренувальних даних 40 та 50 матриць дуже гарний результат. Такої кількості даних у навчальному наборі аж ніяк недостатньо для досягнення подібних результатів при застосуванні класичних методів навчання інших нейронних мереж для виконання трансдуктивних задач.

У результаті роботи проведено теоретичний аналіз архітектур нейронних мереж, методи їх оптимізації, а також альтернативні методи вирішення поставленої задачі та метрики якості для їх оцінювання. Для оптимізації параметрів нейронної мережі було проаналізовано й використано існуючі набори даних ARC-AGI.

Цей напрямок роботи з нейронними мережами з метою досягти справжнього AGI ще дуже молодий і недостатньо розвіданий. Для досягнення більшої впевненості на початку дослідницької роботи в даному напрямку, було прийнято кількість тренувальних матриць в декілька десятків, щоб скоріше зрозуміти правильність вибору методу. Справа в

тому, що з більшою кількістю вхідних даних ймовірніше швидше отримати тренд подальшого розвитку вирішення задач ARC–AGI. Якщо одразу починати з дуже малої кількості вхідних даних, то опиняєшся у невизначеності – варто чи не варто продовжувати, тому що немає ще твердих критеріїв вибору тренду (головного напрямку) в вирішенні поставленої задачі, що зазначено в підрозділі 1.2.

Завдяки вибраній тактиці та отриманим результатам можна твердо сказати, що задача буде вирішена і з більш жорсткими умовами на кількість тренувальних даних через невеликий проміжок часу.

В майбутньому потрібно і далі удосконалювати архітектуру моделі та винаходити нові методи збагачення тренувальних даних.

Ця робота підтвердила можливість використання аналогій з біологічним світом і когнитивним розвитком всього живого, особливо, людини. Адже дитина, як і все живе, знайомиться з оточенням ретельно досліджуючи незнайомі сутності всіма органами почуттів, отримуючи багатогранну гамму сприйняттів. Що це, як не збагачені дані надані трансдуктивній моделі для прийняття рішення?

Для доступності в майбутньому наслідків досягнення AGI, треба весь час пам’ятати про ефективність того чи іншого методу при вирішенні цієї задачі, щоб користатися AGI могли звичайні люди без наявності суперкомп’ютерів та гігантських баз даних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. On the Measure of Intelligence. *arXiv*. URL: https://arxiv.org/abs/1911.01547 (date of access: 04.03.2025).

2. Chollet F. Deep learning with python. Manning Publications Co. LLC, 2017. 384 p.

3. Deitel H., Deitel P. Intro to python for computer science and data science: learning to program with AI, big data and the cloud. Pearson Education, Limited, 2019.

4. Raj B. Data augmentation | how to use deep learning when you have limited data – part 2. *Medium*. URL: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced (date of access: 04.03.2025).

5. Jain S. NanoNets : how to use deep learning when you have limited data. *Medium*. URL: https://medium.com/nanonets/nanonets-how-to-use-deep-learning-when-you-have-limited-data-f68c0b512cab (date of access: 04.03.2025).

6. Turing A. M. Computing machinery and intelligence. *Brain physiology and psychology*. 2023. P. 213–241. URL: https://doi.org/10.2307/jj.8501509.16 (date of access: 04.03.2025).

7. Francois Chollet, Mike Knoop, Bryan Landers, Greg Kamradt, Hansueli Jud, Walter Reade, and Addison Howard. Arc prize 2024. URL: https://kaggle.com/competitions/arc-prize-2024 (date of access: 04.03.2025).

8. ARC prize 2024: technical report. *arXiv.org e-Print archive*. URL: https://arxiv.org/html/2412.04604v2 (date of access: 08.04.2025).

9. Abstraction and reasoning corpus for artificial general intelligence v1 (ARC-AGI-1). *github*. URL: https://github.com/fchollet/ARC-AGI (date of access: 08.04.2025).

10. Conservativeness of untied auto-encoders. *arXiv.org*. URL: https://arxiv.org/abs/1506.07643 (date of access: 04.03.2025).

11. Alain, G., and Bengio, Y. 2014. What regularized auto-encoders learn from the data generating distribution. In Internaton Conference on Learning Representations.

12. LeCun Y., Bengio Y., Hinton G. Deep learning. *Nature*. 2015. Vol. 521, no. 7553. P. 436–444.

13. Building autoencoders in keras. *The Keras Blog*. URL: https://blog.keras.io/building-autoencoders-in-keras.html (date of access: 04.03.2025).

14. Deep residual learning for image recognition. *arXiv.org*. URL: https://arxiv.org/abs/1512.03385 (date of access: 04.03.2025).

15. Xception: deep learning with depthwise separable convolutions. *arXiv.org*. URL: https://arxiv.org/abs/1610.02357 (date of access: 04.03.2025).

16. Encoder-Decoder with atrous separable convolution for semantic image segmentation. *arXiv.org*. URL: https://arxiv.org/abs/1802.02611 (date of access: 04.03.2025).

17. GitHub – michaelhodel/re-arc: reverse engineering the abstraction and reasoning corpus. *GitHub*. URL: https://github.com/michaelhodel/re-arc (date of access: 04.03.2025).

18. GitHub – neoneye/arc-dataset-tama: Big ARC tasks compatible with the Abstraction and Reasoning Corpus json file format. *GitHub*. URL: https://github.com/neoneye/arc-dataset-tama (date of access: 04.03.2025).

19. GitHub – KSB21ST/MINI-ARC. *GitHub*. URL: https://github.com/ksb21ST/Mini-ARC (date of access: 04.03.2025).

20. Extra ARC tasks for testing. *Kaggle*. URL: https://www.kaggle.com/datasets/andypenrose/extra-arc-tasks-for-testing (date of access: 04.03.2025).

21. Auto-Encoding variational bayes. *arXiv*. URL: https://arxiv.org/abs/1312.6114 (date of access: 04.03.2025).