НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

ТАРАН ВЛАДИСЛАВ ІГОРОВИЧ

УДК 004.032.26 (043.3)

ДИСЕРТАЦІЯ

МЕТОД АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ ДО АПАРАТНОГО ЗАБЕЗПЕЧЕННЯ ЗІ СПЕЦІАЛІЗОВАНОЮ АРХІТЕКТУРОЮ

123 Комп’ютерна інженерія
12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело_____

Науковий керівник: Гордієнко Юрій Григорович, д. ф.-м. н., професор

Київ – 2022

АНОТАЦІЯ

Таран В.І. Метод адаптації глибоких нейронних мереж до апаратного забезпечення зі спеціалізованою архітектурою. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 123 – Комп’ютерна інженерія з галузі знань 12 – Інформаційні технології. – Національний Технічний Університет України «Київський Політехнічний Інститут імені Ігоря Сікорського», Київ, 2022.

Дисертаційна робота присвячена розробці комплексного методу адаптації глибоких нейронних мереж, що дозволяє підвищити продуктивність та ефективність обробки даних глибокими нейронними мережами на апаратному забезпеченні зі спеціалізованою архітектурою.

Вперше було розроблено комплексний метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур.

Розроблено метод адаптивного ітеративного прунінгу для зменшення розміру моделей нейронних мереж за рахунок поступового зменшення розміру мережі шляхом видалення зайвих каналів у згорткових шарах та додатковому навчанні отриманої зменшеної моделі для відновлення точності розпізнавання. Відповідно до розробленого методу, гіперпараметри мережі адаптивно змінюються, щоб компенсувати втрати точності після кожної ітерації прунінгу та зменшити час ітерації обробки даних.

Розроблено метод підвищення ефективності процесу обробки даних нейронними мережами на спеціалізованих обчислювальних архітектурах, що враховує технічні особливості обробки даних за допомогою глибоких нейронних мереж на спеціалізованих прискорювачах, наприклад, ітерація виконання обчислень. Також цей метод дозволяє визначити параметри такі, як

розмір порції даних, щоб збільшити продуктивність обробки даних за рахунок зменшення впливу накладних витрат ініціалізації і передачі даних.

Розроблено метод підвищення ефективності інфраструктури для обробки даних за допомогою глибоких нейронних мереж за рахунок зміни програмної та апаратної складової такої, як операційна система та інтерфейси підключення. Це дозволяє збільшити продуктивність та ефективність обробки даних за допомогою нейронних мереж на цільовій системі.

Розроблено програмний компонент діагностики легеневих аномалій за даними рентген знімків для дослідження ефективності роботи спеціалізованого прискорювача Coral Edge TPU USB в задачах медичного застосунку. В якості архітектури глибокої нейронної мережі для даної задачі було обрано ResNet50, яку було треновано на наборі даних ChestXray та адаптовано під спеціалізований прискорювач відповідно до розробленого комплексного методу адаптації.

Проведено аналіз результатів застосування методу адаптації глибоких нейронних мереж, що включає в себе адаптивний ітеративний прунінг, підвищення ефективності процесу обробки даних нейронною мережею та підвищення ефективності програмно-апаратної складової цільової хост-системи. За результатами застосування розробленого методу адаптивного ітеративного прунінгу було досягнуто прискорення 32,2 із точністю розпізнавання 96,2% (10 ітерацій прунінгу). За результатами аналізу технічних особливостей роботи спеціалізованих обчислювальних архітектур було виявлено, що значні показники прискорення, при використанні TPU у порівнянні з GPU, досягаються на пізніх ітераціях (>3) виконання обробки даних моделями глибоких нейронних мереж, коли витрати на ініціалізацію не впливають на продуктивність. Даний фактор треба враховувати при підвищенні ефективності процесу обробки даних нейронними мережами на прискорювачах зі спеціалізованою архітектурою. В результаті аналізу

факторів, що впливають на продуктивність цільової інфраструктури обробки даних за допомогою глибоких нейронних мереж, було досягнуто значних різниць в продуктивності при застосуванні різних комбінацій забезпечення цільової інфраструктури. При цьому, досягнуте прискорення склало 8,7.

Розроблені методи є складовою комплексного методу адаптації глибоких нейронних мереж і дозволяють підготувати обрану модель нейронної мережі для її застосування на зазначених вище прискорювачах нейронних мереж зі спеціалізованою архітектурою.

Ключові слова: адаптація, глибокі нейронні мережі, спеціалізовані обчислювальні архітектури, тензорні процесорні пристрої, периферійні обчислення.

ABSTRACT

Taran V.I. Method of deep neural networks adaptation for hardware with specialized architecture. - Qualified scientific work on the rights of the manuscript.

Dissertation for the degree of Doctor of Philosophy in the specialty 123 - Computer Engineering and 12 - Information Technologies. - National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2022.

The dissertation work is devoted to the development of the complex adaptation method of deep neural networks, which allows to increase productivity and efficiency of deep neural networks applications on the hardware with specialized architecture.

The complex deep neural networks adaptation method for specialized hardware was developed.

The method of adaptive iterative pruning for decreasing neural network model size was developed, which is based on subsequent decrease of the model size by removing redundant channels in convolution layers and additional model training for accuracy recovery. According to the proposed method, model hyper parameters are changed after every iteration to compensate accuracy loss and to achieve time decreasing of data processing iteration.

The method of neural network data processing efficiency improvement for specialized accelerators was developed. It is based on the technical aspects of deep neural network data processing on hardware with specialized architectures, for example data processing iteration and allows to determine processing parameters for decreasing influence of overheads.

The method of neural network processing infrastructure efficiency improvement was developed. It allows to optimize hardware and software configuration of the target system for increasing deep neural network data processing productivity.

The testing software for medical diagnostics in the context of edge computing was developed. It utilizes the developed deep neural networks adaptation method and specialized accelerator Coral Edge TPU.

The result analysis of the deep neural network adaptation method application was performed. It includes adaptive iterative pruning method, data processing efficiency improvement method and computational infrastructure efficiency improvement method. The speedup up to 32,2 and 96,2% accuracy were achieved after performing 10 iteration of the developed adaptive iterative pruning method. Based on the technical processing properties analysis for specialized processing architectures, some factors were identified, which have influence on the data processing. The considerable speedup values, while utilizing TPU compared to GPU, were achieved on the later data processing iterations (>3) with deep neural networks models, when initialization overheads had small influence on the accelerator performance. Such factor should be taken into account, while improving deep neural networks data processing efficiency on the accelerators with specialized architecture. Based on the deep neural network processing infrastructure analysis of factors, which had influence on the processing productivity, the following was identified. Considerable productivity difference was achieved, while utilizing different software and hardware combinations of the processing infrastructure. The achieved speedup value was up to 8,7.

Developed methods are parts of the complex deep neural networks adaptation method. It allows to prepare the selected neural network model for application on the accelerator with specialized architecture.

Keywords: adaptation, deep neural networks, specialized computing architectures, tensor processing units, edge computing.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці в яких опубліковано основні наукові результати дисертації:

1. **Taran, V.**, Gordienko, Y., Rokovyi, O., Alienin, O., Kochura, Y., Stirenko, S. (2022). Edge Intelligence for Medical Applications under Field Conditions. Lecture Notes on Data Engineering and Communications Technologies, 135, 71-80. Springer. ISSN 23674520. 10.1007/978-3-031-04809-8_6 (Scopus).
2. Alienin, O., Rokovyi, O., Gordienko, Y., Kochura, Y., **Taran, V.**, Stirenko, S. (2022). Artificial Intelligence Platform for Distant Computer-Aided Detection (CADe) and Computer-Aided Diagnosis (CADx) of Human Diseases. Lecture Notes on Data Engineering and Communications Technologies, 135, 91-100. Springer. ISSN 23674512. DOI 10.1007/978-3-031-04809-8_8 (Scopus).
3. Gordienko, Y., Kochura, Y., **Taran, V.**, Gordienko, N., Rokovyi, O., Alienin, O., Stirenko, S. (2021). “Last Mile” Optimization of Edge Computing Ecosystem with Deep Learning Models and Specialized Tensor Processing Architectures. Advances in Computers, 122, 303-341. Academic Press Inc. ISSN 00652458. DOI 10.1016/bs.adcom.2020.10.003 (Scopus).
4. Gordienko, Y., Kochura, Y., **Taran, V.**, Gordienko, N., Rokovyi, O., Alienin, O., Stirenko, S. (2020). Scaling Analysis of Specialized Tensor Processing Architectures for Deep Learning Models. Studies in Computational Intelligence, 866, 65-99. Springer Verlag. ISSN 1860949X. DOI 10.1007/978-3-030-31756-0_3 (Scopus) Q1.
5. Gordienko, Y., Kochura, Y., **Taran, V**., Gordienko, N., Bugaiov, A., Stirenko, S. (2019). Adaptive Iterative Pruning for Accelerating Deep Neural Networks. Procedings of XIth International Scientific and Practical Conference on Electronics and Information Technologies, 8892346, 173-178. IEEE. ISBN 9781728135618. DOI 10.1109/ELIT.2019.8892346 (Scopus).

6. **Taran, V.**, Gordienko, Y., Rokovyi, A., Alienin, O., Stirenko, S. (2020). Impact of Ground Truth Annotation Quality on Performance of Semantic Image Segmentation of Traffic Conditions. Advances in Intelligent Systems and Computing, 938, 183-193. Springer Verlag. ISSN 21945357. DOI 10.1007/978-3-030-16621-2_17 (Scopus) Q3.

7. Kochura, Y.; Gordienko, Y.; **Taran, V**.; Gordienko, N.; Rokovyi, A.; Alienin, O.; Stirenko, S. (2020). Batch Size Influence on Performance of Graphic and Tensor Processing Units During Training and Inference Phases. Advances in Intelligent Systems and Computing, 938, 658-668. Springer Verlag. ISSN 21945357. DOI 10.1007/978-3-030-16621-2_61 (Scopus) Q3.

8. **Taran, V.**, Gordienko, Y., Gordienko, N., Rokovyi, A., Kochura, Y., Alienin, O., Stirenko, S. (2018). Performance evaluation of deep learning networks for semantic segmentation of traffic stereo-pair images. ACM International Conference Proceeding Series, 73-80. Association for Computing Machinery. ISBN 9781450364256, DOI 10.1145/3274005.3274032 (Scopus).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....12

ВСТУП.....13

РОЗДІЛ 1 КОМП’ЮТЕРНІ ТЕХНОЛОГІЇ МОДИФІКАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ.....18

1.1 Спеціалізовані обчислювальні архітектури для глибоких нейронних мереж.....19

1.2 Методи зменшення моделей глибоких нейронних мереж.....26

1.3 Методи апаратно-залежного зменшення моделей глибоких нейронних мереж.....30

1.4 Архітектури глибоких нейронних мереж.....34

1.4.1 Нейронна мережа ResNet50.....35

1.4.2 Нейронна мережа VGG-16.....35

1.4.3 Нейронна мережа PSPNet.....36

1.4.4 Нейронна мережа MobileNet.....37

1.5 Набори даних для навчання нейронної мережі.....37

1.5.1 Набір даних MNIST.....37

1.5.2 Набір даних ImageNet.....38

1.5.3 Набір даних Cityscapes.....39

1.5.4 Набір даних CheXpert.....39

1.6 Фреймворки для роботи із глибокими нейронними мережами.....40

1.7 Висновки до розділу 1.....41

РОЗДІЛ 2 ОСОБЛИВОСТІ АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ ДЛЯ СПЕЦІАЛІЗОВАНИХ ОБЧИСЛЮВАЛЬНИХ АРХІТЕКТУР.....43

2.1 Обчислювальна складність глибоких нейронних мереж.....44

2.2 Периферійні обчислення.....47

2.3 Підвищення ефективності глибоких нейронних мереж.....49

2.4 Метрики оцінки продуктивності роботи моделі.....51

2.5 Застосування фреймворків для роботи із глибокими нейронними мережами на спеціалізованих обчислювальних архітектурах.....52

2.6 Глибокі нейронні мережі та спеціалізовані прискорювачі.....53

2.7 Висновки до розділу 2.....54

РОЗДІЛ 3 КОМПЛЕКСНИЙ МЕТОД АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ ДО АПАРАТНОГО ЗАБЕЗПЕЧЕННЯ ЗІ СПЕЦІАЛІЗОВАНОЮ АРХІТЕКТУРОЮ.....57

3.1 Метод адаптивного ітеративного прунінгу.....57

3.2 Метод підвищення ефективності процесу обробки даних нейронними мережами на спеціалізованих обчислювальних архітектурах.....61

3.3 Метод підвищення ефективності інфраструктури обробки даних глибокими нейронними мережами.....65

3.4 Комплексний метод адаптації глибоких нейронних мереж.....71

3.5 Адаптація моделі глибокої нейронної мережі під спеціалізовану обчислювальну архітектуру.....73

3.6 Програма діагностики легеневих аномалій.....78

3.7 Висновки до розділу 3.....83

РОЗДІЛ 4 АНАЛІЗ РЕЗУЛЬТАТІВ ЗАСТОСУВАННЯ МЕТОДУ АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ.....85

4.1 Аналіз результатів застосування методу адаптивного ітеративного прунінгу.....85

4.2 Аналіз технічних особливостей роботи спеціалізованих обчислювальних архітектур.....92

4.3 Аналіз факторів, що впливають на продуктивність цільової інфраструктури обробки даних глибокими нейронними мережами.....99

4.4 Аналіз продуктивності роботи спеціалізованого прискорювача в практичних умовах в медичному контексті.....105

4.5 Висновки до розділу 4.....110

ВИСНОВКИ.....112

ЛІТЕРАТУРА.....115

ДОДАТОК А Частина програмного коду.....128

ДОДАТОК Б Список публікацій здобувача.....151

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

DNN – Deep Neural Network

GRU – Gated Recurrent Unit

LSTM – Long Short-Term Memory

GPU – Graphic Processing Unit

TPA – Tensor Processing unit Architecture

TPU – Tensor Processing Unit

VPU – Vision Processing Unit

NPU – Neural Processing Unit

FPGA – Field Programmable Gate Array

ASIC – Application-specific integrated circuit

ВСТУП

Актуальність теми. Поточні успіхи машинного навчання, зокрема глибокого навчання та глибоких нейронних мереж, тісно пов’язані із прогресом у сфері високопродуктивних обчислень та спеціалізованих обчислювальних архітектур. Наразі глибокі нейронні мережі досягли високих показників точності у задачах комп’ютерного зору, детекції, розпізнавання та обробки мови, тощо. Це стало можливим за рахунок збільшення кількості шарів мережі, додавання спеціалізованих блоків: *ConvLayer*, *GRU/LSTM*. Але ціною за високу точність стає підвищення вимог до апаратних засобів, на яких виконуються обробка даних за допомогою подібних нейронних мереж – *DNN*. В цьому контексті широкого розповсюдження отримують прискорювачі глибоких нейронних мереж зі спеціалізованою архітектурою – *TPU*, *VPU*, *NPU*. Вони містять спеціалізовані обчислювальні модулі *Tensor Cores*, призначені для прискорення певних шарів мережі. Головним обмеженням подібних прискорювачів є необхідність у спеціальній адаптації нейронних мереж шляхом конвертації архітектури мережі у формат, підтримуваний конкретним пристроєм, або зменшення розміру мережі, оскільки пам’ять *TPU* є значно меншою у порівняні із звичайними *GPU*. Особливо використання подібних прискорювачів є актуальним в сфері *Edge Computing*, де присутні набагато більші обмеження на обчислювальні ресурси та потреби у високій енергоефективності. Підвищення продуктивності та ефективності обробки даних моделями нейронних мереж є однією із ключових задач у сучасній практиці використання глибоких нейронних мереж. Враховуючи зазначені обмеження спеціалізованих прискорювачів, доцільним є розробка методу адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур. Це дозволить виконувати обробку даних за допомогою нейронних мереж на спеціалізованих прискорювачах з високою продуктивністю, при цьому зберігати точність розпізнавання оригінальної мережі.

Зв’язок роботи з науковими програмами, планами, темами.

Тема дисертаційної роботи входить в план наукової роботи затвердженому на кафедрі обчислювальної техніки КПІ ім. Ігоря Сікорського, що враховує розпорядження Кабінету Міністрів України від 2 грудня 2020 р. № 1556-р про схвалення Концепції розвитку штучного інтелекту в Україні. Запропоновані у дисертації методи використані у науково-дослідній роботі НФДУ «Наука для безпеки людини та суспільства» – проєкт «Платформа штучного інтелекту для дистанційного автоматизованого виявлення та діагностики захворювань людини» реєстраційний номер проєкту: 2020.01/0490.

Мета і завдання дослідження. Метою дисертаційної роботи є підвищення продуктивності та ефективності обробки даних глибокими нейронними мережами на апаратному забезпеченні зі спеціалізованою архітектурою.

Для досягнення вказаної мети потрібно вирішити такі завдання:

- Дослідити сучасні методи зменшення розміру та обчислювальної складності моделей глибоких нейронних мереж;
- Покращити метод прунінгу для зменшення розміру моделі нейронної мережі;
- Розробити метод підвищення ефективності процесу обробки даних глибокими нейронними мережами на спеціалізованих обчислювальних архітектурах, включно із використанням на рівні периферійних обчислень (*Edge Computing*);
- Розробити метод підвищення ефективності інфраструктури для обробки даних за допомогою глибоких нейронних мереж;
- Розробити програмний компонент для обробки даних за допомогою глибоких нейронних мереж на прискорювачах зі спеціалізованою архітектурою в медичному контексті.

Об’єкт дослідження – процеси адаптації глибоких нейронних мереж для спеціалізованих обчислювальних пристроїв.

Предмет дослідження – методи адаптації глибоких нейронних мереж для спеціалізованих обчислювальних пристроїв.

Методи дослідження. Методичною основою дослідження є системне опрацювання та аналіз теоретичного матеріалу, присвяченого підвищенню продуктивності та ефективності обробки даних за допомогою моделей глибоких нейронних мереж, адаптації нейронних мереж та сучасним прискорювачам зі спеціалізованою архітектурою. В процесі даного дослідження були використані методи прунінгу, квантизації, статистичного аналізу, метрики обчислювальної складності, продуктивності та точності обробки даних, а також програмні засоби для високопродуктивних обчислень.

Наукова новизна отриманих результатів:

- *Вперше* запропоновано метод адаптації глибоких нейронних мереж, що надає можливість застосовувати комплексний підхід в адаптації і, на відміну від існуючих підходів, дозволяє підготувати модель глибокої нейронної мережі для прискорювачів зі спеціалізованою архітектурою, включно із її використанням на рівні периферійних обчислень (*Edge Computing*).
- *Вперше* розроблено метод підвищення ефективності процесу обробки даних глибокими нейронними мережами на спеціалізованих обчислювальних архітектурах, що враховує накладні витрати ініціалізації та передачі даних на спеціалізованих прискорювачах та, на відміну від існуючих підходів, дозволяє визначити гіперпараметри мережі, щоб підвищити продуктивність роботи мережі за рахунок зменшення накладних витрат при обробці даних.
- *Вперше* розроблено метод підвищення ефективності інфраструктури для обробки даних за рахунок зміни програмної та апаратної конфігурації і, на відміну від існуючих підходів, дозволяє зменшити час

виконання ітерації обробки даних глибокою нейронною мережею на цільовій системі.

- *Набув подальшого розвитку* метод прунінгу за рахунок ітеративного зменшення розміру моделі глибокої нейронної мережі та адаптивної зміни гіперпараметрів додаткового навчання, що дозволяє компенсувати втрати точності після кожної ітерації прунінгу та зменшити час ітерації обробки даних глибокою нейронною мережею.

Практичне значення отриманих результатів. Одержані результати дозволяють застосовувати моделі глибоких нейронних мереж на апаратному забезпеченні зі спеціалізованою архітектурою, включно із їх використанням на рівні периферійних обчислень (*Edge Computing*). Розроблений метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур дозволяє використовувати комплексний підхід в адаптації. Цей підхід поєднує методи прунінгу, квантизації, видалення зайвих класів об’єктів, підвищення ефективності процесу обробки даних глибокими нейронними мережами та підвищення ефективності програмно-апаратної складової цільової інфраструктури, до якої підключений спеціалізований прискорювач.

Особистий внесок здобувача. Дисертація є результатом самостійних наукових досліджень, в яких вкладено авторський підхід у адаптації моделей глибоких нейронних мереж до апаратного забезпечення зі спеціалізованою архітектурою. Наукові положення та основні результати, які містяться в дисертації, отримані здобувачем самостійно у процесі науково-дослідницької роботи. В роботах, опублікованих у співавторстві, дисертанту належать: [1] – розроблений комплексний метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур та програмний компонент діагностики легеневих аномалій за даними рентген знімків на базі спеціалізованого прискорювача *Coral Edge TPU*; [2] – адаптована під спеціалізований прискорювач модель глибокої нейронної мережі для діагностики легеневих аномалій; [3] – розроблений метод підвищення

ефективності програмно-апаратної складової цільової інфраструктури із якою використовується спеціалізований прискорювач; [4] – розроблений метод підвищення ефективності процесу обробки даних моделями глибоких нейронних мереж на спеціалізованих обчислювальних архітектурах, що враховує накладні витрати ініціалізації та обміну даними; [5] – розроблений метод адаптивного ітеративного прунінгу для зменшення розміру моделі глибокої нейронної мережі; [6] – аналіз впливу якості набору даних для тренування моделей глибоких нейронних мереж на точність розпізнавання; [7] – аналіз впливу розміру порції даних на час обробки даних нейронною мережею при використанні спеціалізованих прискорювачів; [8] – аналіз продуктивності та точності обробки даних нейронною мережею *PSPNet*.

Апробація результатів дисертації. Основні результати роботи опубліковано та обговорено на міжнародних та всеукраїнских наукових конференціях, зокрема на: The 2nd International Conference on Artificial Intelligence and Logistics Engineering, ICAILE 2022 (м. Київ, 2022); The 2nd International Conference on Computer Science, Engineering and Education Applications, ICCSEEA 2019 (м. Київ, 2019 р.); XIth International Scientific and Practical Conference on Electronics and Information Technologies, ELIT 2019 (м. Львів, 2019 р.); 19th International Conference on Computer Systems and Technologies, CompSysTech 2018 (м. Русе, Болгарія).

Публікації. За результатами дисертаційних досліджень опубліковано 8 наукових статей, що входять до наступних наукометричних баз даних з міжнародним індексом цитування: Scopus – 8, Web of Science – 1.

Структура і обсяг роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, загальних висновків, списку використаних джерел із 118 найменувань та додатків. Загальний обсяг дисертації становить 152 сторінок, з яких 103 сторінок основного тексту, 2 додатки на 24 сторінках, та містить 58 рисунків, 24 формул, 6 таблиць.

РОЗДІЛ 1
КОМП’ЮТЕРНІ ТЕХНОЛОГІЇ МОДИФІКАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ

Глибокі нейронні мережі досягли високих показників точності у задачах комп’ютерного зору, розпізнавання та обробки мови. Це стало можливим за рахунок збільшення кількості шарів мережі, додаванню спеціалізованих блоків: *ConvLayer* у задачах комп’ютерного зору, *GRU/LSTM* – у задачах обробки мови. Але ціною за високу точність стає підвищення вимог до апаратних засобів, на яких виконуються подібні нейроні мережі - *DNN*. В цьому контексті широкого розповсюдження отримують так звані прискорювачі нейронних мереж – *TPU*, *VPU*, *NPU* [1-5]. Вони містять спеціалізовані обчислювальні модулі *Tensor Cores* [6-7] призначені для прискорення певних шарів мережі. Головним обмеженням подібних прискорювачів є необхідність у спеціальній адаптації нейронних мереж шляхом конвертації архітектури мережі у формат, підтримуваний конкретним пристроєм, або зменшення її розміру, оскільки пам'ять *TPU* є значно меншою у порівняні зі звичайними *GPU*. Особливо, використання подібних прискорювачів є актуальним в сфері *Edge Computing* [8], де присутні більш високі обмеження на обчислювальні ресурси та енергоефективність.

Підвищення продуктивності та ефективності роботи моделей є однією із ключових задач у сучасній практиці використання глибоких нейронних мереж. Зазвичай подібні моделі працюють на потужних обчислювальних системах із використанням графічних прискорювачів – *GPU* таких, як *NVIDIA GTX* 1080 Ti, що забезпечує продуктивність 11,34 *TFLOPS* та має 11 *GB GDDR5X* із шиною даних 352 *bit*. Найбільш сучасний прискорювач *NVIDIA Tesla* A100 дозволяє виконувати до 19,5 *TFLOPS* та має 40 *GB HBM2* із шиною даних 5120 *bit*. При цьому, подібні прискорювачі споживають до 500 Ват електроенергії, що для задач *Edge Computing* зі збільшеними вимогами до апаратних ресурсів та енергоефективності є занадто великим.

Тому адаптація моделей глибоких нейронних мереж для систем із жорстким обмеженням обчислювальних ресурсів, зокрема для вбудованих пристроїв, має значну актуальність.

Для підвищення ефективності роботи моделей глибоких нейронних мереж з’являються методи, що досліджують можливість зменшення розміру та обчислювальної складності моделей нейронних мереж, при цьому мінімізуючи втрати точності роботи у порівняні із повноцінними початковими моделями. Сучасні дослідження у цьому напрямку можна виділити в окремі категорії:

- Оптимізація архітектури мережі [9];
- Квантизація – зменшення розмірності ваг параметрів [10];
- Структурне спрощення архітектури мережі [11, 12].

При цьому, структурне спрощення архітектури мережі додатково ділиться на наступні підходи:

- Факторизація тензорів [12];
- Зменшення зв’язності шарів мережі [13];
- Прунінг каналів та шарів мережі [14].

Перші два підходи мають певні обмеження [15], тому наразі прунінг став найбільш широко застосованим підходом зменшення нейронної мережі наряду із комбінаціями інших підходів.

1.1 Спеціалізовані обчислювальні архітектури для глибоких нейронних мереж

Поточні успіхи машинного навчання, зокрема глибокого навчання та глибоких нейронних мереж, тісно пов’язані із прогресом у сфері високопродуктивних обчислень та спеціалізованих обчислювальних архітектур – *TPU, VPU, NPU*. Дані архітектури, у порівняні із архітектурою графічних прискорювачів *GPU*, мають спеціальні обчислювальні модулі, що забезпечують більшу ефективність та продуктивність обробки даних

глибокими нейронними мережами. Їхній паралелізм та архітектурні особливості дозволяють виконувати дані задачі більш ефективно [16-19], враховуючи зростання складності проблем машинного і глибокого навчання, збільшення розміру датасетів, моделей глибоких нейронних мереж, кількості ознак, класів та категорій оброблювальних об’єктів. При цьому зростає потреба у більшій точності розпізнавання, швидкості навчання і опрацювання, що тісно пов’язана із вимогами більших обчислювальних потужностей, об’ємів пам’яті та пропускної здатності мережі. Для вирішення цих проблем використовуються різні обчислювальні інфраструктури: від комп’ютерів із моделлю загальної пам’яті до паралельних апаратних архітектур таких, як високопродуктивні обчислювальні кластери із загальною і розподіленою пам’яттю, та хмарні системи із розподіленою пам’яттю, які наразі найбільш популярні.

Протягом останнього десятиріччя обробка даних глибокими нейронними мережами була невід’ємно пов’язана із використанням графічних обчислювальних пристроїв *GPU* як процесорів загального призначення *GPGPU*. Однак нещодавно почала виникати велика кількість альтернативних платформ, від загальновідомих цифрових сигнальних процесорів *DSP* та програмованих логічних інтегральних схем *FPGA* [20, 21] до доволі нових архітектур спеціалізованих інтегральних схем *ASIC* [22], що включають нейроморфні процесори [23, 24], архітектури тензорних процесорів (*tensor cores*) від *NVIDIA* [25] і (*tensor processing units*) від *Google* [26], які оптимізовані під обчислювальні задачі нейронних мереж.

Для ефективної обробки даних сучасними глибокими нейронними мережами на даних обчислювальних інфраструктурах потрібно враховувати аспекти роботи як самих нейронних мереж, так і спеціалізованих обчислювальних архітектур. Це особливо важливо враховувати під час використання новітніх хмарних систем для глибокого навчання таких, як *Clarifai*, *Google Cloud Vision*, *Rekognition*, *Polly*, *Lex Amazon*, *Microsoft Azure*

Cognitive Services, *IBM Watson*, які стали доволі популярними у сфері наукових та комерційних застосунків нейронних мереж.

На відміну від традиційних задач для високопродуктивних обчислювальних кластерів, задачі машинного та глибокого навчання здебільшого вирішуються за рахунок застосування спеціалізованих прискорювачів таких, як *GPU* та *FPGA*. Сутність цих спеціалізованих архітектур полягає у внутрішній можливості ефективно використовувати паралелізм даних та обчислень. Через це, використання подібних архітектур стало де-факто стандартом для задач машинного та глибокого навчання. У відповідь на нові вимоги у збільшеній обчислювальній потужності з’являються нові багаторівневі архітектури кластерних та хмарних систем, спеціалізованих архітектур тензорних прискорювачів та їх комбінації [27].

Додаткового поштовху у розвитку нових спеціалізованих обчислювальних архітектур дає використання машинного і глибокого навчання як на різноманітних мобільних пристроях, типу смартфонів, планшетів, носимої електроніки, так і на промислових пристроях типу систем безпеки, моніторингу здоров’я, автономного водіння, тощо. Основна складність застосування глибоких нейронних мереж в даному контексті пов’язана із жорсткими вимогами до використання пам’яті, кількості обчислювальних операцій та енергоефективності [28-32]. Для вирішення цих проблем виникають і починають широко використовуватися спеціалізовані обчислювальні архітектури такі, як (*tensor cores*) від *NVIDIA* [25] і (*tensor processing units*) від *Google* [26].

Сучасні тренди прискорення обробки даних глибокими нейронними мережами на мобільних пристроях із використанням *TPA* можна класифікувати у декілька категорій: зменшення моделі мережі, квантизація, структурне спрощення мережі [33, 34]. Наприклад, спрощення мережі включає декілька підходів таких, як факторизація тензорів [35], зменшення зв’язності шарів мережі [36, 37], прунінг каналів та шарів мережі [38-40].

Зазвичай, одні огляди ефективності обробки даних глибокими нейронними мережами пов’язані із різними загальними перспективами, моделями, алгоритмами оптимізації та датасетами. Інші враховують обчислювальні особливості компонентів нейронних мереж із використанням паралелізму на цільових апаратних архітектурах і технік зменшення загальних витрат на цільовій платформі [41]. В цьому контексті, стратегічне бачення результатів застосування подібних технік адаптації глибоких нейронних мереж має велике значення на подальше впровадження та розвиток алгоритмів і спеціалізованих обчислювальних архітектур, націлених на використання у системах із жорсткими вимогами до обчислювальних ресурсів.

Головною складовою спеціалізованих обчислювальних архітектур є тензорні ядра. Це апаратний компонент для прискорення математичних операцій із матрицями, який був представлений компанією *NVIDIA* в архітектурі *Volta*. Тензорні ядра забезпечують роботу із матрицями розмірності 4 x 4 x 4 і забезпечують операцію D = A * B + C, де A, B, C, D є матрицями розмірності 4 x 4. В комбінації із програмною бібліотекою *TensorRT*, тензорні ядра дозволяють виконувати деякі операції трансформації та оптимізації, що дозволяють підвищити продуктивність глибоких нейронних мереж. Наприклад, бібліотека надає можливість прибирати шари мережі, вивід яких не потрібен для загального результату. Подібна операція дозволяє усунути необхідність виконувати обчислення в подібних зайвих шарах. Також є можливість зливання декількох окремих шарів (згортка, зсув, активація) в єдиний шар. Тензорні ядра разом із *TensorRT* дозволяють використовувати зменшену розрядність представлення ваг та параметрів мережі. Наприклад, зменшення точності представлення до *FP16* дозволяє зменшити загальні витрати пам’яті для зберігання моделі мережі та кількості операцій читання/запису у порівнянні із *FP32*/*FP64*. Це покращує можливості використання великих моделей глибоких мереж на апаратурі з обмеженими ресурсами і також дозволяє пришвидшити арифметичні

операції. Додатково, *TensorRT* може автоматично використовувати тензорні ядра архітектури *Volta* у *Tesla V100* для виконання операції прогнозування – *inference*, використовуючи розрядність *FP16*. Як результат, це дозволяє значно зменшити час виконання обчислень. Наприклад, модель *ResNet-50* на *GPU V100 + TensorRT + TCs* (*FP16*) в 4 рази швидше аналогічної моделі (*FP32*). Окрім цього, архітектура *Turing* у *NVIDIA Tesla T4* додатково підтримує арифметику *INT8* та *INT4*, що також дозволяє додатково пришвидшити обчислення.

Нещодавно значної уваги до себе привернула ще одна тензорна архітектура від *Google* – тензорні процесорні пристрої *TPU*, яка дозволяє підвищити ефективність та швидкість обчислень для глибоких нейронних мереж. *TPU* містить 256 x 256 арифметико-логічних пристроїв АЛП, які здатні виконувати 65,536 операцій множення та суми над 8-и бітними цілими числами при кожному циклі. *TPU* працює на частоті 700 MHz і у заявлених тестах дає пришвидшення операцій глибоких нейронних мереж до 30 разів і до 80 разів більшу енергоефективність у порівняні із *CPU* та *GPU* [26]. Це стає можливим за рахунок того, що *TPU* спеціально призначений для виконання операцій глибоких нейронних мереж (рис. 1.1). Наразі доступні декілька варіацій *TPU* від *Google*. До них входять спеціальні портативні прискорювачі *Coral Edge TPU USB Stick* та *Coral TPU Dev board* [2], а також хмарні варіанти інфраструктур із прискорювачами: *Cloud TPU v2* (180 *teraflops*, 64 *GB High Bandwidth Memory* (*HBM*)), *Cloud TPU v3* (420 *teraflops*, 128 *GB HBM*), *Cloud TPU v2 Pod Alpha* (11.5 *petaflops*, 4 *TB HBM*, *2-D toroidal mesh network*) [26]. *Edge TPU* — спеціалізована інтегральна схема *ASIC*, яка підтримує лише цілочисельні операції із розрядністю 8/16 та працює з моделями *TensorFlow Lite* [42].

Компанія *Intel* також випустила декілька модифікацій пристроїв для обробки графічної інформації – *VPU*, що базуються на системі на чипі – *SoC* низької потужності, спеціально призначеній для прискорення застосунків комп’ютерного зору. Наприклад, *Intel Movidius Myriad X* [43] містить

декілька процесорів та обчислювальних блоків, оптимізованих для забезпечення високого паралелізму операцій глибоких нейронних мереж. *VPU* доступні у різних конфігураціях таких, як *USB* версія *Intel Neural Compute Sticks* [3] – *NCS* із 4 Гб вбудованої *RAM*. Подібний пристрій може підключатись до *USB* порта звичайного комп'ютера і виконувати роль со-процесора для збільшення швидкості обробки даних моделями глибокий нейронних мереж. Деякі виробники також пропонують подібні пристрої такі, як *Horned Sungem* [4], що базується на процесорі *Intel Movidius MA245x*. Узагальнені дані по засобам для роботи з нейронними мережами представлені у таблицях 1.1, 1.2.

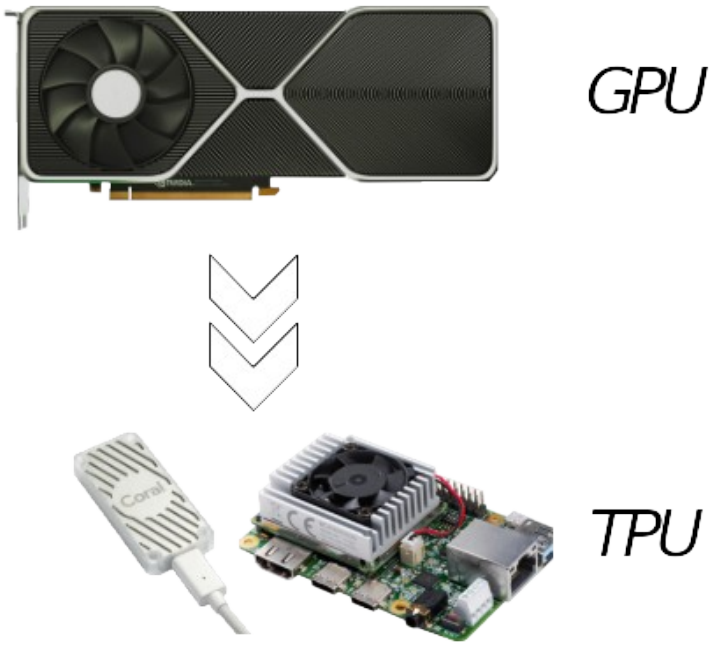

Рисунок 1.1 — Прискорювачі глибоких нейронних мереж

В контексті парадигми *Edge Computing* використання подібних спеціалізованих пристроїв для глибоких нейронних мереж є досить привабливим за рахунок їх високої продуктивності і енергоефективності, низької затримки, та підвищеній приватності. Парадигма *Edge Computing* розглядається в контексті розподілених обчислень, де певні типи обчислень розділяються між декількома рівнями: *Cloud*, *Fog*, *Dew*. Обчислення на рівні *Fog* зазвичай включає граничні обчислювальні пристрої, що виконують

суттєву порцію обробки, зберігання та передачі первинних даних, після чого отримана інформація передається через мережеві зв’язки. Подібна парадигма забезпечує ефективну обробку даних великого об’єму від різноманітних пристроїв *IoT* і відповідає вимогам у забезпеченні обробки у реальному часі, прогнозованості продуктивності та низької затримки. Обчислення рівня *Dew* розширює дану концепцію до мікросервісного рівня, дозволяючи зміщувати обчислення даних ще ближче до кінцевих пристроїв. Це дозволяє розподіляти обробку даних між кінцевими пристроями таких, як ноутбуки, смартфони, носима електроніка та інші гаджети. Зазначені особливості дають можливість виконувати складні обчислювальні задачі, використовуючи парадигму *Edge Computing* навіть в умовах, коли зв’язок із хмарним сервісом не доступний [44-46].

Таблиця 1.1 — Сучасні фреймворки для роботи із глибокими нейронними мережами [47]

Framework	Core development language	Interface	Part running on the edge	Applications
TensorFlow Lite (Google)	C++, Java	Android, iOS Linux	TensorFlow Lite NN API	computer vision speech recognition
Caffe2, Caffe2Go (Facebook)	C++	Android iOS	NNPack	image analysis, video analysis
Apache MXNet	C++, R Python	Linux MacOS Windows	Full Model	object detection recognition
Core ML2 (Apple)	Python	iOS	CoreML	image analysis
ML Kit (Google)	C++ Java	Android iOS	Full Model	image recognition, text recognition, bar-code scanning
AI2GO	C, Python Java, Swift	Linux macOs	Full Model	object detection classification
DeepThings	C/C++	Linux	Full Model	object detection
DeepIoT	Python	Ubilinux	Full Model	human activity recognition user identification
DeepCham	C++ Java	Linux Android	Full Model	object recognition
SparseSep	-	Linux Android	Full Model	mobile object recognition audio classification
DeepX	C++, Java, Lua	Linux Android	Full Model	mobile object recognition audio classification
Edgent	-	Ubuntu	Major part of the DNN	image recognition
daBNN	Arm Assembly, C++, Java	Mobile OSs (Android and others)	Full model	Computer vision
TensorFlow Federated (Google)	Python	iOS, Android	Distributed training	computer vision
EdgeML Library (Microsoft Research)	C++	-	-	image recognition
CONDENSA	Python	Linux, Windows, MacOS	Full Model	language modeling and computer vision

Таблиця 1.2 — Типи пристроїв для роботи із глибоким нейронними мережами [47]

Device type	Name	GPU	CPU	RAM	Flash memory	Power	Applications
(ASICs)	Google Coral Dev Board	GC7000 Lite Graphics + Edge TPU coprocessor	Quad Cortex-A53, Cortex-M4F	1 GB LPDDR4	8 GB LPDDR4	5V DC	image processing
	SparkFun Edge	-	32-bit ARM Cortex-M4F 48MHz	384KB	1MB	6uA/MHz	speech recognition
	Intel Movidius NCS	High Performance VPU	Myriad 2 VPU	1 GB	4 GB	2 trillion 16-bit ops/s within 500 mW	classification computer vision
	BeagleBone AI	-	Cortex-A15 Sitara AM5729 SoC with 4 EVEs	1 GB	16 GB	-	computer vision
	Eta Compute EMC3531	-	ARM Cortex-M3 NXP Coolflux DSP	-	-	-	audio, video analysis
(FPGAs)	ARM ML	-	ARM ML processor	1 GB	-	4 TOPs/W (Tera Operations)	image, voice recognition
Embedded GPUs	Raspberry Pi	400MHz VideoCore IV	Quad Cortex A53 @ 1.2GHz	1 GB SDRAM	32 GB	2.5 Amp	video analysis
	NVIDIA Jetson TX1	Nvidia Maxwell 256 CUDA cores	Quad ARM A57/2 MB L2	4 GB 64 bit LPDDR4 25.6 GB/s	16 GB eMMC, SDIO, SATA	10-W	video, image analysis robotics
	NVIDIA Jetson TX2	Nvidia Pascal 256 CUDA cores	HMP Dual Denver 2/2 MB L2 + Quad ARM A57/2 MB L2	8 GB 128 bit LPDDR4 59.7 GB/s	32 GB eMMC, SDIO, SATA	7.5-W	video, image analysis robotics
	NVIDIA Jetson NANO	Nvidia Maxwell 128 CUDA cores	Quad ARM A57 MPCore	4 GB 64 bit LPDDR4 25.6 GB/s	16 GB eMMC, 5.1 Flash	5 - 10-W	computer vision audio analysis
	OpenMV Cam	-	ARM 32-bit Cortex-M7	512KB	2 MB	200mA @ 3.3V	image processing
Open (ISAs)	RISC-V GAP8	-	nona-core 32-bit RISC-V @250 MHz	16 MiB SDRAM	-	1 GOPs/mW	image, audio processing

1.2 Методи зменшення моделей глибоких нейронних мереж

Особливістю глибоких нейронних мереж є наявність великої кількості надлишкових ваг, які слабо або зовсім не впливають на точність роботи моделі, але їх наявність призводить до збільшення витрат пам’яті та обчислювальних ресурсів системи. Прунінг можна назвати апаратно незалежним методом зменшення нейронних мереж, оскільки здебільшого зменшується складова мережі пов’язана із даними, що обчислюються. В ролі цих даних виступають ваги мережі, що оптимізуються в процесі навчання мережі.

Щоб зменшити витрати пам’яті та збільшити енергоефективність сучасних нейронних мереж при їх розміщенні та роботі на портативних пристроях, був запропонований метод, який дозволяє зменшити кількість пам’яті необхідної для роботи моделі, при цьому зберігаючи точність оригінальної моделі [15]. Спочатку мережа зменшується за рахунок видалення надлишкових зв’язків, зберігаючи лише найінформативніші з них. Далі ваги квантизуються, що дозволяє декільком зв’язкам в мережі використовувати одну і ту саму вагу. При цьому зберігаються тільки найбільш ефективні ваги. В кінці виконується кодування ваг, що дозволяє оптимізувати розподіл зміщення [15].

Техніка прунінгу каналів дозволяє досягти значного зменшення ціни імплементації, в тому числі витрат пам’яті, обчислювальних ресурсів, витрат енергії, не потребуючи при цьому спеціалізованого апаратного забезпечення. Це дає можливість розмістити *CNN* мережу на пристроях із обмеженими ресурсами (*edge platforms*). У роботі [48] запропоновано метод структурного прунінгу надлишкових каналів у натренованій мережі *CNN*. Метод використовує запропоновану постановку проблеми вибору підмножини для прунінгу, яка вирішується за допомогою циклічної факторизації. Експериментально показано зменшення обчислювальної складності у 4,29 та 2,84 разів для моделей *VGG-16* та *ResNet-50* відповідно при втраті точності до 2,5%.

Використання прунінгу та структурної модифікації нейронної мережі зарекомендували себе у задачі сегментації людини. У роботі [49] авторами було продемонстровано можливість тюнингу та зменшення моделі *PSPNet-50* на датасетах *Baidu* та *Flickr*. Метод полягає у прунінгу згорткових шарів мережі та структурній модифікації оригінальної моделі із подальшим двох етапним прунінгом фільтрів (рис. 1.2). Результати показали зменшення кількості параметрів до 7,5 разів при втраті точності до 2%. Додатково було введено два показники втрат, що дозволило покращити точність *IoU* при додатковому навчанні.

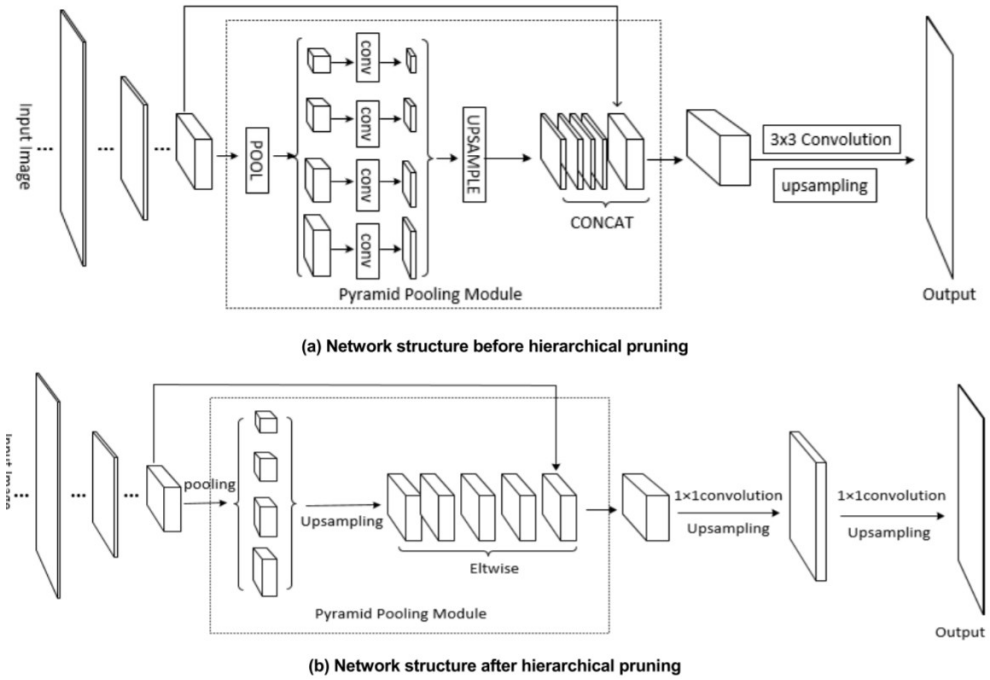

Рисунок 1.2 — Двох етапний метод прунінгу фільтрів [49]

Велика кількість пропускної здатності пам’яті витрачається при передачі даних між локальними буферами та зовнішніми *DRAM*, що призводить до значних втрат продуктивності в апаратних прискорювачах *CNN*, особливо при передачі карт активації. Щоб зменшити навантаження на шину пам’яті, у роботі [50] запропоновано заміщати неважливі блоки мережі на нульовий блок регуляризації карти активації. Дана стратегія має невеликі обчислювальні витрати і може бути легко інтегрована разом з існуючими методами прунінгу для кращої продуктивності. Даний метод дозволяє зменшити до 76% витрати пропускної здатності мережі при втраті точності в межах 1%.

Одна зі особливостей структурної видозміни глибоких нейронних мереж є складність передбачити, які шари, фільтри та параметри найбільше або найменше впливають на загальну точність роботи моделі. Для вирішення даної проблеми Лі та ін. [51] розробили візуалізатор прунінгу (рис. 1.3). Даний застосунок дозволяє проаналізувати складний взаємозв’язок між шарами мережі під час прунінгу та тюнингу, і дозволяє збалансувати ефективність та точність мережі.

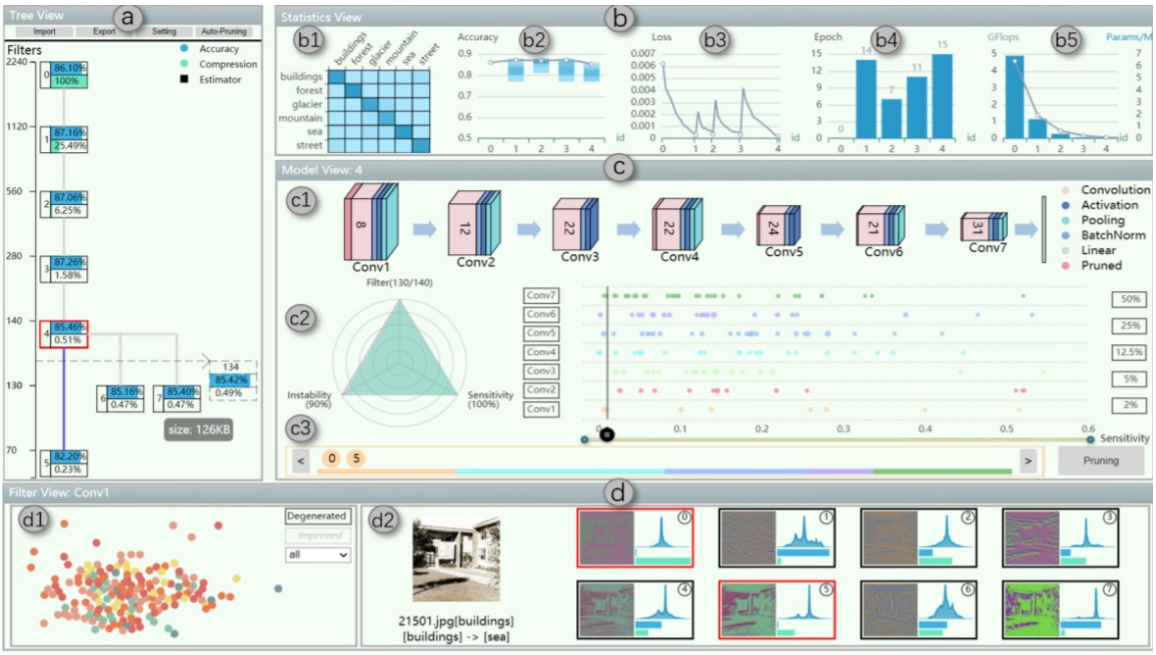

Рисунок 1.3 — Програмне забезпечення візуалізації процесу прунінгу [51]

Ще однією особливістю прунінгу каналів є те, що у результуючій моделі залишається велика кількість надлишкових карт ознак. У роботі [52] автори запропонували алгоритм федеративного прунінгу, який дозволяє досягти більш компактної моделі при незначній втраті точності. Відміна від існуючих методів полягає у тому, що федеративний прунінг видаляє всі зайві фільтри разом із їх картами ознак, комбінуючи ваги параметрів за важливістю каналу замість того, щоб використовувати один статичний критерій. За результатами роботи алгоритму для тренованої на датасеті *CIFAR-10* моделі *VGG-19* було досягнуто 92,5% зменшення загальної кількості параметрів при втраті точності 0,23%.

Аналогічні дослідження в області адаптації глибоких нейронних мереж для цільового засовування розвиваються в Україні. Наприклад, в [53] показано модифікацію структурних параметрів згорткової нейронної мережі для виконання біометричної аутентифікації за даними райдужної оболонки ока. За результатами дослідження було досягнуто точності розпізнавання у 0,95. Також у роботі [54] наведено спосіб, який шляхом реалізації процедури визначення вхідного поля та процедури адаптації структурних параметрів згорткової нейронної мережі, забезпечує розпізнавання особи і емоцій

користувача за клавіатурним почерком. У роботі [55] досліджено можливість удосконалення програмного забезпечення для розпізнавання комп’ютерних вірусів за рахунок використання глибоких нейронних мереж. Експериментально показано високу точність роботи подібного підходу. Для прогнозування надійності програмного забезпечення також використовуються рішення із нейронними мережами. Для покращення точності прогнозування та збільшення швидкості обчислень у дослідженні [56] запропоновано використовувати ансамбль нейронних мереж із 4 моделей, що показало точність роботи до 93,97 %. Можливості нейронних мереж також використовуються в засобах моделювання процесів [57, 58], а також удосконалюється та покращується процес навчання та роботи нейронних мереж [59-62].

1.3 Методи апаратно-залежного зменшення моделей глибоких нейронних мереж

Квантизація, на відміну від прунінгу, зменшує розмір моделі нейронної мережі за рахунок зміни представлення та розмірності даних мережі в обчислювальній системі (рис. 1.4). Тому цей метод можна віднести до апаратно-залежних, оскільки зміна розмірності має підтримуватись архітектурно самою обчислювальною системою. Даний метод полягає у представленні величин параметрів та функцій активації нейронної мережі у зменшеному діапазоні. Зазвичай, під час роботи нейронних мереж на графічних прискорювачах *GPU*, зазначені величини представляються у вигляді чисел із плаваючою комою розмірності 32 біта – *FP32*. Це дозволяє ефективніше використовувати динамічний простір представлення величин і забезпечує високі показники точності моделі під час навчання та роботи. Із негативних наслідків можна відмітити великі обсяги пам’яті графічного процесора та файлу моделі, що зберігає відповідні значення параметрів, а також збільшена обчислювальна складність арифметики із плаваючою комою.

Це автоматично тягне за собою збільшення енерговитрат пристроїв, на яких працюють нейронні мережі.

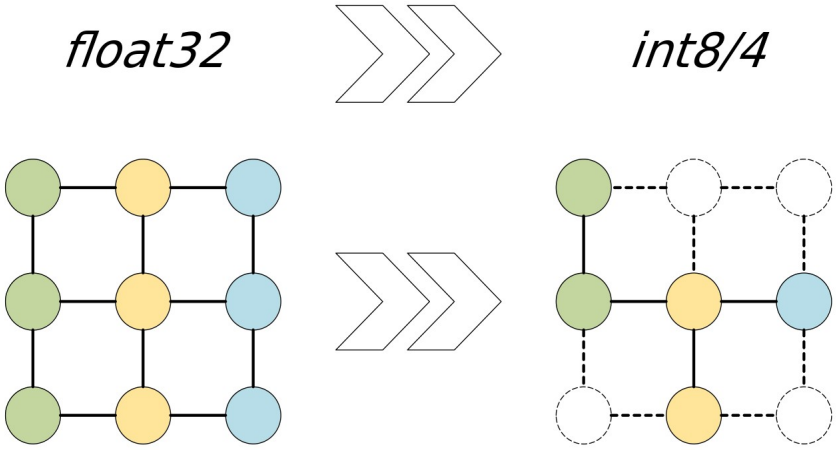

Рисунок 1.4 — Квантизація (зверху) та прунінг (знизу) моделей нейронних мереж

Наведені фактори ускладнюють застосунок нейронних мереж на вбудованих та портативних пристроях у контексті *Edge Computing*, а використання арифметики із плаваючою комою робить неможливим використання подібних нейронних мереж на спеціалізованих обчислювальних архітектурах. Наприклад, повнорозмірна архітектура мережі *ResNet50* [63] містить близько 25,6 млн. параметрів та має обчислювальну складність 3,8 *GFLOPS*, а збережена модель *ResNet50* займає 100 MB. Для зменшення витрат пам’яті та обчислювальної складності ведуться дослідження по зменшенню представлення величин до цілих чисел розмірності 16,8,4 бітів – *INT16*, *INT8*, *INT4*. Використання цілочисельної арифметики дозволяє використовувати нейронні мережі на спеціалізованих обчислювальних архітектурах, наприклад *TPA*. Найбільш екстремальні підходи у квантизації полягають у зменшенні розмірності до декількох бітів і використанні бінарних (-1, 1) та тернарних (-1, 0, 1) параметрів у повно зв'язних та згорткових шарах [64]. Подібний підхід дозволяє уникнути операції множення і обмежитись лише операціями додавання та віднімання. Використання бінарних параметрів дозволяє обмежитись лише побітними операціями. Растергарі та ін. [65] запропонували *XNOR-Networks* (рис. 1.5), де

у згорткових шарах використовуються операції *XNOR*, що дозволяє досягти прискорення до 58 раз та у 32 рази економію у витратах пам’яті.

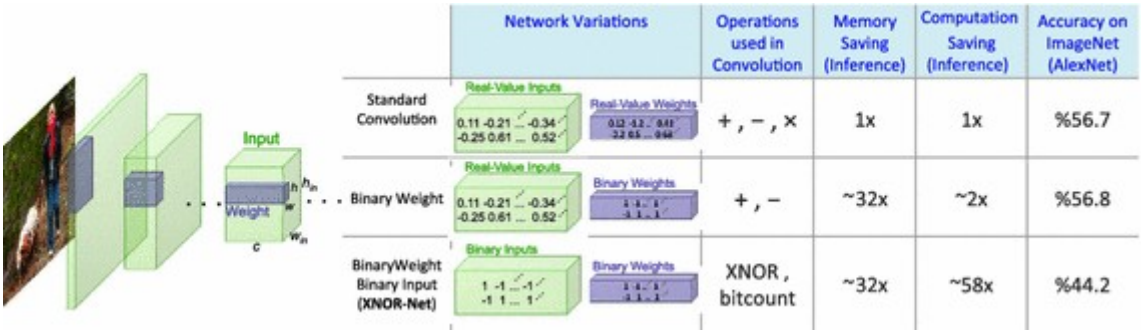

Рисунок 1.5 — Архітектура мережі XNOR-Networks [65]

Параджулі та ін. [66] розробили тернарну мережу, яка продемонструвала ефективну квантизацію та високі показники під час навчання. Використання даного підходу дозволило усунути операції множення і замінити їх на операції зсуву та додавання. Це дало суттєвий виграш у енергоефективності при роботі на спеціалізованих архітектурах та портативних пристроях таких, як *FPGA Zynq Ultrascale ZU9EG* та *Raspberry Pi*. На *FPGA* витрати енергії зменшились зі 49mW при операціях множення до 10mW при операціях зсуву. Аналогічне покращення у швидкодії було досягнуто для *Raspberry Pi*, час виконання операції зменшився з 93 до 67 нано секунд.

Загалом методи квантизації можна поділити на дві категорії:

- *post-training quantization* – квантизація після навчання;
- *quantization aware training* – квантизація під час навчання.

Квантизація після навчання – є найбільш простим методом квантизації у тому плані, що для квантизації вже готової моделі не потрібно виконувати додаткового навчання. Вона полягає у зменшенні розрядності параметрів та активацій вже навченої мережі. Для цього варіанту квантизації можна виділити два різновиди: квантизація лише параметрів та повна квантизація параметрів та активацій [67]. Перший варіант досягається за допомогою простої конвертації параметрів із *FP32* у *INT8* і не потребує додаткової

інформації. Даний варіант дозволяє зменшити розмір кінцевої моделі нейронної мережі, ніяким чином не впливаючи на обчислювальну складність мережі, оскільки обчислення активацій залишається в арифметиці із плаваючою комою. Щоб зменшити обчислювальну складність мережі, використовують другий варіант, а саме квантизацію параметрів та активацій. Цей варіант переводить всі величини та операції у нейронній мережі в цілочисельне представлення *INT8*. Для використання даного варіанту, додатково необхідно застосовувати репрезентативний набір даних для розрахунку динамічного діапазону активацій. Зазвичай використовується порція зі 100-300 елементів, щоб знайти діапазон значень активацій. Варіант квантизації після навчання для параметрів та активацій використовується при перенесенні моделей нейронних мереж на спеціалізовані обчислювальні пристрої, наприклад *Google Coral Edge TPU USB Stick* та *Coral TPU Dev board* [2], оскільки більшість подібних пристроїв підтримують лише цілочисельну арифметику. Наразі більшість сучасних фреймворків для роботи з нейронними мережами такі, як *TensoFlow*, дозволяють виконувати квантизацію після навчання, зменшуючи розрядність до *INT8* без значних втрат точності моделі [68].

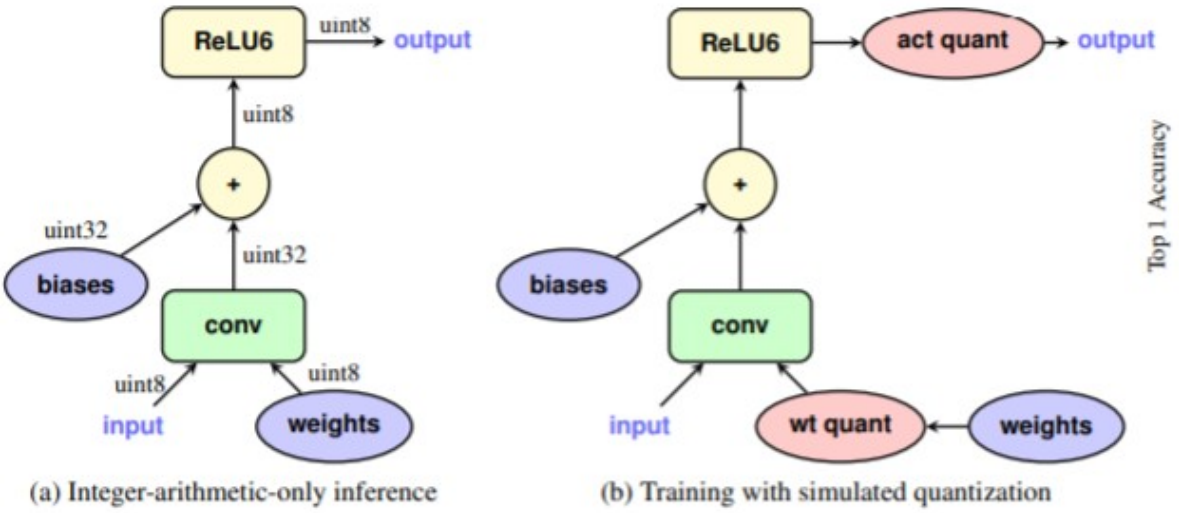

Рисунок 1.6 — Використання цілочисельної арифметики при роботі мережі (а) та тренування з симуляцією квантизації (б) [69]

Квантизація під час навчання є більш складним методом квантизації, що дозволяє досягти значно більшого зменшення розмірності до *INT4*, *INT2*, *INT1*, при незначній втраті точності. Даний спосіб полягає у додаванні спеціальних шарів квантизації у нейрону мережу, в той час як всі обчислення градієнтів для оновлення параметрів мережі залишаються у *FP32*. Джейкоб та ін. [69] продемонстрували метод квантизації нейронної мережі, що базується на апроксимації обчислень із плаваючою комою. За рахунок введення додаткових шарів (рис. 1.6), що симулюють квантизацію, авторам під час навчання квантизованої моделі вдалось досягти майже початкову точність оригінальної моделі, при використанні лише цілочисельної арифметики. На ряду зі зменшенням розміру моделі майже у 4 рази, було досягнуто покращення продуктивності при роботі на архітектурі *ARM*.

Подібний метод квантизації реалізовано у спеціальному модулі *TensorFlow Model Optimization API*, дозволяє виконувати квантизація під час навчання для моделей *TensorFlow* та *Keras*.

1.4 Архітектури глибоких нейронних мереж

В дослідженні було використано декілька різних архітектур глибоких нейронних мереж *ResNet50* [63], *VGG-16* [70], *MobileNet* [71] та *PSPNet* [72] для типів задач класифікації, локалізації та сегментації. Класифікація є початковою задачею у сфері комп’ютерного зору і полягає у присвоєнні певної мітки-класу для кожного елементу-зображення із набору даних. Логічним її розвитком є локалізація, в якій за допомогою карти ознак, побудованої за даними мережі, виділяються зони, які найбільше впливають на присвоєння елементу-зображенню певної мітки-класу. Найбільш складною є сегментація, що виконує присвоєння певної мітки-класу для кожного окремого пікселя на елементі-зображенні. В усіх цих задачах отримати високу точність стає можливим за рахунок використання згорткових шарів, які за допомогою наборів фільтрів виокремлюють певні ознаки із вхідних

даних. При цьому згорткові шари на верхніх рівнях виокремлюють дрібні ознаки-примітиви, а шари на глибших рівнях – складні ознаки, що утворюються за рахунок поєднання більш простих ознак. Наразі одними із широко розповсюджених згорткових архітектур глибоких нейронних мереж є *ResNet50*, *VGG-16*, *PSPNet*, спеціалізована під периферійні пристрої мережа *MobileNet*.

1.4.1 Нейронна мережа ResNet50

ResNet50 [63] є однією із найбільших нейронних мереж. В її архітектурі використовуються оригінальна організація зв’язків між шарами, що дозволяє виконати навчання мережі подібного розміру. На відміну від звичайного послідовного з’єднання шарів, в даній мережі певна кількість згорткових шарів об’єднуються у групи між якими додатково вводяться обхідні зв’язки. Подібна організація дає можливість навчати шари мережі не тільки враховуючи значення із попередніх шарів, але також використовуючи залишкові дані отримані з попередніх груп згорток. Це дозволяє уникнути проблему деградації точності та згасаючих градієнтів, що притаманна для глибоких моделей, та тренувати моделі з глибиною до 152 шарів із подальшим покрашенням точності роботи моделі. Архітектура мереж типу *ResNet* наведено на рис. 1.7.

1.4.2 Нейронна мережа VGG-16

VGG-16 [70] складається із 16 згорткових шарів і має доволі однорідну архітектуру з фільтрами розміром 3 × 3 для всіх згорткових шарів. Ця мережа добре документована і часто використовується для виокремлення ознак із графічних даних. Ваги тренованої мережі є загальнодоступними і використовуються у застосунках для комп’ютерного зору.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		[3×3, 64; 3×3, 64]×2	[3×3, 64; 3×3, 64]×3	[1×1, 64; 3×3, 64; 1×1, 256]×3	[1×1, 64; 3×3, 64; 1×1, 256]×3	[1×1, 64; 3×3, 64; 1×1, 256]×3
conv3_x	28×28	[3×3, 128; 3×3, 128]×2	[3×3, 128; 3×3, 128]×4	[1×1, 128; 3×3, 128; 1×1, 512]×4	[1×1, 128; 3×3, 128; 1×1, 512]×4	[1×1, 128; 3×3, 128; 1×1, 512]×8
conv4_x	14×14	[3×3, 256; 3×3, 256]×2	[3×3, 256; 3×3, 256]×6	[1×1, 256; 3×3, 256; 1×1, 1024]×6	[1×1, 256; 3×3, 256; 1×1, 1024]×23	[1×1, 256; 3×3, 256; 1×1, 1024]×36
conv5_x	7×7	[3×3, 512; 3×3, 512]×2	[3×3, 512; 3×3, 512]×3	[1×1, 512; 3×3, 512; 1×1, 2048]×3	[1×1, 512; 3×3, 512; 1×1, 2048]×3	[1×1, 512; 3×3, 512; 1×1, 2048]×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Рисунок 1.7 — Архітектура мережі *ResNet* [63]

1.4.3 Нейронна мережа PSPNet

PSPNet [72] мережа призначена для виконання сегментації зображень. Основною перевагою мережі є можливість виділяти складні ознаки контексту зі сцени. В даній архітектурі вводиться модуль пірамідних об’єднань, який розширює можливості мережі виділяти малі ознаки на рівні пікселів, що забезпечує більш високу точність розпізнавання. Даний модуль дозволяє усунути проблеми такі, як неспівпадаючі взаємозв’язки, подібність категорій та непомітні класи. Модуль пірамідних об’єднань розділяє карту ознак на чотири рівні масштабування 1 × 1, 2 × 2, 3 × 3 та 6 × 6. Потім формується об’єднане представлення для чотирьох різних масштабів. За рахунок цього, отримана інформація із вхідних даних є більш інформативною у порівнянні зі стандартним глобальним об’єднанням. Архітектура *PSPNet* використовую мережу *ResNet* для первинної обробки вхідних даних, після чого отримана карта ознак передаються далі у модуль пірамідних об’єднань.

1.4.4 Нейронна мережа MobileNet

MobileNet [71] архітектура глибокої нейронної мережі спеціально розроблена для використання в контексті периферійних обчислень на портативних пристроях та прискорювачах. Основними компонентами мережі є згортки із обмеженою глибиною, при якій до кожного каналу застосовується лише один фільтр. Далі згортки розміру 1 × 1 об’єднують результати попереднього кроку. Подібна двокрокова згортка зменшує обчислювальну складність, що є важливим фактором для вбудованих пристроїв. Для подальшого зменшення мережі, розробниками передбачено можливість зміни гіперпараметрів множників розміру. Модель *MobileNet* може використовуватись для задач детекції, локалізації об’єктів, розпізнавання та інших задач комп’ютерного зору.

1.5 Набори даних для навчання нейронної мережі

Навчання нейронної мережі потребує великої кількості навчальних даних. Наразі доступна велика кількість загальних та спеціалізованих наборів даних призначених під різні застосунки. Для задач комп’ютерного зору одними зі стандартних наборів даних, які широко використовуються при роботі із глибокими нейронними мережами, є *MNIST* [73], *ImageNet* [74] та спеціалізовані набори даних для розпізнавання дорожнього і міського оточення *Cityscapes* [75] та медичного діагностування рентген знімків *Chexpert* [76].

1.5.1 Набір даних MNIST

MNIST [73] це великий набір даних рукописних цифр 28 × 28 (рис. 1.8), який став де факто стандартом для тестування точності роботи нейронних мереж у задачах класифікації та інших систем комп’ютерного зору. Даний

набір даних є похідною від більшого за розміром повного набору *NIST Special Database*, що містить рукописні цифри, великі та малі літери. Загалом набір даних містить близько 60 000 зображень, що діляться на підмножини для тренування, перевірки та тестування.

1.5.2 Набір даних ImageNet

Ще одним великим набором даних для задач класифікації та локалізації є *ImageNet* [74] (рис. 1.9). Даний набір містить 1 000 класів, організованих за семантичною організацією *WordNet*. Великі обсяги та різноплановість *ImageNet* дозволяє створювати більш складні та точні моделі для різних задач комп’ютерного зору. Загалом набір має різні варіації і містить більше 1 000 000 зображень для тренування моделей.

Рисунок 1.8 — Набір даних *MNIST* [73]

Рисунок 1.9 — Набір даних *ImageNet* [74]

1.5.3 Набір даних Cityscapes

Також для спеціалізованих застосунків глибоких нейронних мереж наявні набори даних, призначені для більш вузько направлених задач комп’ютерного зору. Одним із таких спеціалізованих наборів є *Cityscapes* [75] для сегментації дорожнього та міського оточення (рис. 1.10). Набір містить велику кількість точно-анотованих зображень – 5 000, що були записані у трьох містах Німеччини. Додатково, набір включає 20 000 грубо-анотованих зображень для 23 інших міст. Отримані зображення розподілені по 30 класам, до яких відносяться: пішоходи, автомобілі, дорога, дорожні знаки, дерева та ін.

Рисунок 1.10 — Набір даних *Cityscapes* [75]

1.5.4 Набір даних CheXpert

CheXpert [76] спеціалізований набір даних для застосування глибоких нейронних мереж у медичному контексті. Набір містить більше 200 000 рентген зображень грудної клітини і дає можливість виявляти 14 аномалій на рентген знімках пацієнтів (рис. 1.11). Для перевірки точності роботи моделі окремо підготовлено 200 знімків, опрацювання яких виконувались сертифікованими радіологами.

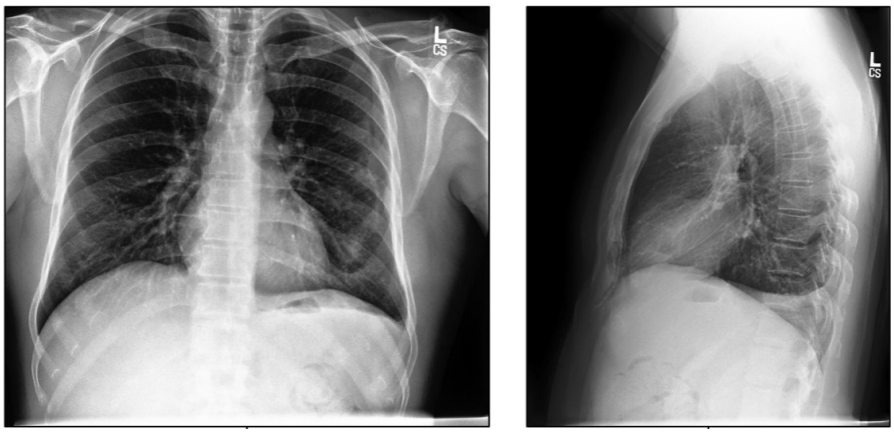

Рисунок 1.11 — Набір даних CheXpert [76]

1.6 Фреймворки для роботи із глибокими нейронними мережами

Використання моделей нейронних мереж наразі значно полегшується завдяки широкому розповсюдженню фреймворків для штучного інтелекту. Одними із популярних з них є *TensorFlow* [77] і його модифікація для периферійних обчислень *TF Lite* [78], *PyTorch* [79], *Caffe* [80], *MXNet* [81] та інші.

Також широкого розповсюдження отримують спеціалізовані прискорювачі. Так наприклад, компанія *Intel* випускає спеціалізований прискорювач *Neural Compute Stick* [3], що базується на процесорі *Movidius Myriad* [43]. Для роботи моделей нейронних мереж на даному прискорювачі використовується *Intel Movidius Neural Compute SDK – NCSDK* [82], при цьому заявлена підтримка моделей розроблених на фреймворках *TensorFlow*, *Caffe* та інші після їх конвертації засобами *OpenVINO* [83]. Обробка даних нейронними мережами на тензорних ядрах, що доступні на нових графічних процесорах серії *RTX* 3000 від компанії *NVIDIA,* також потребує спеціалізованого програмного забезпечення *TensorRT* [84], що підтримує фреймворки *PyTorch* та *TensorFlow*. Для процесорів від компанії *AMD* доступний спеціалізований інструментарій розробника для роботи з нейронними мережами – *ZenDNN* [85] та *ROCm* [86]. Інші наявні спеціалізовані прискорювачі для периферійних обчислень такі, як *Coral Edge*

TPU [2], *Horned Sungem* [4], *Huawei HiKey SoC* [5] також потребують певної конвертації та адаптації моделей нейронних мереж для роботи на даних пристроях. *Coral Edge TPU* від компанії *Google* використовує додаткові бібліотеки для мови *Python* – *PyCoral*, що надають інтерфейси для завантаження моделей у прискорювач та виконання обчислень. Зазначений прискорювач підтримує моделі розроблені у фреймворку *TF Lite* завдяки їх обробці у спеціалізованому компіляторі – *EdgeTPU Compiler* [87]. Ще один прискорювач *Horned Sungem* також може виконувати обчислення різних моделей нейронних мереж за умови їх попередньої конвертації у підтримуваний формат. Незважаючи на те, що даний прискорювач базується на процесорі *Intel Movidius*, розробники використовують власний інструментарій *SungemSDK* [88] для роботи із нейронними мережами. Нещодавно компанія *Huawei* представила свій прискорювач нейронних мереж на базі *HiKey SoC* та інструментарій розробника для роботи із нейронними мережами *HiAI SDK* [89].

1.7 Висновки до розділу 1

В даному розділі було розглянуто сучасні технології зменшення глибоких нейронних мереж для підвищення продуктивності та ефективності обробки даних. До таких технологій відносяться методи прунінгу та квантизації, що дозволяють зменшувати розмір та обчислювальну складність моделей глибоких нейронних мереж при цьому мінімізуючи втрати точності розпізнавання у порівняні із повноцінними початковими моделями.

Подальше збільшення продуктивності можна досягти шляхом використання прискорювачів зі спеціалізованою обчислювальною архітектурою для глибоких нейронних мереж. Дані спеціалізовані прискорювачі *TPU*, *VPU* та *NPU*, у порівнянні із графічними прискорювачами *GPU*, мають спеціальні обчислювальні модулі, що забезпечують більшу ефективність та продуктивність обробки даних за допомогою глибоких

нейронних мереж. Головним обмеженням подібних спеціалізованих прискорювачів є необхідність у спеціальній адаптації моделей глибоких нейронних мереж шляхом конвертації архітектури мережі у формат підтримуваний конкретним пристроєм або зменшення розміру мережі, оскільки пам’ять *TPU* є значно меншою у порівняні із звичайними *GPU*. Особливо, використання подібних прискорювачів є актуальним в сфері периферійних обчислень – *edge computing*, де присутні більш високі обмеження на обчислювальні ресурси та енергоефективність.

Враховуючи вище сказане, доцільним є розробка методу адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур, що дозволить збільшити продуктивність та ефективність обробки даних нейронними мережами за рахунок застосування спеціалізованих прискорювачів, при цьому зберігаючи точність оригінальної мережі.

РОЗДІЛ 2
ОСОБЛИВОСТІ АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ ДЛЯ СПЕЦІАЛІЗОВАНИХ ОБЧИСЛЮВАЛЬНИХ АРХІТЕКТУР

Після проведення огляду сучасних методів прунінгу та квантизації для зменшення глибоких нейронних мереж було виявлено, що більшість досліджень при роботі із нейронними мережами зосереджені на використанні одного з методів, або прунінгу, або квантизації. Незважаючи на інтенсивні дослідження [90], результати поєднання обох методів для адаптації та зменшення нейронної мережі залишаються недостатньо вивченими. Також, більшість сучасних досліджень обмежуються підвищенням продуктивності моделі глибокої нейронної мережі лише за допомогою зазначених вище методів, не покращуючи при цьому ефективність самої обчислювальної системи, до якої відноситься апаратна та програмна складова.

На основі зазначених вище фактів, для адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур було поставлено наступні задачі:

- Дослідити методи прунінгу та квантизації для зменшення глибоких нейронних мереж;
- Провести аналіз ефективності роботи апаратного забезпечення зі спеціалізованою архітектурою *TPA*, а саме *Coral Dev Board – Edge TPU* із підтримкою *TensorFlow* і *TensorFlow Lite*, та *Horned Sungem*, що базується на чипі *Intel Movidius*, для пришвидшення обробки даних моделями глибоких нейронних мереж у контексті *Edge Computing*;
- Дослідити вплив апаратного та програмного забезпечення цільової системи, із якою використовуються портативні прискорювачі *TPU*, на продуктивність та ефективність обробки даних моделями глибоких нейронних мереж в цілому;

- Розробити комплексний метод адаптації глибоких нейронних мереж для різних сфер застосування в контексті *Edge Computing*;
- Розробити портативний застосунок діагностики легеневих аномалій за даними рентген знімків для оцінки ефективності роботи спеціалізованих прискорювачів *TPU* в практичних умовах.

Відповідно до запропонованого комплексного методу адаптації глибоких нейронних мереж можна використовувати комплексний підхід, що поєднує методи прунінгу, квантизації, видалення зайвих класів об’єктів, підвищення ефективності апаратного та програмного забезпечення системи, до якої підключений прискорювач *TPU*.

2.1 Обчислювальна складність глибоких нейронних мереж

В даній роботі будуть розглядатись глибокі нейронні мережі, що навчаються за принципом «навчання із вчителем» для задач комп’ютерного зору. Даний принцип полягає у тому, що параметри мережі – ваги оптимізуються за рахунок опрацювання спеціального набору даних із заздалегідь визначеними мітками – класами. Набір даних ділиться на певні частини: тренувальна, валідаційна, тестова. Основною метою навчання є як можна точніше визначити ймовірність відношення певних даних до певного класу. Процес навчання продовжується певну кількість епох – повний прохід по датасету, поки рівень точності моделі не перестане зростати. Вхідні дані представляються у вигляді 4-и вимірного тензору $N \times C \times H \times W$, де N – розмір батчу (порція даних); C – кількість кольорів у даних (наприклад, для *RGB* – 3); H та W – висота та ширина. Загалом цей тензор представляє певну порцію зображень.

В задачах комп’ютерного зору найчастіше використовуються нейронні мережі, що містять згорткові шари *CNN*. В цих шарах нейрони згруповані по певним рівням – фільтрам, що дозволяють виокремлювати різні ознаки із вхідних даних. Кількість обчислень, що виконуються у кожному згортковому

шарі, залежить від різних факторів таких, як кількість каналів в шарі, розміру зображення, зв’язок із іншими шарами. Операція згортки виконується для кожного фільтру розмірності $C_{\text{in}} \times K_h \times K_w$, де C_{in} – кількість каналів вхідного тензору, K_h та K_w – розмірність фільтру. Обчислювальна складність згорткового шару розраховується за формулою:

$$W_{conv} = O\left(N \cdot C_{out} \cdot C_{\text{in}} \cdot H_i \cdot W_i \cdot K_h \cdot K_w\right), \tag{2.1}$$

де C_{out} – кількість фільтрів у згортковому шарі, *i* – номер попереднього шару з якого отримані дані. Слід зауважити, що на розмірність $H_i \times W_i$ також можуть впливати використання додаткових операцій таких, як крок із пропуском – *striding*, вирівнювання – *padding* та розширення – *dilating*.

Шар об’єднання – *pooling* зменшує розмір вхідного тензору на кількість каналів вхідного тензору. Це дозволяє зменшити розмір моделі, а його використання після згорткових шарів покращує виділення моделлю високорівневих ознак, яке відноситься до виокремлення більш великих частин у вхідних даних. Використання даного шару після згорткового шару збільшує кількість обчислень на W_{pool}, але глобально дозволяє зменшити обчислення на кількість каналів. Шар об’єднання виконує операції над суміжними регіонами – каналами розмірності $H_i \times W_i$ і може обчислювати максимальне – *max pooling* або середнє – *average pooling*. Даний шар має наступну обчислювальну складність:

$$W_{pool} = O\left(N \cdot C_{\text{in}} \cdot H_i \cdot W_i\right). \tag{2.2}$$

Шар нормалізації – *batch normalization* виконує нормалізацію даних в одній порції відносно нульового середнього та одиничної дисперсії. Це зменшує коваріаційний зсув та покращую сходження під час навчання. Обчислювальна складність залежить від розмірності вхідного тензору:

$$W_{bn} = O\left(N \cdot C_{\text{in}} \cdot H_i \cdot W_i\right). \tag{2.3}$$

Повно-зв'язний шар реалізує операцію матричного добутку і має обчислювальну складність:

$$W_{fcn} = O\left(C_{out} \cdot C_{\text{in}} \cdot N\right). \tag{2.4}$$

В загальному вигляді глибокі нейронні мережі представляють собою послідовність шарів, що обробляють результати, отримані із попередніх шарів. В деяких архітектурах таких, як *ResNet* [63], шари мережі можуть, окрім результатів попередніх шарів, отримувати результати від більш віддалених шарів. Загальна обчислювальна складність глибокої нейронної мережі включає в себе обчислення великої кількості шарів із врахуванням складних взаємозв’язків та потоків даних. Оскільки шари глибоких нейронних мереж оперують 4-и вимірними тензорами і велика кількість операцій зосереджена в шарах, це дозволяє реалізовувати паралельне виконання на рівні шарів. В цьому сенсі, оцінити кількість операцій для одного оператору глибокої мережі вкрай важко. Не зважаючи на це, робилися спроби оцінити продуктивність для оптимізованих реалізацій множення матриць [91], для різних розмірів порції даних при використанні прискорювачів *GPU* [91] та *CPU* [92, 93] в розподілених обчислювальних інфраструктурах. Інші спроби оцінити продуктивність моделі глибокої мережі базувались на розрахунку кількості операцій у мережі і мали похибку 10-30% [94]. Також були спроби оцінити вимоги по обміну даними для згорткових шарів та шарів об’єднання [95]. Однак в цілому, оцінити обчислювальну складність глибокої нейронної мережі є доволі важко, враховуючи внутрішню структурну складність більшості сучасних глибоких мереж. В подальшому це ускладнюється наявністю різних архітектур апаратного забезпечення із підтримкою паралелізму, наприклад такі, як *TPA*.

З практичної точки зору, наявність внутрішнього паралелізму даних у *TPA* дає можливість пришвидшити обробку даних глибокими нейронними мережами за рахунок збільшення розміру порції даних та оцінити його вплив на час навчання та виконання мережі. Той факт, що більшість операторів у шарі глибокої мережі є незалежними від кількості даних у порції даних, наприклад, зображень у згорткових мережах, це дозволяє виконувати розпаралелювання обчислень за рахунок розподілення обчислень певної

порції даних між різними обчислювальними ресурсами таких, як ядра або пристрої *GPU* та *TPU*, поди, хости, тощо.

2.2 Периферійні обчислення

Периферійні обчислення – *edge computing* набувають широкого розповсюдження та наразі забезпечують функціональність, подібну до парадигми хмарних обчислень. Основна різниця цих двох парадигм полягає у рівні на якому фактично відбувається обробка даних. Обробка даних у парадигмі периферійних обчислень відбувається безпосередньо на кінцевих пристроях до яких надходять дані із сенсорів, актуаторів, тощо. На противагу, за парадигмою хмарних обчислень, отримані дані передаються засобами локальної мережі на обчислювальні вузли, які зазвичай більш віддалені від кінцевих пристроїв.

Використання парадигми периферійних обчислень дозволяє усунути необхідність передачі даних з метою їх зберігання або обробки, що дає наступні переваги:

- більш швидка обробка та аналіз даних, зокрема у реальному часі;
- забезпечення вимог захисту даних та приватності;
- зменшення витрат на побудову та обслуговування системи;
- висока енергоефективність;
- низькі витрати пропускної здатності мережі;
- висока масштабованість;
- висока автономність.

Враховуючи ці переваги, використання засобів глибокого навчання в даній парадигмі є доволі перспективним. В рамках парадигми периферійних обчислень з’являються нові прискорювачі типу система на чипі *SoC*, що спеціалізуються на більш ефективній роботі глибоких нейронних мереж на периферійних пристроях. Дані периферійні прискорювачі *TPU*, *NPU*, *VPU*, завдяки оптимізованій внутрішній архітектурі, забезпечують високу

продуктивність та ефективність при роботі із глибокими нейронними мережами різного типу. Останнім часом з’явилася велика кількість периферійних прискорювачів із відповідними *SDK*, включаючи *NVIDIA Jetson Nano* [1], *Google Coral* [2], *Intel Movidius* [3], *Horned Sungem* [4], *HiSilicon Kirin 970* [5]. Але зазвичай дані прискорювачі мають обмежені обчислювальні ресурси, що не дозволяє в повній мірі використати потенціал глибоких нейронних мереж через їх вкрай високі вимоги до обчислювальних ресурсів, пам’яті, енергії, тощо. Через це існує потреба у розробці методів та засобів адаптації глибоких нейронних мереж для їхнього використання у парадигмі периферійних обчислень. В цьому напрямі проводились декілька досліджень, пов’язаних із комбінуванням обчислювальних парадигм у гібридні хмарно-периферійні архітектури [96].

Окрім обчислювальних обмежень, при використанні нейронних мереж на периферійних пристроях, додатково присутні накладні витрати пов’язані із пропускною здатністю каналів вводу/виводу, підтримкою програмного забезпечення в рамках певної ОС, обмеженнями при обміні даними. Ці накладні витрати рідко оцінюються при застосуванні нейронних мереж в реальних практичних умовах, але їх присутність може призводити до значного зниження продуктивності роботи системи. В цьому сенсі, правильна оцінка ефективності та прогнозування продуктивності периферійних систем при їхньому використанні в практичних умовах має велике значення. Незважаючи на наявність заявленої продуктивності периферійних обчислювальних пристроїв від виробника та певних тестів [8, 97], вплив додаткових накладних витрат на периферійну систему, враховуючи різні варіанти конфігурації та застосування різних периферійних обчислювальних пристроїв, залишається не достатньо дослідженим на даних момент.

2.3 Підвищення ефективності глибоких нейронних мереж

Зменшення витрат пам’яті та підвищення енергоефективності глибоких нейронних мереж дає можливість використовувати дані моделі мереж на периферійних та портативних пристроях. Для цього використовуються методи, що зменшують розмір мережі та дозволяють зберегти точність, близьку до оригінальної моделі [15]. Спочатку зайві зв’язки у мережі прибираються за допомогою прунінгу, далі виконується оптимізація ваг мережі при якій декілька зв’язків використовують спільні ваги. Це дозволяє використовувати пам’ять для зберігання даних нейронної мережі більш ефективно за рахунок зменшення кількості унікальних ваг. Відомі дослідження, пов’язані із покращенням часу виконання моделей нейронних мереж за рахунок прунінгу ваг та зменшення розмірів мережі, [11-15, 30-40, 98-103], а також деякі адаптивні підходи [104-107].

Оскільки після зменшення моделі нейронної мережі шляхом прунігу, змінюється лише частина, пов’язана із даними які опрацьовуються, наступним кроком в адаптації мережі для спеціалізованих пристроїв є зменшення обчислювальної складності мережі за рахунок використання внутрішніх архітектурних можливостей цільових прискорювачів. До цього можна віднести квантизацію як метод зміни представлення даних у мережі з представлення із плаваючою комою *FP32* у цілочисельне представлення *INT8*. При цьому, перевага полягає не лише у зменшенні витрат пам’яті на зберігання самих даних моделі нейронної мережі, а й можливість переходу до цілочисельної арифметики, що дозволяє значно зменшити обчислювальну складність, пришвидшити роботу моделі та покращити енергоефективність в цілому. Дана зміна представлення мінімальним чином впливає на втрати точності. Подальше зменшення представлення дозволяє обмежитись лише бінарними операціями.

В рамках *Edge Computing* важливим фактором, який впливає на продуктивність глибоких нейронних мереж є апаратна та програмна складова

системи, на якій застосовується прискорювач *TPU*. При неефективному налаштуванні системи всі переваги від методів прунінгу та квантизації для підвищення продуктивність нейронних мереж можуть бути втрачені через низьку пропускну здатність інтерфейсу підключення прискорювача або через використання несумісного програмного забезпечення. Тому, при адаптації та підготовці нейронної мережі для роботи на цільовій системі з метою досягнення максимально можливої продуктивності, доцільно приділити увагу конфігурації та налаштуванню системи, на якій буде працювати прискорювач нейронних мереж.

Окрім зазначених вище підходів адаптації нейронних мереж, ще одним важливим фактором, який впливає на продуктивність роботи моделі нейронної мережі, є якісний та точний набір даних великого обсягу, що максимально покриває задачу, яку буде вирішувати нейронна мережа. Наразі доступні великі набори даних загального призначення, але їх застосування в конкретних вузько направлених задачах дає можливість налаштувати мережу для виявлення лише загальних ознак, характерних певним об’єктам. При цьому глибші рівні мережі, що виділяють складніші ознаки, характерні для конкретного призначення мережі, залишаються слабо навчаними. Такий процес налаштування мережі називається її тюнінгом або додатковим навчанням під конкретне застосування і потребує спеціалізованих наборів даних. Оскільки підготовка таких наборів є досить кропітким процесом, що потребує багато часу, необхідно чітке визначення класів, з якими працюватиме модель. Кількість цих класів впливатиме як на складність підготовки набору даних, так і на складність самої мережі. Більша кількість класів може потребувати збільшення кількості шарів мережі, тому видалення зайвих неважливих класів може дозволити зменшити загальну структуру мережі і збільшити її продуктивність.

2.4 Метрики оцінки продуктивності роботи моделі

Основними метриками, що використовуються при роботі із нейронними мережами, є точність – *accuracy*, втрата – *loss* та час виконання – *inference time*. Ці три базові показники дозволяють оцінити продуктивність обробки даних моделлю мережі при навчанні та тестуванні на певному наборі даних. Оскільки в даній роботі використовуються методи для модифікації глибоких нейронних мереж, додатково було введено метрики, що дозволяють оцінити рівень зменшення моделі та її прискорення, а саме:

- відносне зменшення розміру моделі мережі:

$$Z_i = N_i / N, \tag{2.5}$$

де N – розмір оригінальної моделі мережі, N_i – розмір моделі після *i*-ітерації прунінгу;

- прискорення після ітерації прунінгу:

$$S_i = t_i / t_0, \tag{2.7}$$

де t_0 – час обробки даних для оригінальної моделі мережі, t_i – час обробки даних для моделі після *i*-ітерації прунінгу.

Незважаючи на широке розповсюдження нових тензорних обчислювальних архітектур, зокрема периферійних прискорювачів нейронних мереж, вони є пропрієтарними розробками без детальної інформації про їх внутрішню організацію. В даних спеціалізованих архітектурах можуть бути присутні накладні витрати на ініціалізацію моделі мережі та завантаження даних, які важливо враховувати при роботи з ними. Тому, доцільно вводити додаткові проміжки при замірі часу обробки даних, щоб оцінити загальний вплив фази ініціалізації пристрою на продуктивність нейронної мережі:

- час обробки даних із накладними витратами:

$$t_{overhead}^{inf} = t_i^{inf} / N, \tag{2.8}$$

де $i = 1, 2$ - для 1-ї та 2-ї ітерації; N - кількість елементів у наборі даних;

- час обробки даних без накладних витрат:

$$t^{inf} = t_i^{inf} / N, \tag{2.9}$$

де $i > 2$ - для ітерації 3 і наступних.

Окрім метрик для розрахунку точності та продуктивності моделі мережі важливо оцінювати час, за який буде проходити навчання мережі. Це важливо в умовах адаптивного додаткового навчання мережі при виконанні прунінгу. Ідея прунінгу полягає у зменшенні мережі, через що здатність мережі виокремлювати нову інформації не падає у значній мірі і попередня точність розпізнавання може бути відновлена у процесі додаткового навчання. Враховуючи, що ітерацій додаткового навчання може бути велика кількість, в залежності від обраних гіперпараметрів та швидкості стискання мережі, загальний час додаткового навчання може бути навіть більший ніж час простого навчання оригінальної мережі. Для оцінки часу роботи моделі мережі використовується наступна формула часу як функції від гіперпараметрів:

$$t_{regime} = f_{regime}(s, b, i), \tag{2.10}$$

де *regime* – час обробки даних для режиму навчання (*train*) або роботи (*inf*); s – розмір зображення у форматі $H \times W$, $H=W$, H – висота, W – ширина; b – розмір порції даних від 8 до максимального розміру для якого вистачає внутрішньої пам’яті обчислювального пристрою; i – номер ітерації для одного з двох типів обчислювального пристрою D, $D=G$ для графічного прискорювача та $D=T$ спеціалізованого прискорювача *TPU*.

2.5 Застосування фреймворків для роботи із глибокими нейронними мережами на спеціалізованих обчислювальних архітектурах

Застосування фреймворків для нейронних мереж значно спрощує процес розробки моделей нейронних мереж, а також надає можливість прискорення нейронних мереж за рахунок їхнього виконання на високопродуктивних прискорювачах таких, як *GPU*. Кожний із розглянутих

фреймворків підтримується своїм співтовариством і розробниками, що роблять реалізації сучасних архітектур нейронних мереж під дані фреймворки.

Важливим недоліком є відсутність взаємної сумісності фреймворків для нейронних мереж між собою. Це означає, що певна модель мережі розроблена під конкретний фреймворк не може бути напряму перенесена і працювати в іншому фреймворку без певної конвертації. Подібна проблема відсутності взаємної сумісності є досить актуальною. При подальшому розгляді, дана проблема ще більше ускладнюється у зв’язку із розповсюдженням і використанням спеціалізованих прискорювачів для нейронних мереж, особливо в контексті периферійних обчислень. Кожен із подібних прискорювачів підтримуються своїм розробником, який випускає під нього власний інструментарій розробника – *SDK*, а також розробляє певні розширення для роботи із певними фреймворками.

Враховуючи, наведені вище фактори про фреймворки, інструментарії та прискорювачі, можна побачити, що значної складності при роботі із нейронними мережами додає відсутність уніфікованості при роботі із різним програмним та апаратним забезпеченням. Кожен розробник надає свій спеціалізований інструментарій для роботи із мережами, що підтримується певним набором фреймворків. Даний фактор ускладнює процес застосування або переносу моделей, розроблених на інших фреймворках. Також, при виборі певного прискорювача нейронних мереж, слід враховувати його можливості, оскільки певні операції шарів нейронної мережі можуть апаратно не підтримуватись.

2.6 Глибокі нейронні мережі та спеціалізовані прискорювачі

Узагальнюючи вище сказане, було прийнято рішення дослідити роботу різних архітектур нейронних мереж на різних прискорювачах таких, як

NVIDIA GPUs, спеціалізованих прискорювачах *Coral Edge TPU* та *Horned Sungem*, використовуючи фреймворки *TensorFlow* та *TF Lite*.

В роботі були використані наступні архітектури глибоких нейронних мереж, а саме: *ResNet50* та архітектура мережі подібної до *VGG-16* із модифікованим розміром вхідних даних 28 × 28 (рис. 2.1).

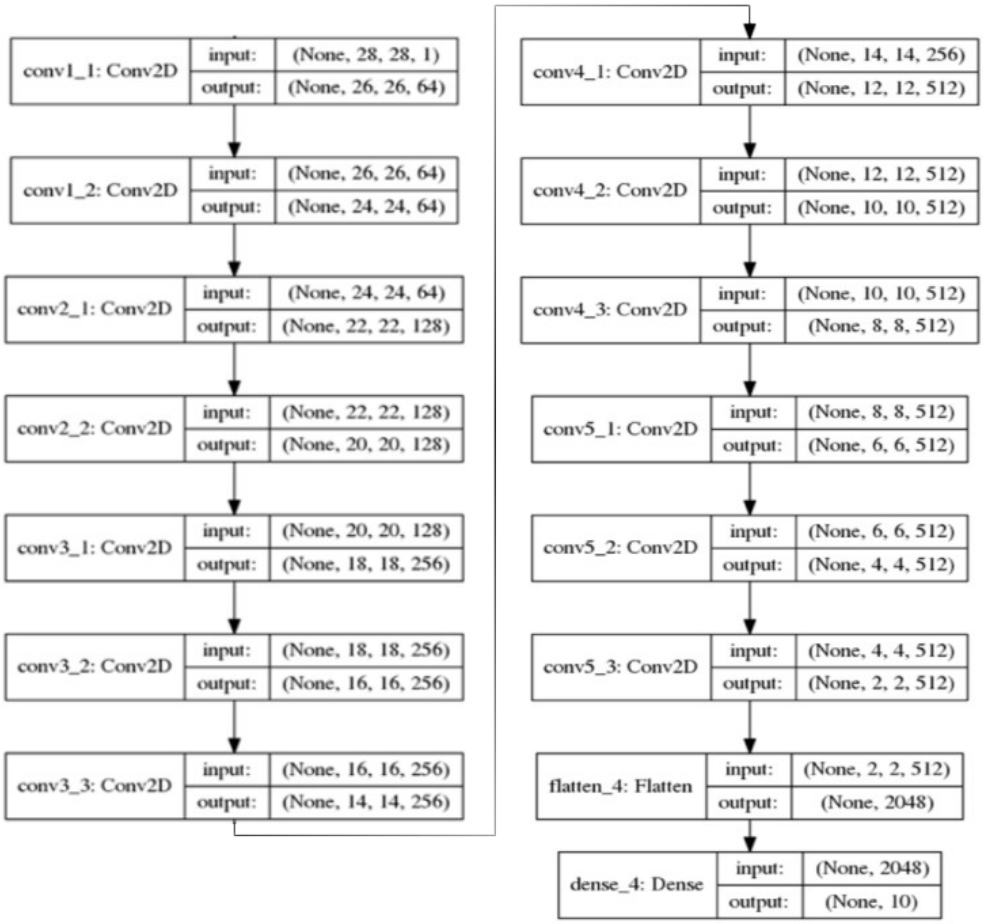

Рисунок 2.1 — Архітектура мережі подібної до *VGG-16* із модифікованим розміром вхідних даних

2.7 Висновки до розділу 2

В даному розділі було показано, що процес адаптації глибоких нейронних мереж, в тому числі із застосуванням спеціалізованих прискорювачів, є доволі багатогранним і зроблено теоретичне припущення, що цей процес потребує врахування різних факторів для збільшення

продуктивності, ефективності та точності обробки даних нейронними мережами. До таких факторів можна віднести:

- високу обчислювальну складність глибоких нейронних мереж, що містять згорткові шарів – *CNN* в своїй структурі, тому в процесі адаптації мережі доцільно зменшувати дані шари за рахунок прунінгу та видалення зайвих класів;
- необхідність враховувати внутрішню архітектуру спеціалізованих прискорювачів для нейронних мереж, які в більшій мірі працюють із цілочисельним представленням даних – *INT8*, що потребує зміну представлення даних у мережі шляхом її квантизації. При цьому, перехід до цілочисельної арифметики дозволяє значно зменшити обчислювальну складність, пришвидшити роботу моделі та покращити енергоефективність в цілому;
- підвищення ефективності процесу обробки даних глибокими нейронними мережами із використанням спеціалізованих прискорювачів, оскільки в них можуть бути присутні додаткові накладні витрати на ініціалізацію моделі мережі на пристрої, завантаження та обробку даних, тощо;
- конфігурація апаратної та програмної складової цільової системи на якій застосовується спеціалізований прискорювач, оскільки при неефективному налаштуванні системи всі переваги від методу адаптації нейронних мереж може бути втрачена через низьку пропускну здатність інтерфейсу підключення прискорювача до системи або через використання несумісного програмного забезпечення та інструментарію. Тому, доцільним є також підвищувати ефективність самої цільовій системі, на якій буде працювати прискорювач нейронних мереж, з метою збільшення продуктивності обробки даних.

Із наведених вище факторів, відомих теоретичних фактів і зроблених теоретичних припущень слідує, що для збільшення продуктивності та ефективності обробки даних при адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур необхідно розробити комплексний метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур, що включає в себе:

- метод адаптації моделі глибокої нейронної мережі за рахунок її зменшення та зміни представлення даних;
- метод підвищення ефективності процесу обробки даних нейронною мережею на спеціалізованих прискорювачах, враховуючи їх технічні особливості роботи;
- метод підвищення ефективності програмної та апаратної складової цільової системи для глибоких нейронних мереж.

РОЗДІЛ 3
КОМПЛЕКСНИЙ МЕТОД АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ ДО АПАРАТНОГО ЗАБЕЗПЕЧЕННЯ ЗІ СПЕЦІАЛІЗОВАНОЮ АРХІТЕКТУРОЮ

За результатами проведеного дослідження та згідно поставленого завдання було запропоновано комплексний метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур, що включає в себе:

1. Метод адаптивного ітеративного прунінгу для зменшення розміру моделі мережі та збільшення продуктивності обробки даних глибокими нейронними мережами;
2. Метод підвищення ефективності процесу обробки даних нейронними мережами, що враховує технічні особливості роботи спеціалізованих обчислювальних архітектур;
3. Метод підвищення ефективності інфраструктури для глибоких нейронних мереж із використанням спеціалізованих прискорювачів нейронних мереж.

Відповідно до запропонованого методу адаптації моделей глибоких нейронних мереж згорткового типу, можна використовувати комплексний підхід, що поєднує методи прунінгу, квантизації, видалення зайвих класів об’єктів, та підвищення ефективності апаратного та програмного забезпечення системи до якої підключений прискорювач *TPU*. Використання даного набору засобів адаптації дає можливість обробляти дані за допомогою моделей глибоких нейронних мереж із більшою продуктивністю та ефективністю.

3.1 Метод адаптивного ітеративного прунінгу

Запропонований метод адаптивного ітеративного прунінгу базується на методі занулення нейронів активації [99], при якому визначаються і

прибираються нейрони, які мінімально впливають на точність мережі. Після даного методу прунінгу мережа додатково навчається, при цьому ваги, що залишилися, ініціалізуються значеннями, отриманими з моделі до застосування прунінгу. З метою визначення кількості нейронів у каналах з активаціями близьких до нуля використовується метрика середнього відсотку нулів (*APoZ*). Дане значення розраховується для всіх шарів моделі, що дозволяє визначити зайві ваги, канали та шари. Далі виконується видалення зайвих ваг, каналів та шарів із певним порогом для значення *APoZ* – рівень прунінгу. Ця операція дозволяє значно зменшувати розмір моделі глибокої нейронної мережі із втратою точності. Щоб відновити початкову точність для моделі після прунінгу, виконується її додаткове навчання.

Даний адаптивний метод прунінгу є подібним до стандартного підходу прунінгу та додаткового навчання, за винятком адаптивної складової зміни гіперпараметрів (P_i) на кожній *i*-ітерації прунінгу, де P_i може бути розміром порції даних (B_i), кількістю епох додаткового навчання (R_i), мінімальною кількістю каналів у шарах (c_{min}), та інші. Дані гіперпараметри можуть змінюватись за адаптивними законами такими, як:

- Постійний

$$P_i = const, \tag{3.1}$$

для $\forall i$, де *i* — номер ітерації прунінгу;

- Лінійний

$$P_i = P_{i-1} + \alpha. \tag{3.2}$$

- Мультиплікативний

$$P_i = \beta P_{i-1}. \tag{3.3}$$

- Залежний від контексту

$$P_i = f\left(P_{i-1}, \nabla N_{i-1}, \nabla L_{i-1}, \nabla t_{i-1}, \ldots\right), \tag{3.4}$$

де α, β – параметри тюнінгу; ∇ – градієнт; N_{i-1}, L_{i-1}, t_{i-1} – розмір моделі, значення функції втрати та час виконання моделі на попередній ітерації прунінгу $i-1$ відповідно. Значення параметрів тюнінгу α, β та функції f може бути встановлено емпіричним шляхом на основі певних критеріїв таких, як

збільшення прискорення моделі, зменшення функції втрати, збільшення точності та інші.

На даному етапі обраним критерієм було збільшення прискорення, порівнюючи час виконання моделі нейронної мережі після ітерації прунінгу t_i та час виконання початкової моделі t_0. Щоб компенсувати зменшення точності після ітерації прунінгу, гіперпараметри розміру порції даних (B_i) та кількість епох (R_i) були обрані ключовими під час процесу додаткового навчання моделі мережі після прунінгу. Мінімальна кількість каналів в шарі (c_{min}) була фіксованою і дорівнювала 5.

З точки зору адаптивності методу було сформульовано наступні принципи, що керують зміною гіперпараметрів в ході прунінгу, в контексті використання модифікованих моделей на портативних спеціалізованих прискорювачах:

1. Зі зменшенням розміру моделі (N_i) на кожній ітерації прунінгу (i), максимально можлива порція даних (B_i) може бути збільшена. В свою чергу, це збільшує кількість даних – зображень, що обробляються паралельно;

2. Оскільки зменшення розміру моделі (N_i) та збільшення порції даних (B_i) покращує паралелізм, але призводить до зменшення точності роботи моделі (A_i), необхідно виконувати додаткові адаптивні кроки під час додаткового навчання для компенсації втрати точності A_i. Такі кроки включають в себе:

• Збільшення кількості епох додаткового навчання на кожній ітерації прунінгу;

• Зміна рівня прунінгу для більш повільного/швидкого зменшення нейронної мережі;

• Зміна мінімальної кількості каналів в шарі, що має залишатись після прунінгу;

• Зміна стандартних гіперпараметрів таких, як швидкість навчання, затухання навчання та інші.

Запропонований багатокроковий алгоритм адаптивного ітеративного прунінгу наведений на рис. 3.1 та включає в себе наступні основні кроки:

1. Ініціалізація моделі нейронної мережі, її первинне навчання та тестування точності;
2. Виконання адаптивного ітеративного прунінгу:

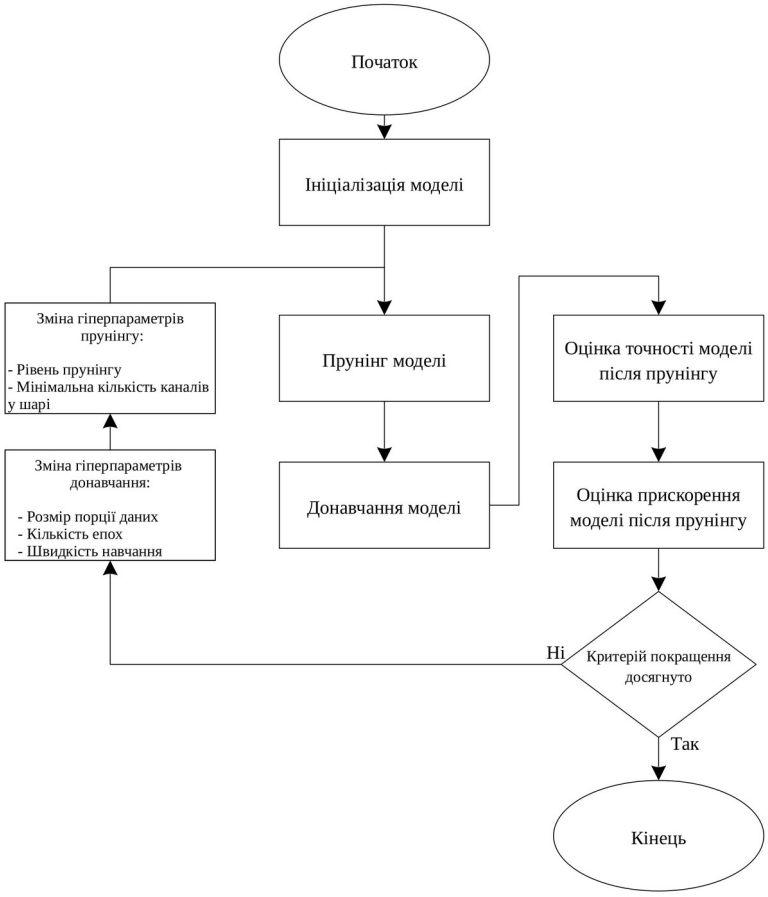

Рисунок 3.1 — Адаптивний ітеративний прунінг моделі глибокої нейронної мережі

• Прунінг моделі виконується із рівнем, який підбирається, враховуючи поточну модель мережі, набір даних, цільову точність та прискорення мережі;

• Додаткове навчання мережі із новими гіперпараметрами: розмір порції даних (B_i), який може бути збільшений через зменшення розміру моделі мережі (N_i); кількість епох додаткового навчання (R_i), яка може збільшуватись для компенсації зменшення мережі після кожної ітерації прунінгу; мінімальна кількість каналів у шарах мережі (c_{min}), що має

залишитись після прунінгу; швидкість навчання та рівень затухання, які розраховуються, щоб компенсувати втрати точності.

В ході тестувань було визначено, що обраний критерій – збільшення прискорення обробки даних моделлю мережі без значного зменшення точності, не може бути досягнуто, використовуючи постійний або лінійний закони. Використання більш складних законів таких, як мультиплікативний та адаптивний із врахуванням контексту, дає можливість досягти обраного критерію покращення. Збільшення прискорення обробки даних моделлю досягається наступними факторами:

- Зменшення розміру моделі нейронної мережі (N_i) на кожній ітерації прунінгу, що призводить до зменшення кількості обчислювальних операцій при обробці даних;
- Поступове збільшення розміру порції даних (B_i) до максимально можливого наряду зі зменшенням моделі покращує паралелізм нейронної мережі, оскільки збільшується кількість даних, що обробляються одночасно.

Недоліками наявних методів ітеративного прунінгу є занадто сильний або слабкий прунінг. В результаті чого, отримані моделі є не достатньо пришвидшеними або точними [99]. Розроблений метод адаптивного ітеративного прунінгу дає можливість нівелювати дані недоліки за рахунок адаптивної зміни зазначених гіперпараметрів на кожній ітерації прунінг, враховуючи поточні характеристики моделі нейронної мережі.

3.2 Метод підвищення ефективності процесу обробки даних нейронними мережами на спеціалізованих обчислювальних архітектурах

Спеціалізовані тензорні обчислювальні архітектури *TPA*, призначені для виконання обробки даних глибокими нейронними мережами, отримали широке розповсюдження за останні роки. При цьому, потрібно враховувати обчислювальну складність алгоритмічно різних компонентів нейронних

мереж для їх ефективного застосування на подібних тензорних архітектурах. Для дослідження та демонстрації кардинальної різниці між архітектурами *TPU* та *GPU*, а також алгоритмічно різних моделей нейронних мереж *VGG16* та *ResNet50*, було проведено аналіз роботи архітектур *TPU* та *GPU*. Результати аналізу дозволяють спрогнозувати прискорення та час обробки даних моделями нейронних мереж при різних значеннях гіперпараметрів, наприклад, розміру порції даних, та розміру зображень. Також, це дає можливість порівняти спеціалізовані тензорні прискорювачі *TPU* та графічні прискорювачі *GPU*, беручи до уваги їх принципово різні архітектури. Вплив гіперпараметрів та архітектурних особливостей роботи різних прискорювачів нейронних мереж можна описати наступним чином:

$$t_{regime} = f_{regime}(Dd, P(B, R, c, s)), \tag{3.5}$$

де *regime* – час обробки даних для режиму навчання (*train*) або роботи (*inf*); *d* – номер ітерації для одного з двох типів обчислювального пристрою *D*, *D=G* для графічного прискорювача та *D=T* для спеціалізованого прискорювача *TPU; P* – гіперпараметри нейронної мережі, до яких відносяться:

- *B* – розмір порції даних;
- *R* – кількість епох;
- c – кількість каналів у шарах мережі;
- s – розмір вхідного зображення.

Основні фактори, що впливають на ефективність обробки даних нейронними мережами наведені на рис. 3.2.

Одним із способів досягнення більшої продуктивності обробки даних глибокими нейронними мережами на прискорювачах *GPU* це збільшення розміру порції даних, що дозволяє компенсувати довгі за часом операції та інші накладні витрати, за рахунок використання паралелізму при обробці даних. Зазвичай подібні порції даних отримуються із повного набору даних в ході його перемішування, при цьому одна обробка набору даних називається епохою. Оскільки повний процес тренування нейронної мережі може займати

десятки і навіть сотні епох, обрання найбільшого розміру порції даних, що може опрацювати прискорювач, дає великий виграш у паралелізмі та продуктивності. Маючи основним критерієм отримання максимальної точності, обрання кращого розміру порції даних є компромісом між мінімальним та максимальним розміром порції даних. Це є складною оптимізаційною задачею. Емпірично доведено, що розмір порції даних має бути не сильно малим і не сильно великим, щоб досягнути сходження алгоритму навчання та високого узагальнення даних нейронною мережею [108-111].

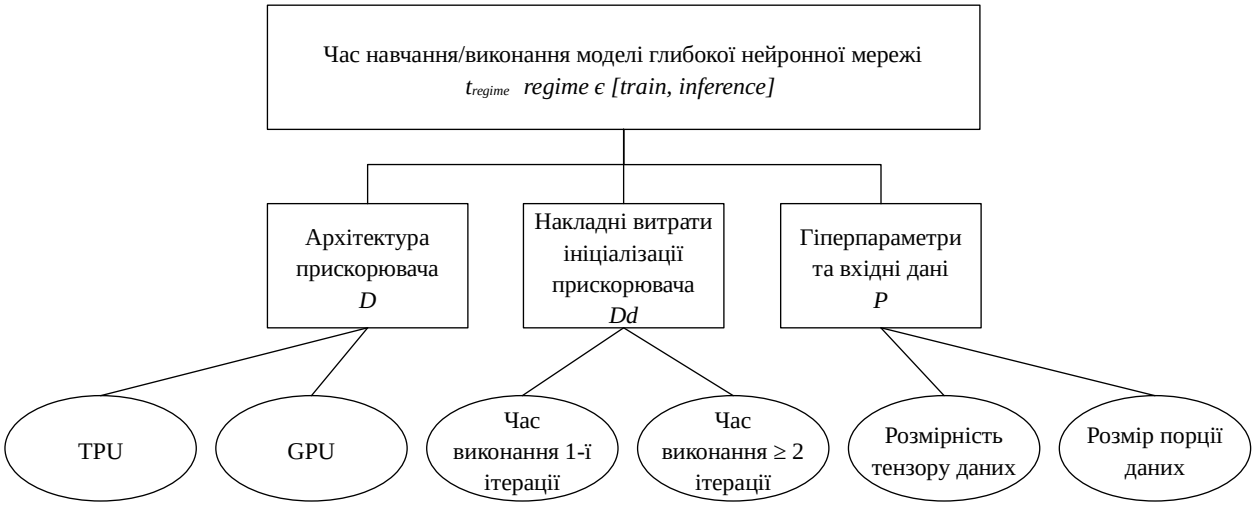

Рисунок 3.2 — Фактори, що впливають на ефективність обробки даних нейронними мережами

Більшість операцій у машинному навчанні моделюються як операції над тензорами. В свою чергу, архітектура нейронних мереж представляється у вигляді направленого ациклічного графу, у якому вершинами є певні обчислювальні операції, а ребрами – потік даних. Обчислення в подібному графі характеризуються двома основними параметрами:

- Обсяг роботи (W), чому відповідає загальна кількість вершин графа;
- Діаметр (D), що є максимальним шляхом між двома вершинами графа.

Зазвичай, ці параметри характеризують обчислювальну складність графа у паралельних системах і дозволяють приблизно оцінити час обчислень. Але враховуючи, що нейронні мережі застосовуються на певних обчислювальних архітектурах, наприклад *GPU* або *TPU*, їх внутрішня організація може додатково вносити свій вклад у час виконання обчислень. Тому, важливим фактором при роботі із глибокими нейронними мережами, особливо при їх адаптації, є можливість оцінити ефективність їх роботи при використанні певної обчислювальної архітектури.

Навчання та тестування нейронної мережі включає в себе велику кількість обчислень, в тому числі, попередня обробка вхідних даних, їх форматування, ініціалізація моделі, тощо. Всі ці операції можуть вносити суттєвий вплив у час обробки даних моделлю нейронної мережі і є додатковими накладними витратами. Обсяг цих накладних витрат може бути різним для різних обчислювальних архітектур, в тому числі спеціалізованих, і в деяких випадках, може бути невідомим в світлі пропрієтарності подібних архітектур.

Для демонстрації основних відмінностей між обчислювальними архітектурами *GPU*/*TPU* та фактичних показників обчислювальної складності для алгоритмічно різних моделей глибоких нейронних мереж було проведено аналіз масштабних залежностей часу виконання моделі та прискорення для обох обчислювальних архітектур при різних значеннях гіперпараметрів.

Аналіз проводився з використанням моделей *VGG16* та *ResNet50*, набору даних *MNIST* і включав наступні кроки:

- Заміри часу обробки даних моделлю нейронної мережі для 1 і 2-ї ітерацій, на яких присутні найбільші затримки, що пов’язані із ініціалізацією обчислювального пристрою та підготовкою даних;
- Заміри часу обробки даних моделлю нейронної мережі для 3-ї ітерацій, на якій відсутні затримки, що пов’язані із ініціалізацією обчислювального пристрою та підготовкою даних.

Параметри, що змінювались в ході аналізу:

- Обчислювальна архітектура – *GPU* / *TPU*;
- Режим роботи моделі нейронної мережі – навчання / тестування;
- Розмір вхідних даних – 28, 32 48, 64, 80, 96;
- Розмір порції даних – 8, 16, 20, 24, 28, 32, …, 112, 128, 160.

Із проведеного аналізу слідую, що:

- *TPU* має більші накладні витрати – затримку для обох режимів – навчання та тестування на 1-й та 2-й ітераціях, у порівнянні із відповідними ітераціями 1-2 для *GPU*. Ця затримка особливо суттєво велика на 1-й ітерації;
- *TPU* демонструє набагато більші затримки на першій ітерації, на відміну від наступних ітерації ≥ 3, при цьому великі затримки для *GPU* мають місце лише на 1-й ітерації;
- *TPU* показує більші затримки у всіх режимах – навчання та тестування для 1 і 2-ї ітерації, у порівнянні із 3-ю ітерацією та всіма ітераціями для *GPU*;
- Відмінності архітектур *GPU* та *TPU* виражаються у різних затримках ініціалізації під час підготовки даних та компіляції моделі. На початкових 1-2 ітераціях *GPU* має меншу затримку, через що продуктивність *TPU* на початкових ітераціях не вища за продуктивність *GPU*.
- На ітераціях ≥ 3 *TPU* демонструю значно менший час і більшу продуктивність обробки даних моделлю, у порівняні із *GPU*.

3.3 Метод підвищення ефективності інфраструктури обробки даних глибокими нейронними мережами

Незважаючи на значний прогрес у розвитку застосунків штучного інтелекту, глибокого навчання та периферійних обчислень, значні складнощі залишаються у процесі розгортання подібних застосунків та моделей

глибоких нейронних мереж на цільових інфраструктурах периферійних обчислень – *Edge Computing*. До таких інфраструктур можна віднести спеціалізовані прискорювачі, контролери, сервери, їх комбінації, тощо. Проблеми під час розгортання часто пов’язані із затримками та накладними витратами під час обробки даних моделями нейронних мереж на різних прискорювачах, енергоефективністю подібних застосунків, складнощами під час міграції та адаптації. Наразі існує багато методів, які дозволяють зменшити обчислювальні витрати нейронних мереж – прунінг та квантизація, збільшити їх продуктивність за рахунок використання спеціалізованих обчислювальних архітекутр, тощо. Враховуючи велику кількість наявних архітектур – *TPU*, *VPU*, *NPU* та можливі нові архітектури, швидка міграція моделей нейронних мереж на різне програмне та апаратне забезпечення залишається доволі складною задачею. Для її вирішення пропонувались різні рішення, що базуються на використанні віртуальних машин, контейнерів, та інших специфічних підходів [112-115]. Але вплив подібних організацій інфраструктур та їх накладні витрати, що впливають на продуктивність обробки даних нейронними мережами, особливо у контексті периферійних обчислень для конкретних застосунків, залишається слабо вивченим. Тому, пошук кращьої конфігурації обчислювальної інфраструктури для збільшення продуктивності обробки даних має значний практичний інтерес. Підвищення ефективності цільової обчислювальної інфраструктури, в загальному вигляді, можна представити у наступному символьному вигляді:

$$\text{is_opt} = f\left(HW, SW\right) \tag{3.6}$$

$$HW = f\left(D\left(com\right)\right) \tag{3.7}$$

$$SW = f\left(SDK, OS\right) \tag{3.8}$$

де f - невідома функція, яка може бути визначена емпірично; *HW* та *SW* — дві складові обчислювальної інфраструктури — апаратна та програмна відповідно; *com* — інтерфейс за яким підключається пристрій *D* до цільової системи; *SDK* (*Software Development Kit*) та *OS* (*Operating System*) є

основними складовими програмної частини, які прямо можуть впливати на продуктивність роботи побудованої обчислювальної інфраструктури.

В рамках периферійних обчислень важливим фактором, який впливає на продуктивність глибоких нейронних мереж в цілому, є апаратна та програмна складова системи, на якій застосовується прискорювач *TPU*. При неефективному налаштуванні системи всі переваги від методів прунінгу та квантизації для підвищення продуктивності нейронних мереж, а також переваги від використання самого спеціалізованого прискорювача, можуть бути втрачена через низьку пропускну здатність інтерфейсу підключення прискорювача до системи або через використання несумісного програмного забезпечення. Для виявлення зазначеної залежності було досліджено вплив конфігурації апаратного та програмного забезпечення системи на продуктивність прискорювача *TPU*. За результатами проведеного аналізу були виявлені основні фактори, налаштування яких дозволяє збільшити продуктивність глибоких нейронних мереж та спеціалізованих прискорювачів (рис. 3.3).

Рисунок 3.3 — Фактори цільової інфраструктури, які впливають на продуктивність нейронних мереж

Враховуючи велику кількість спеціалізованих прискорювачів нейронних мереж, було обрано два розповсюджених периферійних прискорювача – *ECA* для глибоких нейронних мереж *Google Coral EdgeTPU* та *Horned Sungem* на базі чипу *Intel Movidius*. Продуктивність даних прискорювачів було досліджено для різних конфігурацій цільової інфраструктури, враховуючи використання різного програмного забезпечення. Також до уваги бралась можливість застосування нейронних мереж на загальних прискорювачах типу *CPU* та *GPU*. Конфігурації інфраструктури, що досліджувались, наведено у таблиці 3.1.

Таблиця 3.1 — Конфігурації тестової інфраструктури

Hardware	GPU	Horned Sangem/ usb2	Horned Sangem/ usb3	Coral stick/ usb2/ standard	Coral stick/ usb3/ standard
Chip	NVIDIA 1080 Ti	Intel Movidius MA245x	Intel Movidius MA245x	Google Edge TPU	Google Edge TPU
OPs	0.6 TFLOPs	0.1 TOPs	0.1 TOPs	4 TOPs	4 TOPs
Memory	11 GB	512 MB	512 MB	8 MB	8 MB
I/O interface (rate)	PCIe 2.0 ×16 (64 Gbps)	USB 2.0 (480 Mbps)	USB 3.0 (5 Gbps)	USB 2.0 (480 Mbps)	USB 3.0 (5 Gbps)
OS	Linux	Linux	Linux	Linux/ Windows	Linux/ Windows
Overclocking	No	No	No	No/Max	No/Max

В якості операційних систем було обрано дві популярні: *Windows* 10 та *Ubuntu* 18. Для підключення спеціалізованих прискорювачів *TPU* використовувався інтерфейс вводу/виводу *USB* у двох версіях: 2.0 – із максимальною пропускною здатністю 480 *Mbps* та 3.0 – 5 *Gbps*. Для *Coral EdgeTPU* було протестовано два режими роботи: «без прискорення» – зі стандартною продуктивністю та «прискорення» – зі збільшеною

продуктивністю. За замовчуванням пристрій працює в режимі «без прискорення». Наведені комбінації параметрів було обрано для тестування, беручи до уваги можливість їх легкої зміни кінцевим користувачем, при цьому їх модифікація дає суттєве збільшення продуктивності обробки даних нейронними мережами.

В якості тестового застосунку було обрано задачу розпізнавання постаті людини, що є різновидом задачі комп’ютерного зору. В даній задачі виконується розпізнавання та встановлення положення кінцівок людини і взаємного поєднання між ними. Модель мережі *MobileNet* було використано в якості нейронної мережі для даної задачі. Розпізнавання проводилось на тестовому відео фрагменті різної роздільної здатності 1280×720, 640×480, 480×360, що складався із 389 кадрів (рис. 3.4). Для роботи із моделлю нейронної мережі використовувався *TensorFlow* та високорівневий інтерфейс *Keras*.

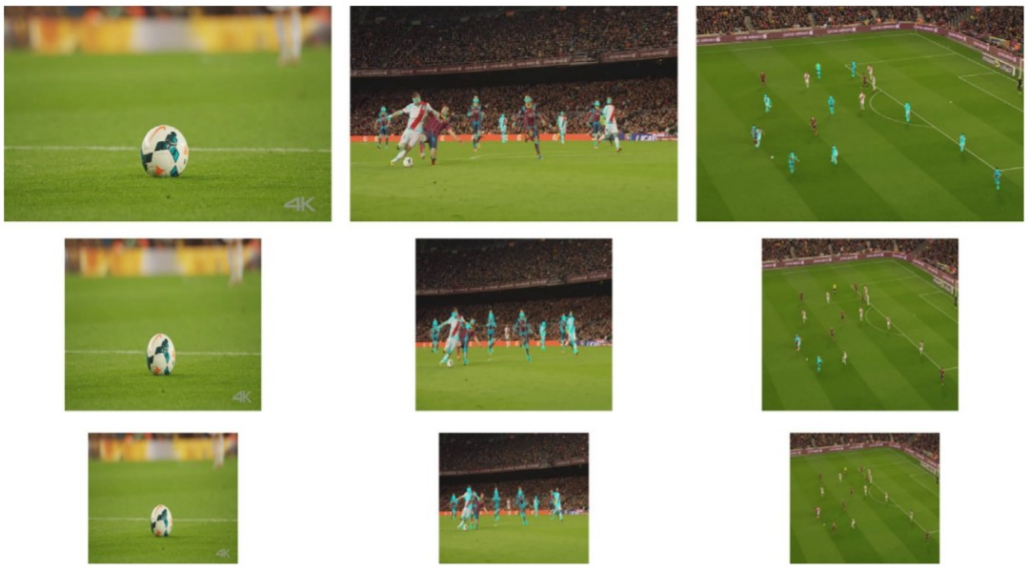

Рисунок 3.4 — Тестовий вхідний набір даних різного розміру

Для виконання аналізу було розроблено тестову програму, що виконую задачу детекції та зберігає час обробки зображення обраною моделлю нейронної мережі. В якості основних проміжків життєвого циклу моделі, обрано дві фази:

- Фаза виконання детекції, що виконує розпізнавання постаті людини, за допомогою моделі *MobileNet*;
- Фаза повної ітерації, що включає накладні витрати, пов’язані із завантаженням кадру зображення, його попередньою обробкою, детекцією та обробкою результату.

За результатами аналізу було досягнуто значної різниці в продуктивності при роботі з різними комбінаціями забезпечення системи. Зміна операційної системи на *Linux* дозволяє отримати прискорення у 2,13 рази, подальша зміна інтерфейсу підключення на *USB3* дає додаткові 4,04, а також використання спеціального режиму підвищеної продуктивності для *Coral Edge TPU* надає додаткові 4. Загалом було досягнуто прискорення до 8,7 раз при різних конфігураціях обчислювальної інфраструктури.

3.4 Комплексний метод адаптації глибоких нейронних мереж

Узагальнюючи отримані результати досліджень, було запропоновано багаторівневий комплексний метод адаптації глибоких нейронних мереж для різних сфер застосування в контексті *Edge Computing* (рис. 3.5). Відповідно до запропонованого методу, його застосування забезпечується в рамках задач глибоких нейронних мереж для згорткових шарів *CNN*, оскільки саме їх використання в архітектурі мережі призводять до найбільшого збільшення обчислювальної складності. Для роботи із глибокими нейронними мережами використовується фреймворк *TensorFlow* та високорівневий інтерфейс роботи зі структурою нейронної мережі *Keras*. Для подальшого покращення продуктивності, може бути додатково використаний фреймворк *NVIDIA TensorRT*. В цілому запропонований метод адаптації може бути використаний для широкого спектру задач глибоких нейронних мереж таких, як класифікація, сегментація, детекція рухів людини, а також для медичних застосунків комп’ютерної діагностики захворювань людини. Більшої

продуктивності після адаптації було досягнуто для архітектур нейронних мереж *VGG*-16, *ResNet50*, *PSPNet*, *ICNet* та ін.

У сфері периферійних обчислень – *Edge Computing* даний метод дозволяє адаптувати зазначені архітектури глибоких нейронних мереж для їхнього використання на портативних пристроях зі спеціалізованою архітектурою *TPU*, *VPU*, *ASIC*, застосунках в рамках Інтернету речей – *IoT*, та медичній сфері. Дана адаптація дає можливість застосовувати моделі глибоких нейронних мереж на спеціалізованих обчислювальних пристроях, наприклад, *Coral Edge TPU USB Stick*, *Intel Movidius Myriad 2* та ін. Дані пристрої мають невелику вартість, у порівнянні з високопродуктивними графічними процесорами *GPU*. Вони потребують незначних змін у системі для роботи, оскільки працюють як допоміжні процесори, і можуть бути підключеними до основної системи через універсальну послідовну шину *USB3*, що забезпечує пропускну здатність даних на рівні 5 *Gbps*. Враховуючи, що *Coral Edge TPU USB Stick* витрачає всього 2 Вт енергії, це дає змогу використовувати даний прискорювач у задачах *Edge Computing* з високими обмеженнями на обчислювальні ресурси та енергоефективність.

Відповідно до запропонованого методу (рис. 3.5) для адаптації моделей глибоких нейронних мереж можна використовувати комплексний підхід, що поєднує методи прунінгу, квантизації, видалення зайвих класів об’єктів, та підвищення ефективності апаратного та програмного забезпечення системи до якої підключений прискорювач *TPU*.

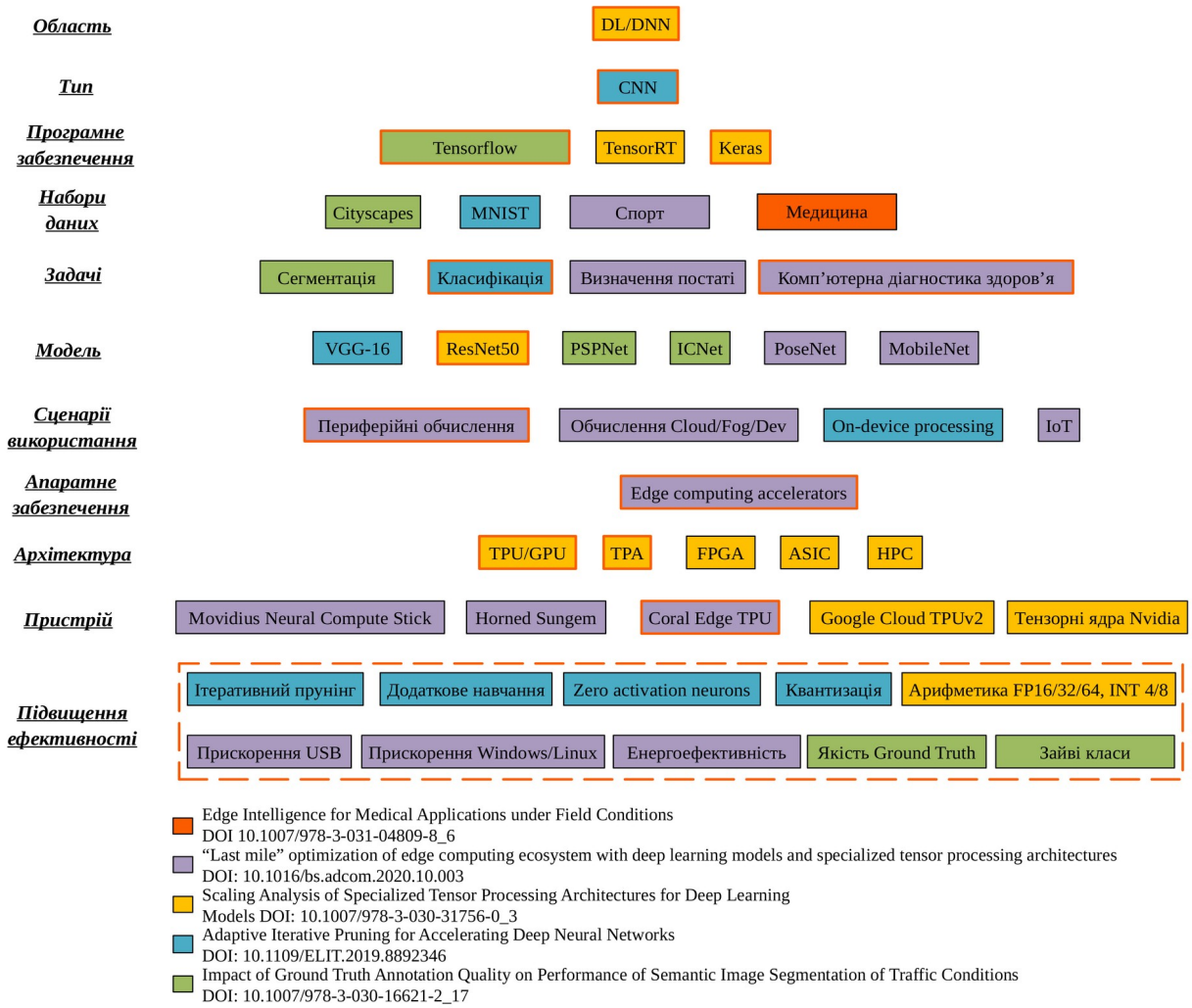

Рисунок 3.5 — Рівні застосування комплексного методу адаптації глибоких нейронних мереж

3.5 Адаптація моделі глибокої нейронної мережі під спеціалізовану обчислювальну архітектуру

Процес адаптації моделі під спеціалізований прискорювач полягає у використанні кроків із запропонованого методу і має наступний символьний вигляд.

$$\text{nn_adp} = f\left(D, quant\left(aprun\left(M_{train}\right), dtype\right)\right) \tag{3.9}$$

$$adp = \text{nn_adp}\left(M\right) \cdot \text{is_opt}\left(HW, SW\right), \tag{3.10}$$

де M_{train} є тренована модель під певну задачу; *aprun* — застосування розробленого методу адаптивного прунінгу для зменшення розміру глибокої нейронної мережі; *quant* — зміна розрядності представлення даних нейронної

мережі у формат *dtype* (наприклад, *uint8*) для її застосування на спеціалізованому прискорювачі *D*. В цілому, процедура адаптації глибоких нейронних мереж під спеціалізовану обчислювальну архітектуру включає в себе: адаптацію самої нейронної мережі (*nn_adp)* та підвищення ефективності цільової обчислювальної інфраструктури (*is_opt)*.

Основні кроки під час виконання процесу адаптації моделі наведено на рис. 3.6. В результаті, початкова повнорозмірна модель зменшується за допомогою адаптивного ітеративного прунінгу, виконується її квантизація та компіляція під конкретний прискорювач. Слід зауважити, що зазначені кроки при адаптації моделі можуть бути пропущені, в залежності від кінцевої мети адаптації та обраного прискорювача. В даному випадку, в якості моделі, було обрано модифіковану мережу *ResNet50* для виконання діагностики легеневих аномалій на основі рентген знімків. Відповідно до цієї задачі, мережу було змінено наступним чином: зменшено кількість вхідних каналів даних до 1, оскільки вхідні дані представлені як чорно-білі зображення, а також зменшено кількість класів моделі до 2, що забезпечує розпізнавання аномалій. Додатково, зменшення кількості класів дає можливість спростити архітектуру мережі. Фрагмент коду, що демонструє модифікацію моделі *ResNet50* наведено на рис. 3.7.

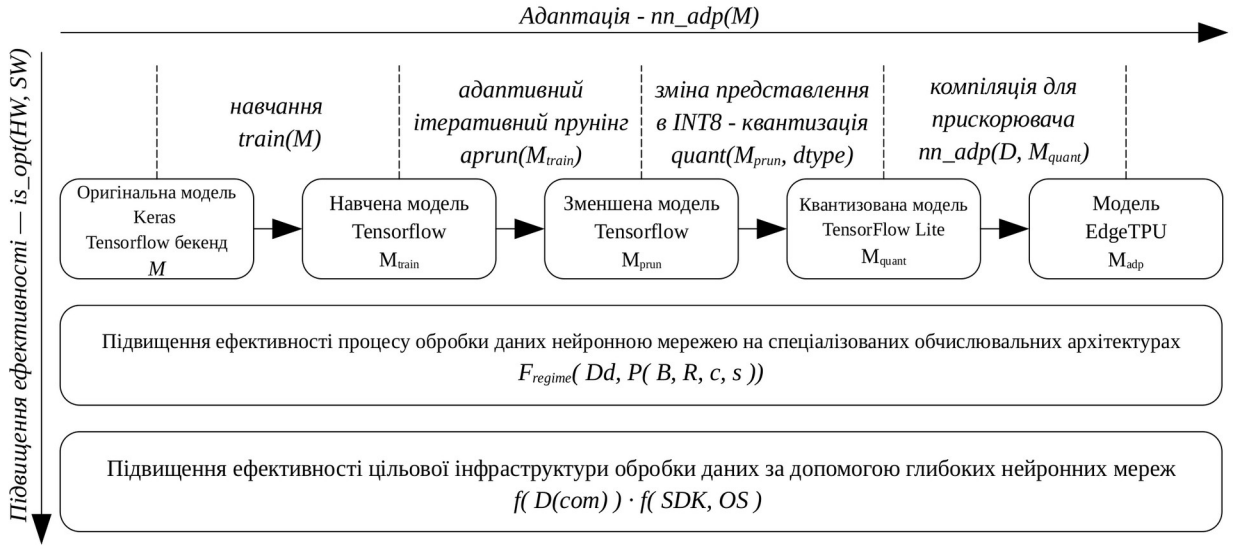

Рисунок 3.6 — Процес адаптації моделі нейронної мережу під прискорювач *TPU*

```
# create ResNet 50 with custom input shape for chest x-ray
resnet50 = ResNet50(weights='imagenet', include_top=False,
                    input_tensor=Input(shape=(image_size, image_size, 1)))
# chest X-ray classifier
classifier = resnet50.output
classifier = GlobalAveragePooling2D()(classifier)
classifier = MaxPooling2D(pool_size=(1,1))(classifier)
classifier = Flatten()(classifier)
classifier = Dropout(0.5)(classifier)
classifier = Dense(128, activation='relu')(classifier)
classifier = Dense(num_classes, activation='softmax')(classifier)
# freeze ResNet
resnet50.trainable = False
# stack to original ResNet 50
chest_resnet = Model(inputs=resnet50.input, outputs=classifier)
```

Рисунок 3.7 — Фрагмент коду модифікації моделі *ResNet50*

Основні шари моделі *ResNet50* ініціалізуються вагами *ImageNet*, що дозволяє отримати базову модель здатну виокремлювати загальні ознаки із вхідних даних. Подальший тюнінг моделі під задачу детекції легеневих аномалій виконується шляхом додаткового навчання модифікованого класифікатору (рис. 3.7) на наборі даних *CheXpert*, при цьому базова частина моделі *ResNet50* заморожується і не приймає участь у додатковому навчанні. Далі виконується зменшення розміру моделі шляхом застосування розробленого методу адаптивного ітеративного прунінгу (рис. 3.8).

В разі, якщо обрана модель є достатньо малою або потреба у зменшенні моделі відсутня, крок прунінгу можна пропустити і обмежитись лише навчанням моделі (рис. 3.9).

```
for i in range(0, pruning_iterations):
    for pln in prune_layer_name:
        # model pruning
        layer = model.get_layer(name=pln)
        apoz = identify.get_apoz(model, layer, x_val)
        high_apoz_channels = identify.high_apoz(apoz, method="std",
                                                cutoff_std=0.1)
        model = delete_channels(model, layer, high_apoz_channels)
        model.compile(optimizer='adam',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        # get total params after pruning
        params_after_prining=np.sum([K.count_params(p) for p in
                                     set(model.trainable_weights)])
        prune_level.append(1-(params_after_prining/original_model_params))
        # get pruning level
        print("Pruning level: "+str(1-(params_after_prining
                                        /original_model_params)))
        # test pruned model "before training"
        test_loss, test_accuracy = model.evaluate(x_val,
                                    y_val,
                                    batch_size=batch_size,
                                    verbose=2)
        # re-traing pruned model
        results = model.fit(chest_dataset,
                            epochs=num_epochs,
                            batch_size=batch_size,
                            verbose=training_verbosity,
                            validation_data=(x_val, y_val),
                            callbacks=[early_stopping, reduce_lr])
        # test pruned model "after re-training"
        test_loss, test_accuracy = model.evaluate(x_val,
                                    y_val,
                                    batch_size=batch_size,
                                    verbose=2)
```

Рисунок 3.8 — Фрагмент коду виконання адаптивного ітеративного прунінгу моделі

```
# train model
checkpoint = ModelCheckpoint("resnet50_imagenet_grayscale_best",
                             monitor='loss', verbose=1, save_best_only=True,
                             mode='auto', save_freq=100)
opt = tf.keras.optimizers.Adam(learning_rate=0.01)
chest_resnet.compile(optimizer=opt, loss='categorical_crossentropy',
                     metrics=["accuracy"])
chest_resnet.fit(chest_dataset, batch_size=batch_size, epochs=num_epochs,
                 callbacks=[checkpoint])
```

Рисунок 3.9 — Фрагмент коду навчання моделі

За результатами виконання кроків навчання та прунінгу отримуємо зменшену навчену модель готову до адаптації під спеціалізований прискорювач нейронних мереж. В якості спеціалізованого прискорювача було обрано *Google Coral Edge TPU*. Даний прискорювач працює із моделями у форматі *TFLite*. Оскільки, після описаних вище кроків було отримано модель *TensorFlow*, необхідно адаптувати модель для роботи на прискорювачі (рис. 3.10). Для цього виконується конвертація моделі *TensorFlow* у *TFLite*, а також додатково виконується зміна представлення значень у моделі з *FP32* у *INT8*. Дана зміна представлення – квантизація необхідна умова для застосування моделі на даному прискорювачі так, як пристрій підтримує лише операції цілочисельної арифметики.

```
# convert to quantized tflite int8 + edge TPU optimization
converter = tf.lite.TFLiteConverter.from_keras_model(chest_resnet)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8
tflite_model = converter.convert()
```

Рисунок 3.10 — Фрагмент коду, що виконує адаптацію моделі під спеціалізований прискорювач

Після виконання адаптації модель сумісна із обраним прискорювачем і готова для компіляції. Цей процес формує граф моделі для *Edge TPU*, що є сумісним із конкретним середовищем виконання. В даній роботі було використано середовище виконання 13-ї версії, що підтримує виконання операцій шарів в обраній моделі. Для компіляції використовуються засоби *SDK* для *Coral Edge TPU* (рис. 3.11).

```
! edgetpu_compiler --min_runtime_version 13 "$TFLITE_FILE"
```

Рисунок 3.11 — Компілятор моделі для *Coral Edge TPU*

За результатами роботи компілятора отримуємо модель мережі готову для застосування на спеціалізованому прискорювачі *Coral Edge TPU*. Із наведеного нижче виводу про статус компіляції маємо інформацію про отриманий граф моделі (рис. 3.12).

```
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 4873 ms.

Input model: /content/gdrive/MyDrive/Chest ResNet50
EdgeTPU/chest_resnet50_grayscale_uint8.tflite
Input size: 23.12MiB
Output model: chest_resnet50_grayscale_uint8_edgetpu.tflite
Output size: 22.86MiB
On-chip memory used for caching model parameters: 7.59MiB
On-chip memory remaining for caching model parameters: 0.00B
Off-chip memory used for streaming uncached model parameters: 15.21MiB
Number of Edge TPU subgraphs: 1
Total number of operations: 77
Compilation succeeded!

Operator                        Count      Status

MEAN                            1          Mapped to Edge TPU
PAD                             2          Mapped to Edge TPU
MAX_POOL_2D                     1          Mapped to Edge TPU
ADD                             16         Mapped to Edge TPU
FULLY_CONNECTED                 1          Mapped to Edge TPU
CONV_2D                         53         Mapped to Edge TPU
QUANTIZE                        2          Mapped to Edge TPU
SOFTMAX                         1          Mapped to Edge TPU
```

Рисунок 3.12 — Статус компіляції моделі для спеціалізованого прискорювача

Важливою інформацією є кількість під-графів у моделі *Edge TPU*. Згідно наведеній вище інформації було побудовано один граф без під-графів. Це свідчить про те, що всі операції у моделі мережі підтримуються прискорювачем і можуть виконуватись на ньому. Продуктивність роботи при цьому є максимальною. В разі якщо певні операції не підтримуються прискорювачем, їх виконання перенаправляється на *CPU*, що веде за собою зменшення продуктивності обробки даних моделлю нейронної мережі.

В кінці процесу адаптація моделі глибокої нейронної мережі під спеціалізовану обчислювальну архітектуру – прискорювач *Coral Edge TPU* отримаємо готову модель *TFLite Edge TPU*. Фрагмент коду, що демонструє

завантаження адаптованої моделі у прискорювач *Coral Edge TPU* показано на рис. 3.13.

```
def make_tpu_interpreter(flash_drive_model):
    model_file, *device = flash_drive_model.split('@')
    try:
        interpreter = tflite.Interpreter(
            model_path=model_file,
            experimental_delegates=[
                tflite.load_delegate(EDGETPU_SHARED_LIB,
                                     {'device': device[0]} if device
                                     else {})
            ])

        return interpreter, 0

    except ValueError:
        return None, None
```

Рисунок 3.13 — Фрагмент коду завантаження адаптованої моделі у спеціалізований прискорювач

3.6 Програма діагностики легеневих аномалій

Для дослідження ефективності роботи прискорювача *Coral Edge TPU USB Stick* в практичних умовах, було розроблено тестову портативну програму для діагностики легеневих аномалій за даними рентген знімків. В якості архітектури глибокої нейронної мережі для даної задачі було обрано *ResNet50*, що було треновано на наборі даних *ChestXray* та адаптовано під спеціалізований прискорювач, відповідно до запропонованого методу адаптації. В якості класів, з якими працює модель, було обрано два класи: відсутність аномалій у легенях – *normal*, та наявність аномалій – *abnormal*. Основна мета розробленої програми була продемонструвати ефективну продуктивність прискорювача *TPU* у порівнянні із системою в загальній комплектації без додаткових графічних прискорювачів. До подібних систем можна віднести звичайний персональний комп’ютер, що наявний у кабінеті лікаря. Оскільки, подібна система може не відповідати сучасним вимогам для

роботи із моделями глибоких нейронних мереж, важливо, щоб портативне програмне забезпечення для нейронних мереж було максимально сумісним із більшістю систем та не потребувало великої кількості ресурсів для встановлення та налаштування. Враховуючи це, додатковою вимогою для розробленої тестової програми було забезпечити мінімальні витрати ресурсів системи на якій вона виконується. Для досягнення цієї вимоги було обрано *TF Lite* – портативне середовище виконання для нейронних мереж. Для можливості використовувати прискорювач *Coral Edge TPU USB Stick* на цільовій системі, додатково встановлюється середовище виконання для *EdgeTPU*. Загалом, при розгортанні тестової програми на цільовій системі під керуванням ОС *Windows*, було досягнуто мінімальні витрати обсягів пам’яті у розмірі 27,37 MB. До них відносяться:

- Інтерпретатор *Python* програм – 26 MB;
- Середовище виконання *TF Lite* – 0,63 MB;
- Середовище виконання для *EdgeTPU* – 0,74 MB.

При цьому, обсяг бібліотек для роботи з глибокими нейронними мережами становить всього 5% від загального обсягу додаткового ПЗ. Після встановлення зазначеного ПЗ тестова програма може виконуватись на цільовій системі.

Тестова програма представляється як портативна альтернатива повнофункціональної платформи для діагностики захворювань легень. Вона дозволяє робити базову діагностику наявності хвороби легень у «польових» умовах. Для роботи лікарю достатньо мати флеш-накопичувач із підготовленими моделями нейронних мереж, отриманих із повнофункціональної платформи, та портативний прискорювач *Coral Edge TPU USB Stick*.

Після запуску, програми очікує флеш-накопичувач із моделлю нейронної мережі (рис. 3.14), при цьому функціонал програми може залежати від наданої нейронної мережі. Відповідно до потреб, лікар може завантажити

відповідну модель нейронної мережі для проведення потрібної діагностики захворювання.

Рисунок 3.14 — Портативна програма для діагностики легеневих аномалій

При підключені флеш-накопичувача із нейронною мережею програма автоматично завантажує наявну на ньому модель і, відповідно цьому, стають доступні діагностичні функції. В даному прикладі використовується глибока нейронна мережа *ResNet50* для діагностики легеневих аномалій за рентген знімками по двом класами: «відсутність» аномалії та «присутність» аномалії. Після запуску процесу обробки, програма проводить діагностику рентген знімків, отриманих від лікаря (рис 3.15).

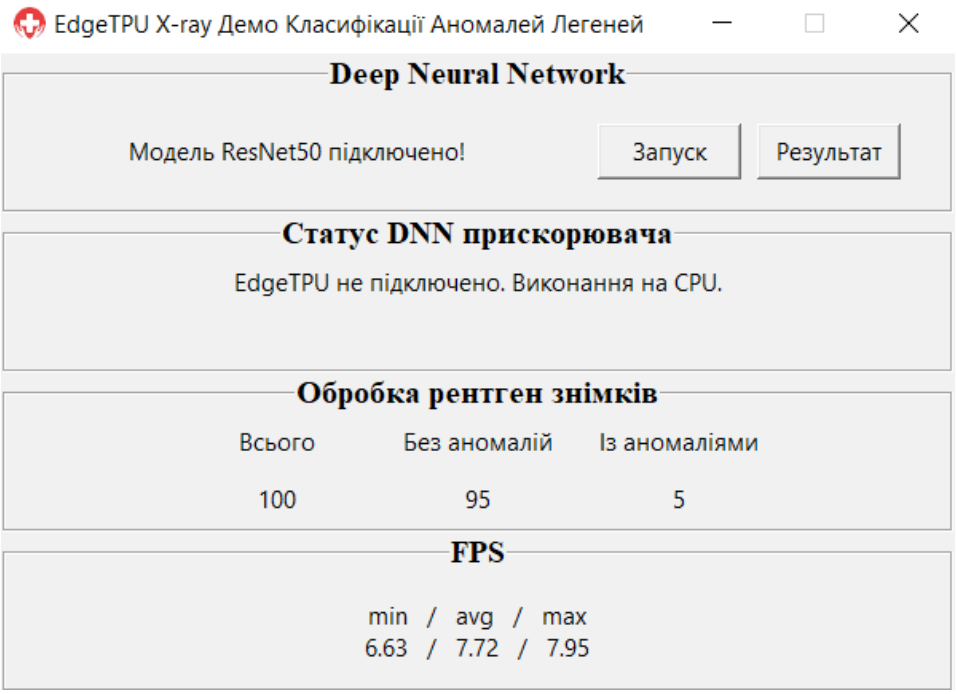

Рисунок 3.15 — Завантаження моделі *ResNet50* при підключенні флеш-накопичувача та проведення діагностики легень

Переглянути результати діагностики певного рентген знімку лікар може натиснувши кнопку «Результат», після чого відкривається бокове вікно із зображенням рентгену та результатами діагностики (рис. 3.16).

Слід зауважити, що в разі якщо лікар не підключив прискорювач *Edge TPU*, то програма працюватиме у резервному режимі на *CPU* системи. Після підключення прискорювача *TPU*, програма автоматично переведе виконання нейронної мережі на нього (рис. 3.17, 3.18).

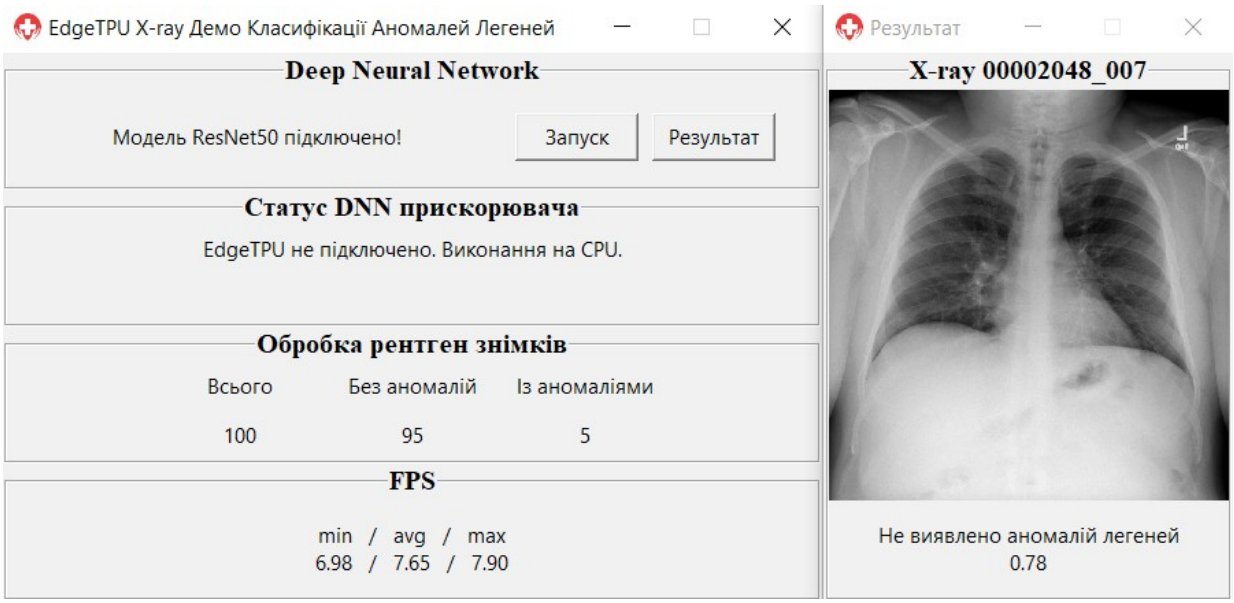

Рисунок 3.16 — Зображення рентген знімку та результатами діагностики у боковому вікні

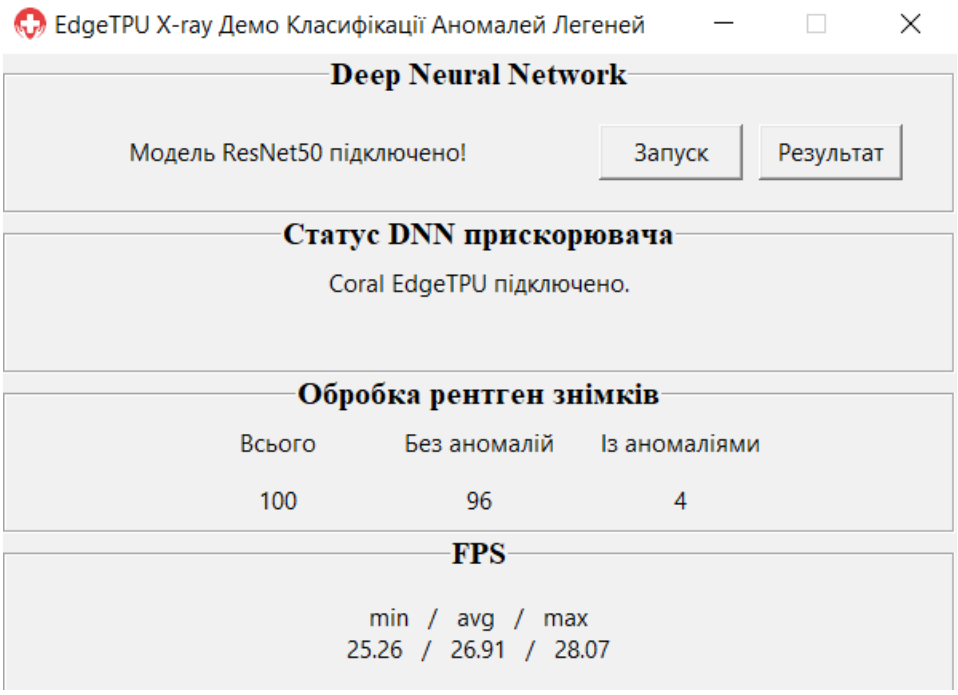

Рисунок 3.17 — Виконання нейронної мережі *ResNet50* на *Coral EdgeTPU USB Stick*

Важливим аспектом роботи програми є те, що дані рентген знімків нікуди не передаються і оброблюються локально на комп’ютері лікаря. Це дозволяє забезпечити захист персональних даних пацієнта. Також,

використання даної портативної програми для діагностики, дозволяє проводити аналіз рентген знімків в умовах нестійкого Інтернет зв’язку або його повної відсутності, коли доступ до повнофункціональної платформи для діагностики захворювань не можливий.

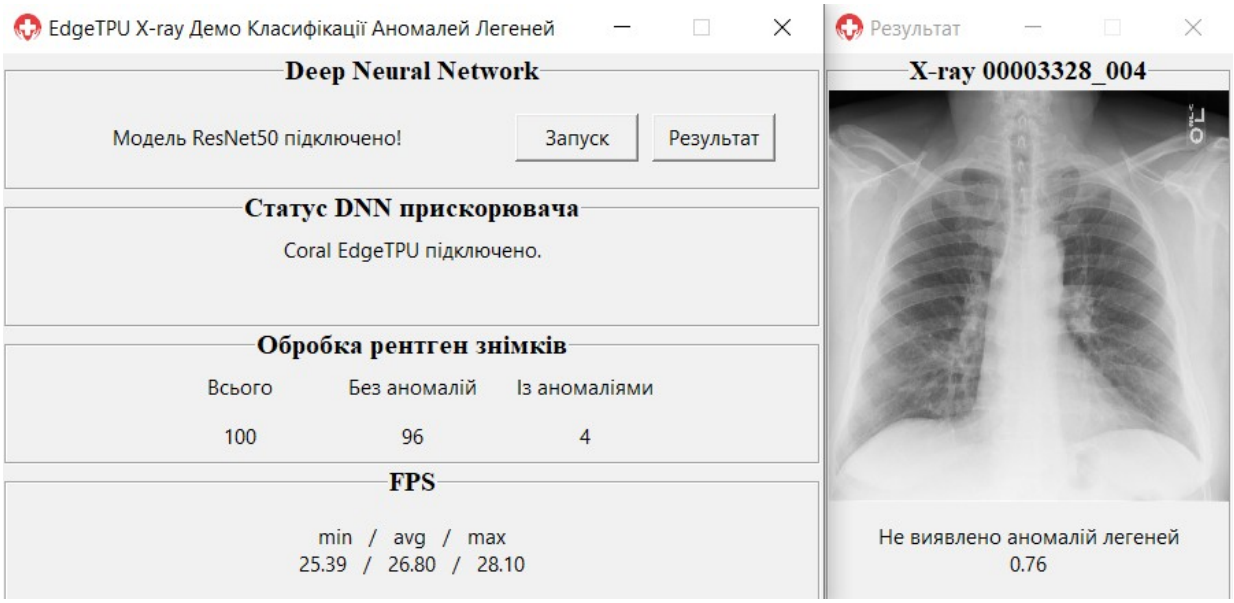

Рисунок 3.18 — Результат діагностики при підключеному прискорювачі *EdgeTPU*

Основний код програми діагностики легеневих аномалій наведено у додатку А.

3.7 Висновки до розділу 3

В даному розділі було розроблено комплексний метод адаптації глибоких нейронних мереж до апаратного забезпечення зі спеціалізованою архітектурою. Відповідно було:

1. Розроблено метод адаптивного ітеративного прунінгу для зменшення розміру моделей нейронних мереж, що базується у поступовому зменшенні розміру мережі, шляхом видалення зайвих каналів у шарах та додаткового навчання отриманої зменшеної моделі для відновлення

точності роботи. При цьому, гіперпараметри мережі адаптивно змінюються, щоб компенсувати втрати точності після кожної ітерації прунінгу та досягти обраного критерію такого, як прискорення;

2. Розроблено метод підвищення ефективності процесу обробки даних нейронними мережами на спеціалізованих обчислювальних архітектурах, що враховує технічні особливості обробки даних за допомогою глибоких нейронних мереж на спеціалізованих прискорювачах, наприклад, ітерація виконання обчислень, та дозволяє визначити параметри такі, як розмір порції даних, щоб збільшити продуктивність обробки даних за рахунок зменшення впливу накладних витрат ініціалізації та передачі даних;

3. Розроблено метод підвищення ефективності обчислювальної інфраструктури для глибоких нейронних мереж, що дозволяє налаштувати програмну та апаратну складову таку, як ОС та інтерфейси підключення, що дозволяє збільшити продуктивність та ефективність обробки даних нейронними мережами на цільовій системі.

Також показано процес адаптації глибоких нейронних мереж для спеціалізованого обчислювального прискорювача *Coral Edge TPU* із використанням розробленого комплексного методу.

Для дослідження ефективності роботи спеціалізованого прискорювача *Coral Edge TPU USB* в практичних умовах, було розроблено тестову портативну програму для діагностики легеневих аномалій за даними рентген знімків. В якості архітектури глибокої нейронної мережі для даної задачі було обрано *ResNet50*, що було треновано на наборі даних *ChestXray* та адаптовано під спеціалізований прискорювач, відповідно розробленого методу.

РОЗДІЛ 4
АНАЛІЗ РЕЗУЛЬТАТІВ ЗАСТОСУВАННЯ МЕТОДУ АДАПТАЦІЇ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ

Запропонований комплексний метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур призначений для перетворення моделей глибоких нейронних мереж у моделі сумісні із сучасними прискорювачами нейронних мереж, наприклад *Google Coral Edge TPU*. Метод включає в себе комплексний підхід в адаптації моделей нейронних мереж та інфраструктури для нейронних мереж, за рахунок використання наступних етапів:

1. Зменшення розміру моделі мережі, за рахунок використання розробленого методу адаптивного ітеративного прунінгу;
2. Адаптація моделі нейронної мережі для її сумісності із спеціалізованим прискорювачем, шляхом перетворення представлення з *FP32* у *INT8* методом квантизації.
3. Підвищення ефективності процесу обробки даних нейронною мережею на спеціалізованому прискорювачі, що базується на проведеному аналізі технічних особливостей роботи спеціалізованих обчислювальних архітектур;
4. Підвищення ефективності цільової інфраструктури для нейронних мереж на основі отриманих результатів аналізу впливу апаратного та програмного забезпечення на ефективність обробки даних за допомогою моделей глибоких нейронних мереж.

4.1 Аналіз результатів застосування методу адаптивного ітеративного прунінгу

Для отримання чисельних показників прискорення та втрат точності, при застосуванні розробленого методу адаптивного ітеративного прунінгу,

було проведено зменшення глибокої нейронної мережі *VGG*-16, модифікованої для задачі класифікації. В ході навчання використовувався набір даних *MNIST*. Після кожної ітерації прунінгу проводились заміри точності роботи зменшеної моделі «до» і «після» додаткового навчання. Протягом перших ітерації прунінгу мережі, втрати точності були в межах 1-2%, але із подальшим зменшенням розміру моделі втрати точності становили до 80%. Щоб компенсувати ці втрати, було виконано поступове збільшення епох додаткового навчання моделі мережі після прунінгу. В ході застосування даного методу прунінгу було виявлено, що досягнути обраного критерію покращення – найменшого часу обробки даних моделлю мережі без значних втрат точності не можливо, використовуючи постійний та лінійний законі зміни гіперпараметрів. Але використання мультиплікативного та адаптивного законів зміни гіперпараметрів, дозволяє досягти зазначеного критерію покращення. Нижче наводяться результати для прунінгу із застосуванням мультиплікативного закону:

$$P_i = b\,P_{i-1}. \tag{4.1}$$

за яким змінюються наступні гіперпараметри під час додаткового навчання:

- Розмір порції даних – B_i, що змінюється відповідно формулі:

$$B_i = b_B B_{i-1}, \tag{4.2}$$

де $b_B = 2$, та B_0 – початковий розмір порції даних дорівнює 256,

- Кількість епох навчання – R_i, що змінюється відповідно формулі:

$$R_i = b_R R_{i-1}, \tag{4.3}$$

де $b_R = 2$, та R_0 – початкова кількість епох додаткового навчання дорівнює 1.

Початкова модель мережі *VGG*-16 до застосування адаптивного ітеративного прунінгу містила кількість параметрів $N_0 > 14$млн. Після 9-ї ітерації прунінгу кількість параметрів стала $N_9 = 4826$, що відповідає її зменшенню у 3050 разів і дозволило досягти прискорення $S = 32{,}2$, порівнюючи із початковою моделлю мережі. При цьому, фінальна точність

становила $A_9 = 0{,}962$ і була навіть більшою за точність початкової моделі $A_0 = 0{,}957$. Таблиця 4.1 містить абсолютні значення для різних ітерацій прунінгу та епох додаткового навчання, де:

- i – номер ітерації прунінгу;
- Z_i – порція видалених параметрів після i ітерації прунінгу;
- S_i – прискорення моделі після i ітерації прунінгу;
- A_i – точність додатково навченої моделі після i ітерації прунінгу.

Таблиця 4.1 — Показники моделі після кожної ітерації прунінгу

i	Z_i	S_i	A_i
INITIAL	–	1	**0.957**
0	0.88946	2.65	0.958
1	0.97100	5.22	0.965
2	0.98217	7.79	0.974
3	0.99820	15.2	0.974
4	0.99902	19.9	0.969
5	0.99942	24.8	0.967
6	0.99957	30.6	0.963
7	0.99965	31.7	0.960
8	0.9997	30.7	0.957
9	0.99973	32.2	0.962

Збільшення прискорення S_i відбувається набагато повільніше ніж зменшення розміру мережі $C_i = 1/(1-Z)_i$, і потребує досить сильного зменшення, щоб досягти високого прискорення (рис. 4.1). Із графіку видно, що збільшення прискорення має наступну тенденцію $S_i \sim O(C_i^{0,5})$. Ця тенденція справедлива для менших значень рівня прунінгу d_{cut}, що відповідає більш агресивному прунінгу і дозволяє досягти високих показників прискорення S_i. На противагу, більші значення d_{cut}, що відповідає слабшому прунінгу, ведуть до досягнення меншого прискорення в межах $S_i < 10$.

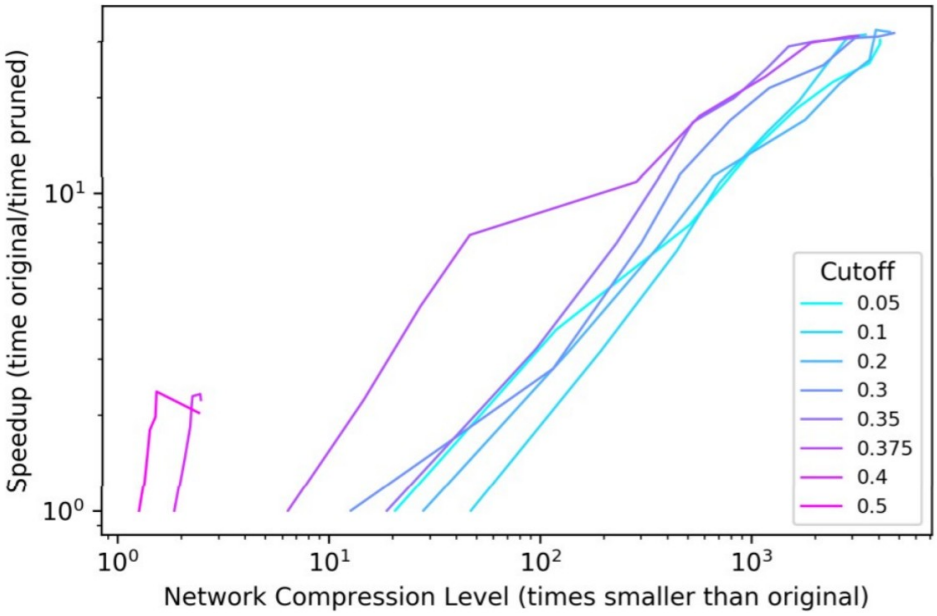

Рисунок 4.1 — Рівень зменшення мережі у порівнянні із збільшенням прискорення

Тенденція відновлення точності роботи моделі нейронної мережі, в процесі додаткового навчання після прунінгу, демонструється на рис. 4.2, де показано величину втрати точності LA_i у порівнянні із отриманим прискоренням мережі для різних значень рівня прунінгу d_{cut}.

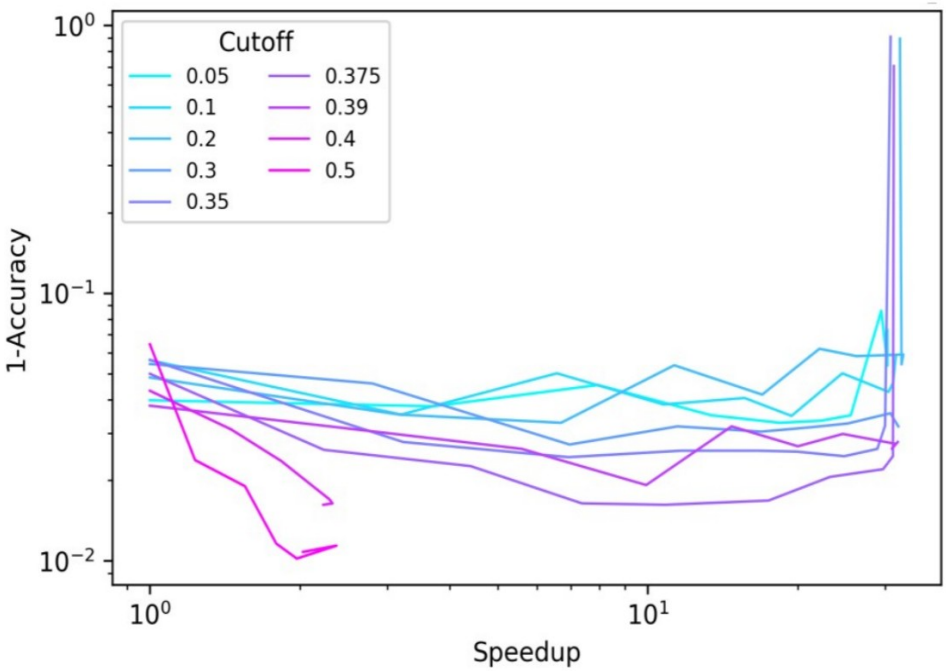

Рисунок 4.2 — Величина втрати точності відносно прискорення мережі

Використання запропонованого адаптивного підходу при зміні гіперпараметрів дозволило уникнути негативних наслідків втрати точності від збільшення прискорення мережі за рахунок прунінгу, і дозволило досягти прискорення $S_i \sim 30$. В результаті, було отримано значно меншу версію початкової моделі нейронної мережі при мінімальних показниках втрати точності. Особливу увагу слід приділити підбору значення рівня прунінгу d_{cut} на кожній ітерації, оскільки завеликі значення d_{cut} $0{,}4-0{,}5$ не дають можливість досягти високих показників прискорення, що демонструється у лівій нижній частині на рис. 4.1. Відмічено певний проміжок значень, які забезпечують мінімальні втрати точності при досягнені суттєвого прискорення, наприклад, при значенні $d_{cut}=0{,}375$ досягається прискорення $S_i=17$ при втраті точності 2% або для $d_{cut}=0{,}2$ – досягається $S_i>30$ при втраті точності 5% (рис. 4.2).

Додатково було проведено аналіз тенденції відновлення точності на моделях мережі отриманих після звичайного прунінгу. Для цього було взято 10 моделей ідентичної архітектури, але з відмінною кількістю параметрів, що залишились після кожної ітерації прунінгу. Параметри кожної із 10 моделей було повторно ініціалізовано випадковим чином і проведено їх навчання із загальною кількістю ітерацій – епох, що відповідає сумі кількості епох на кожній ітерації прунінгу, відповідно формулі 4.3, і дорівнює 1023. Таким чином, процес має наступний вигляд:

- Виконання прунінгу моделі мережі *i*-раз;
- Повторна ініціалізація моделі мережі;
- Навчання моделі загальною кількість епох 1023 з кроком, що відповідає кількості епох для кожної ітерації прунінгу (формула 4.3).

Метою було порівняти тенденцію відновлення точності для звичайного процесу прунінг із навчанням і ітеративного прунінгу із додатковим навчанням на кожній ітерації. На рис. 4.3 показано тенденцію відновлення точності 10 моделей після прунінгу в процесі їх навчання (кольорові лінії) у порівнянні моделлю, отриманою у процесі адаптивного ітеративного

прунінгу (чорна лінія). Графік втрати точності для аналогічних моделей показано на рис 4.4.

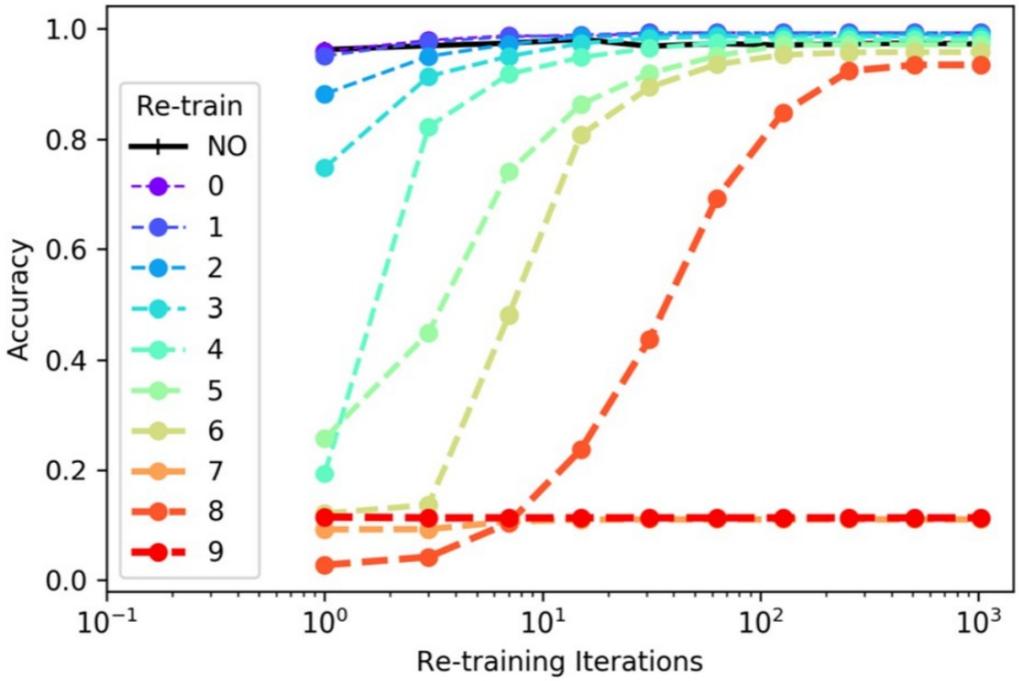

Рисунок 4.3 — Тенденцію відновлення точності для 10 моделей мереж після прунінгу (кольорові) та моделі після адаптивного ітеративного прунінгу (чорна)

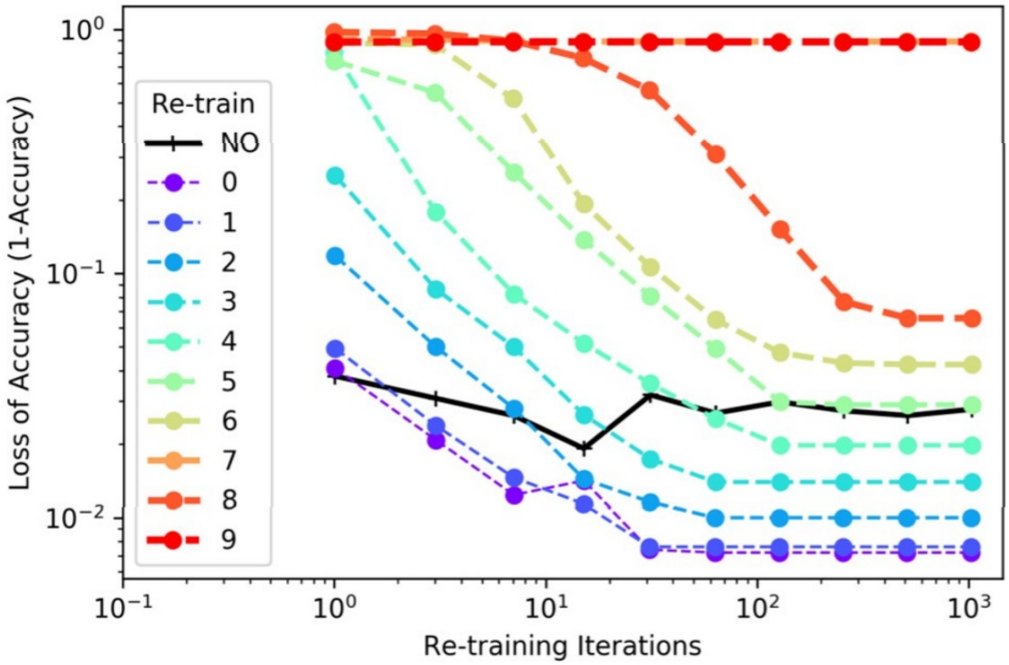

Рисунок 4.4 — Тенденцію втрати точності для 10 моделей мереж після прунінгу (кольорові) та моделі після адаптивного ітеративного прунінгу (чорна)

У таблиці 4.2 наведені чисельні значення точності A_i^k для моделей, отриманих після k-ітерацій звичайного прунінгу та i-ітерації навчання, а також прискорення моделі S^k.

Таблиця 4.2 — Значення точності для моделей після k-ітерацій звичайного прунінгу

i	$A^0{}_i$	$A^1{}_i$	$A^2{}_i$	$A^3{}_i$	$A^4{}_i$	$A^5{}_i$	$A^6{}_i$	$A^7{}_i$	$A^8{}_i$	$A^9{}_i$
INITIAL	–	–	–	–	–	–	–	–	–	–
0	0.959	0.9506	0.8818	0.748	0.194	0.2578	0.1206	0.0916	0.0272	0.1142
1	0.9792	0.9762	0.95	0.9138	0.8222	0.4494	0.1362	0.092	0.0416	0.1126
2	0.9876	0.9854	0.972	0.95	0.9176	0.7406	0.4802	0.106	0.1034	0.1126
3	0.9858	0.9886	0.9856	0.9736	0.9482	0.863	0.8076	0.1094	0.2378	0.1126
4	0.9926	**0.9924**	0.9884	0.9826	0.9646	0.9194	0.894	0.1098	0.4376	0.1126
5	**0.9928**	0.9924	**0.99**	**0.986**	0.9746	0.9508	0.9354	0.1098	0.6922	0.1126
6	0.9928	0.9924	0.99	0.986	**0.9802**	0.9702	0.9526	0.1098	0.8482	0.1126
7	0.9928	0.9924	0.99	0.986	0.9802	**0.971**	0.957	0.1098	0.9234	0.1126
8	0.9928	0.9924	0.99	0.986	0.9802	0.971	**0.9576**	0.1098	**0.9344**	0.1126
9	0.9928	0.9924	0.99	0.986	0.9802	0.971	0.9576	0.1098	0.9344	0.1126
S^k	**2.2**	**4.4**	**6.6**	**13**	**17**	**21.2**	**28.2**	**28.7**	**28.9**	**29.2**

З графіків 4.3 та 4.4 слідує, що модель, зменшена за допомогою розробленого методу адаптивного ітеративного прунінгу після 10 кроків ітеративного прунінгу та додаткового навчанням, досягла більшої точності – 0,962 у порівнянні з моделлю після 8-ми ітерацій прунінгу з навчанням з нуля – 0,9344. Окремо слід відмітити, що після 10 кроків звичайного прунінгу, модель не відновила свою точність роботи через потрапляння функції оптимізації у локальний мінімум в ході навчання.

За результатами застосування методу адаптивного ітеративного прунінгу було проаналізовано ефективність розробленого методу у порівняні зі звичайним методом прунінгу. Розроблений метод базується на адаптивній зміні гіперпараметрів таких, як швидкість навчання, розмір порції даних, кількість епох, рівень прунінгу. Це дозволяє компенсувати втрати точності за рахунок додаткового додаткового навчання мережі з вагами, що залишилися після чергової ітерації прунінгу. В ході тестування на архітектурі *VGG*-16 для

задачі класифікації на наборі даних *MNIST* було продемонстровано наступні показники:

- Після адаптивного ітеративного прунінгу досягається прискорення у 15,2 раз при втраті точності менше 3% (4 ітерації прунінгу);
- Після адаптивного ітеративного прунінгу досягається прискорення у 30,6 раз при втраті точності менше 4% (7 ітерацій прунінгу);
- Після адаптивного ітеративного прунінгу було досягнуто максимального прискорення 32,2 із точність роботи 96,2% (10 ітерацій прунінгу);
- Після звичайного прунінгу і навчання мережі було досягнуто прискорення 13 і точність роботи 98,6% (3 ітерації прунінгу та 63 епохи навчання).

Отримані результати підтверджують ефективність розробленого методу адаптивного ітеративного прунінгу.

4.2 Аналіз технічних особливостей роботи спеціалізованих обчислювальних архітектур

Для порівняльного аналізу роботи глибоких нейронних мереж на спеціалізованих прискорювачах *TPU* та графічних прискорювачах *GPU* було проведено заміри часу виконання ітерації для моделей *VGG16* та *ResNet50* для задачі класифікації на наборі даних *MNIST*. В якості спеціалізованого прискорювача було використано *Google TPUv2*, доступ до якого надавався як хмарний ресурс із виділенням 8 ядер *TPU*. В якості графічного прискорювача використовувалась *NVIDIA Tesla K80*. На основі отриманих показників часу виконання ітерації на різних прискорювачах при різних розмірах порції даних було відмічено наявність додаткових затримок – накладних витрат при застосуванні моделі мережі на спеціалізованому прискорювачі (рис. 4.5). При цьому, затримки для графічного прискорювача також присутні, але менш виражені (рис 4.6). Подібні накладні витрати на 1-й та 2-й ітераціях можна

пояснити додатковими витратами часу на ініціалізацію моделі мережі на спеціалізованому прискорювачі та завантаження у пам’ять пристрою даних для обробки.

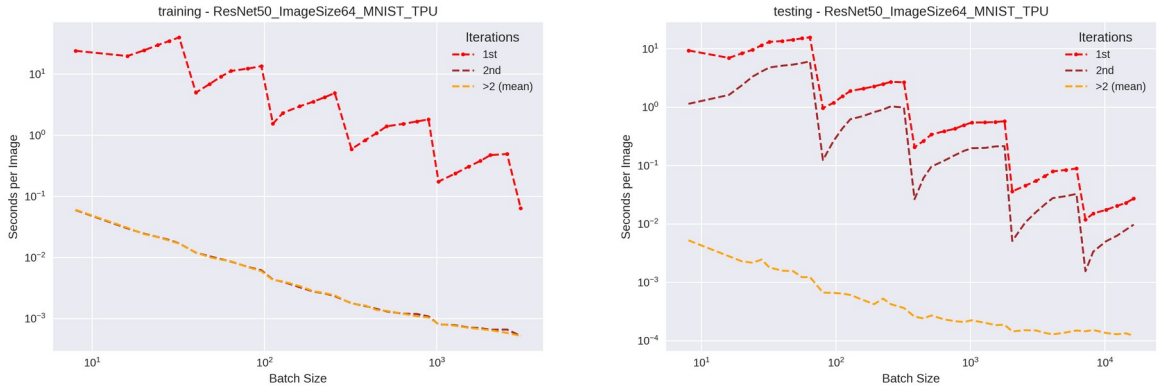

Рисунок 4.5 — Час виконання ітерацій при різних розмірах порції даних для спеціалізованого прискорювача *TPU* – етап навчання (лівий) та етап виконання (правий)

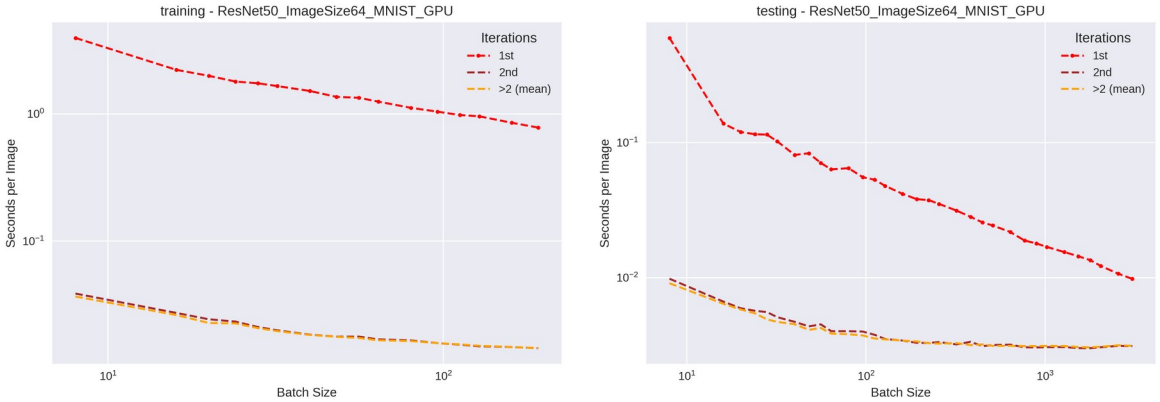

Рисунок 4.6 — Час виконання ітерацій при різних розмірах порції даних для графічного прискорювача *GPU* – етап навчання (лівий) та етап виконання (правий)

Подібні накладні витрати часу присутні на 1-й ітерації для обох прискорювачів, але для *TPU* суттєві затримки виявляються і на 2-й ітерації, і у порівнянні із *GPU* є набагато більшими. Для дослідження впливу додаткових затримок часу через присутні накладні витрати було побудовано графіки залежності часу виконання ітерації в залежності від розміру вхідних

даних при різних розмірах порції даних для різних ітерацій: 1-а, 2-а та ≥3. На рис. 4.7 приведено заміри часу виконання 1-ї та 2-ї ітерацій моделі *VGG16* для обох прискорювачів: *TPU* – T1, T2 та *GPU* – G1, G2 при різних розмірах порції даних від 8 до 160. Аналогічні дані приведені на рис. 4.8 для 3-ї ітерації, коли накладні витрати найменше виражені.

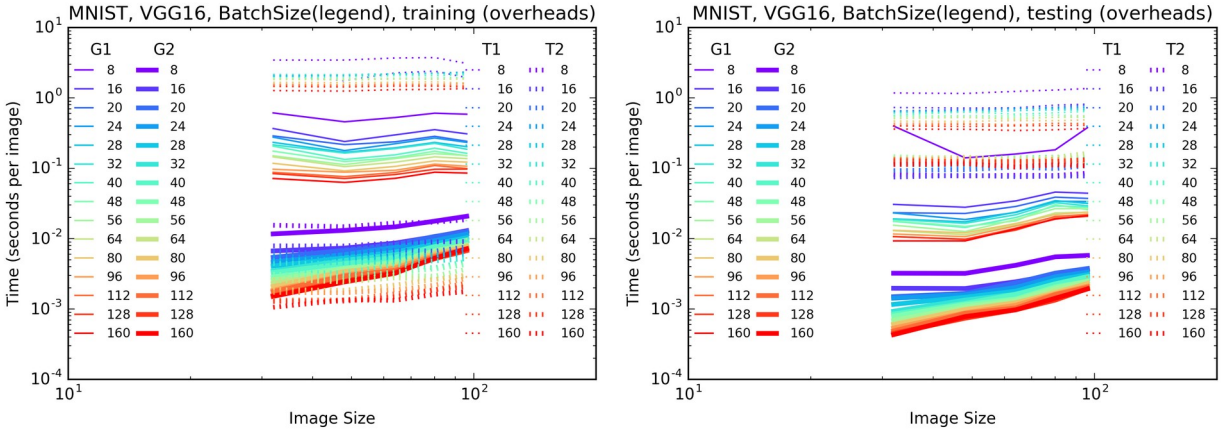

Рисунок 4.7 — Час виконання 1-ї та 2-ї ітерацій навчання (лівий) і виконання (правий) моделі *VGG16* в залежності від розміру вхідних даних для обох прискорювачів: *TPU* – T1, T2 та *GPU* – G1, G2 при різних розмірах порції даних від 8 до 160

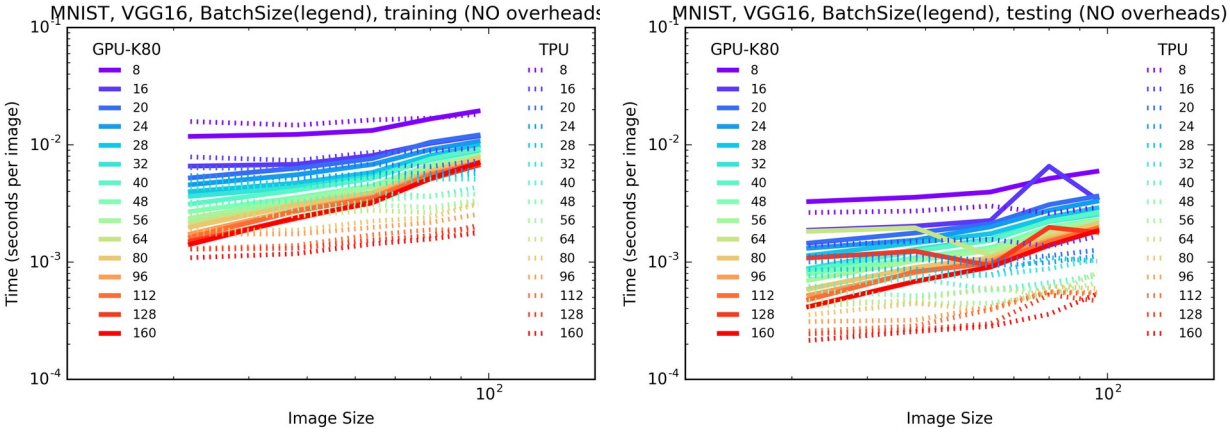

Рисунок 4.8 — Час виконання 3-ї ітерацій навчання (лівий) і виконання (правий) моделі *VGG16* в залежності від розміру вхідних даних для обох прискорювачів: *TPU* – T3 та *GPU* – G3 при різних розмірах порції даних від 8 до 160

З отриманих графіків слідує, що у *TPU* значно більші накладні витрати, що супроводжуються збільшеним часом ітерації T1, T2 у порівнянні із *GPU* – G1, G2. Через це на початкових ітераціях продуктивність *TPU* не перевищує GPU, а для малих розмірів порції даних навіть менша. В той же час, тенденція збільшення часу ітерації зі збільшенням розміру даних для *TPU* є менш вираженою у порівнянні із *GPU*. На рис. 4.9. та 4.10 приведені аналогічні дані для моделі *ResNet50*.

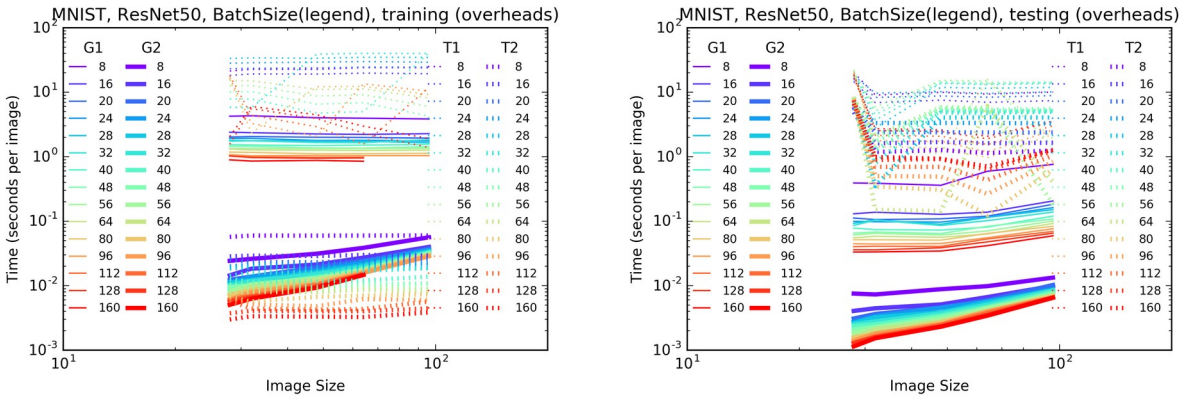

Рисунок 4.9 — Час виконання 1-ї та 2-ї ітерацій навчання (лівий) і виконання (правий) моделі *ResNet50* в залежності від розміру вхідних даних для обох прискорювачів: *TPU* – T1, T2 та *GPU* – G1, G2 при різних розмірах порції даних

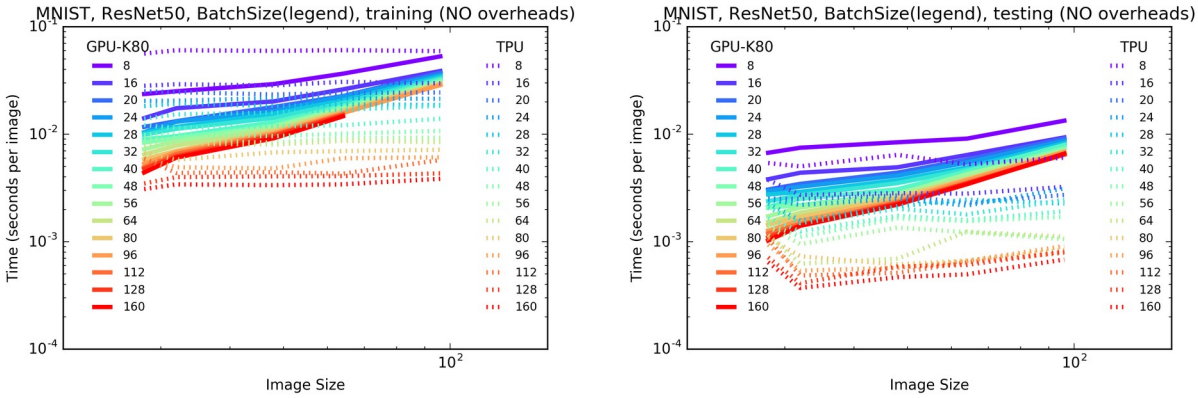

Рисунок 4.10 — Час виконання 3-ї ітерацій навчання (лівий) і виконання (правий) моделі *ResNet50* в залежності від розміру вхідних даних для обох прискорювачів: *TPU* – T3 та *GPU* – G3 при різних розмірах порції даних

Отримані результати, що показано на рис. 4.7-4.10, демонструють майже однакову продуктивність *TPU* у порівнянні із *GPU* при малих розмірах порції даний від 8 до 48. З цього слідую, що для більш ефективної роботи прискорювача *TPU*, краще обирати більші розміри порції даних ≥ 64.

Для порівняння залежності часу ітерації від розміру порції даних було проведено екстраполяцію отриманих результатів, на основі чого було побудовано графіки часу ітерації від розміру порції даних для обох прискорювачів в обох режимах роботи (рис. 4.11-4.13).

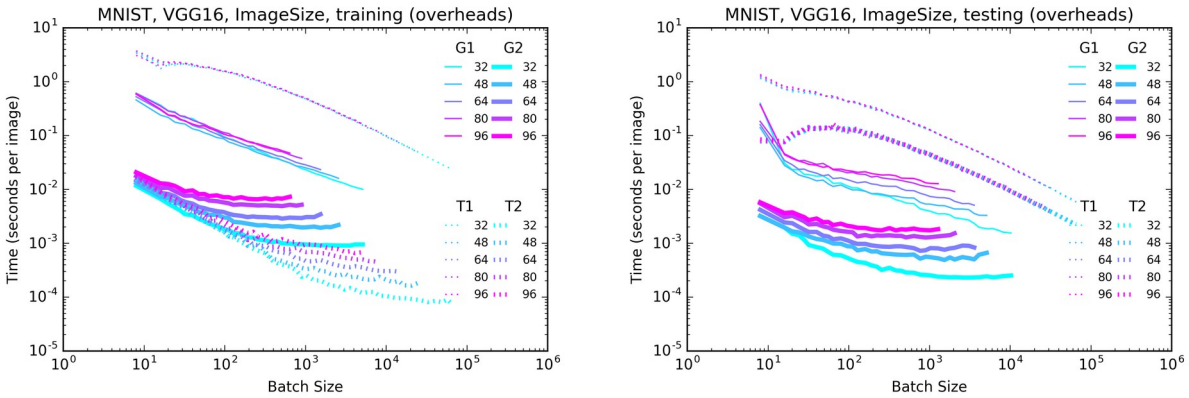

Рисунок 4.11 — Час виконання 1-ї та 2-ї ітерацій навчання (лівий) і виконання (правий) моделі *VGG16* в залежності від розміру порції даних

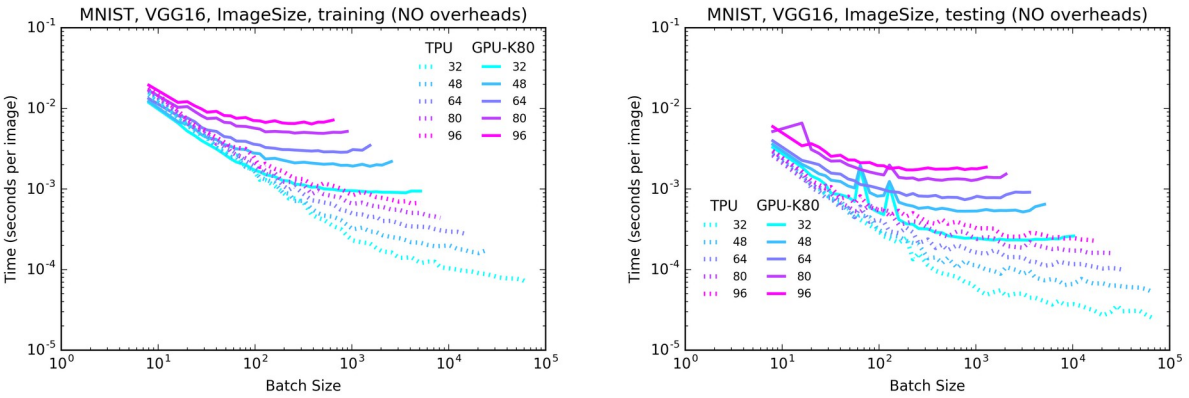

Рисунок 4.12 — Час виконання 3-ї ітерацій навчання (лівий) і виконання (правий) моделі *VGG16* в залежності від розміру порції даних

За даними часу виконання ітерацій на обох типах прискорювачів нейронних мереж було розраховано прискорення *TPU* у порівнянні із *GPU* в залежності від розміру порції вхідних даних (рис. 4.14-4.16).

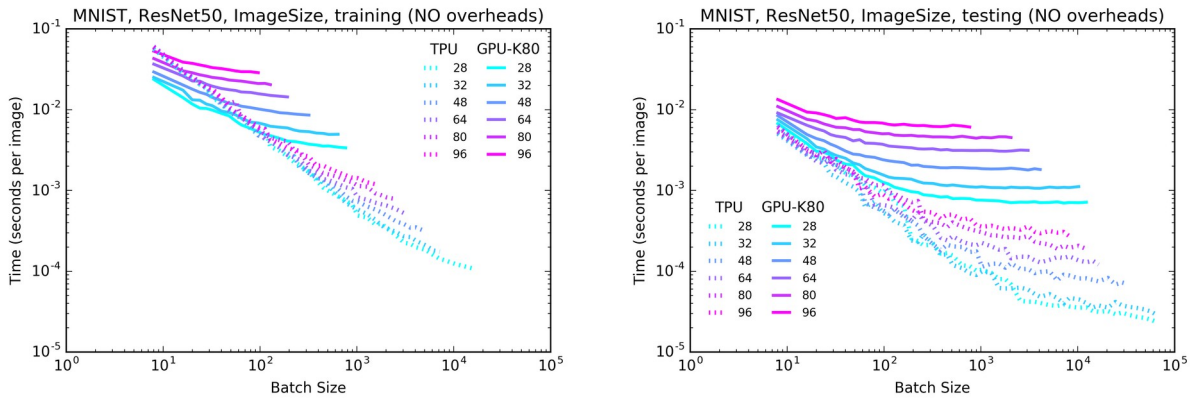

Рисунок 4.13 — Час виконання 3-ї ітерацій навчання (лівий) і виконання (правий) моделі *ResNet50* в залежності від розміру порції даних

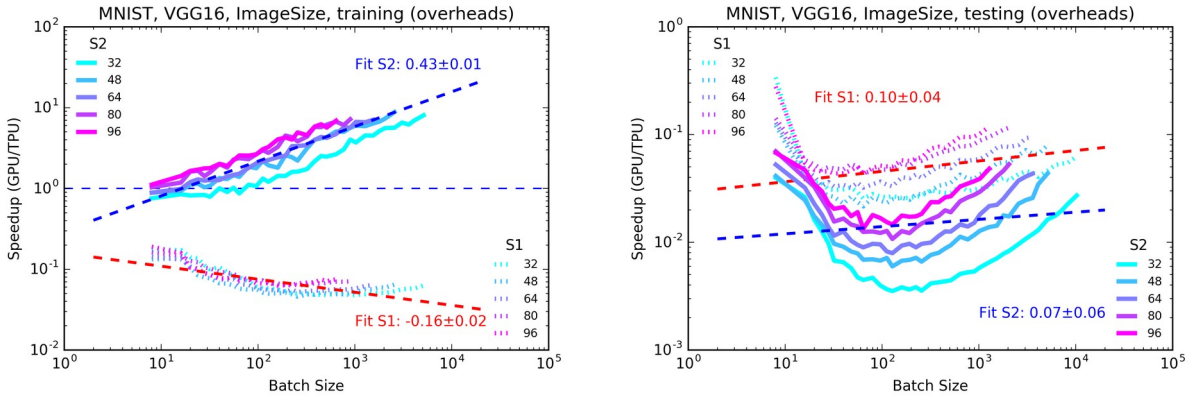

Рисунок 4.14 — Прискорення *TPU* на 1-й та 2-й ітераціях навчання (лівий) і виконання (правий) моделі *VGG16* в залежності від розміру порції даних

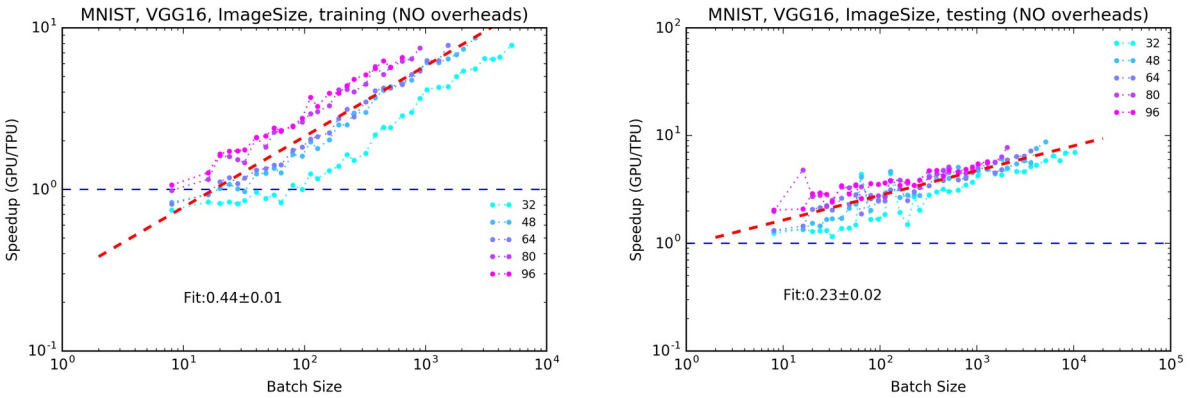

Рисунок 4.15 — Прискорення *TPU* на 3-й ітерації навчання (лівий) і виконання (правий) моделі *VGG16* в залежності від розміру порції даних

Отримані графіки прискорення підтверджують наявність великих накладних витрат – підвищеного часу виконання початкових ітерацій для прискорювача *TPU*, що призводить до значно меншої продуктивності спеціалізованого прискорювача на початкових ітераціях, порівняно із *GPU*. Також, при невеликих розмірах порції даних, прискорення *TPU* близьке до 1. Тому, для ефективнішої роботи прискорювача, необхідно обирати більші розміри порції даних, що дозволить використовувати внутрішній паралелізм обробки даних спеціалізованим прискорювачем більш ефективно.

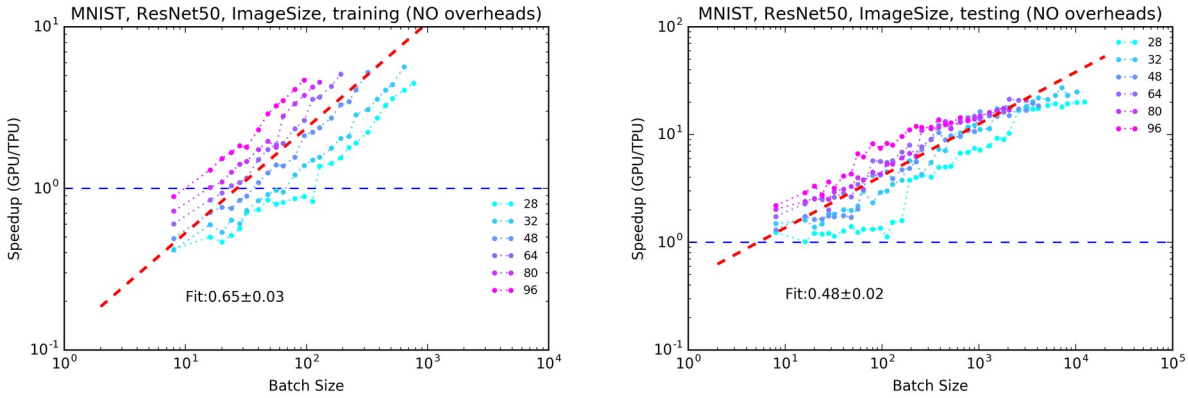

Рисунок 4.16 — Прискорення *TPU* на 3-й ітерації навчання (лівий) і виконання (правий) моделі *ResNet50* в залежності від розміру порції даних

За результатами аналізу технічних особливостей роботи спеціалізованих обчислювальних архітектур було виявлено, що значні показники прискорення при використанні *TPU* у порівнянні з *GPU* досягаються на пізніх ітераціях обробки даних моделями глибоких нейронних мереж, коли витрати на ініціалізацію не впливають на продуктивність. Слід зауважити відмінності в архітектурі *GPU* та *TPU*, що виражаються у різних затримках ініціалізації під час підготовки даних та компіляції моделі. На початкових 1-й та 2-й ітераціях *GPU* має меншу затримку, через що продуктивність *TPU* на початкових ітераціях не вища за продуктивність *GPU*.

4.3 Аналіз факторів, що впливають на продуктивність цільової інфраструктури обробки даних глибокими нейронними мережами

Для аналізу впливу факторів, які впливають на продуктивність обробки даних моделями глибоких нейронних мереж на цільовій інфраструктурі, було досліджено вплив апаратного та програмного забезпечення, що може бути наявним у кінцевого користувача. До такого забезпечення відносяться операційна система, інтерфейс підключення прискорювача нейронних мереж, наявний прискорювач, режим його роботи, тощо. В якості факторів, що розглядались були:

- Операційна система – *Windows*/*Linux*;
- Інтерфейс підключення – *USB*/*PCIe*;
- Прискорювач нейронних мереж – *Coral Edge TPU USB Stick*/*Horned Sungem*/*GPU NVIDIA GTX 1080 Ti*/*CPU*;
- Режим роботи прискорювача – «стандартний»/«*overclocking*».

Розглядались саме ці фактори, оскільки їх зміна можлива кінцевим користувачем без зайвих зусиль і, при цьому, може досягатись значне збільшення продуктивності обробки даних. Ще однією особливістю є те, що зазначені фактори не впливають на точність роботи моделі мережі, тому їхнє налаштування веде лише до підвищення продуктивності і ефективності обробки даних.

Тестування комбінацій зазначених параметрів виконувалось для різних розмірів вхідних даних – зображень. За результатами виконання обробки даних моделлю нейронної мережі було отримано час виконання ітерації та розраховано прискорення для різних комбінацій апаратного та програмного забезпечення цільової інфраструктури. В якості моделі нейронної мережі використовувалась *MobileNet v1 075* для задачі детекції постаті людини. До уваги брались додаткові накладні витрати на передачу даних, їх попередню обробку та опрацювання результатів.

Час виконання ітерації було виміряно для 4 різних типів прискорювачів: *CPU – Intel Core i5-3337U* (рис. 4.17), *GPU – NVIDIA GTX 1080 Ti* (рис. 4.18), *TPU – Coral Edge TPU* (рис. 4.19) та *Horned Sungem* (рис. 4.20).

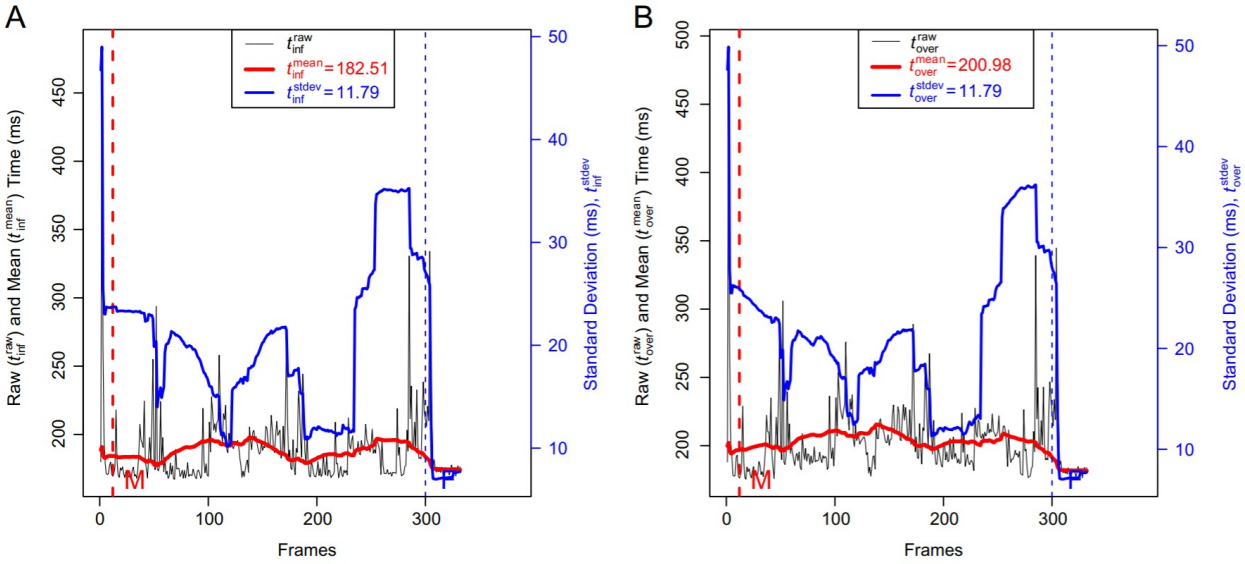

Рисунок 4.17 — Час виконання детекції (A) та повної ітерації (B) для *CPU*

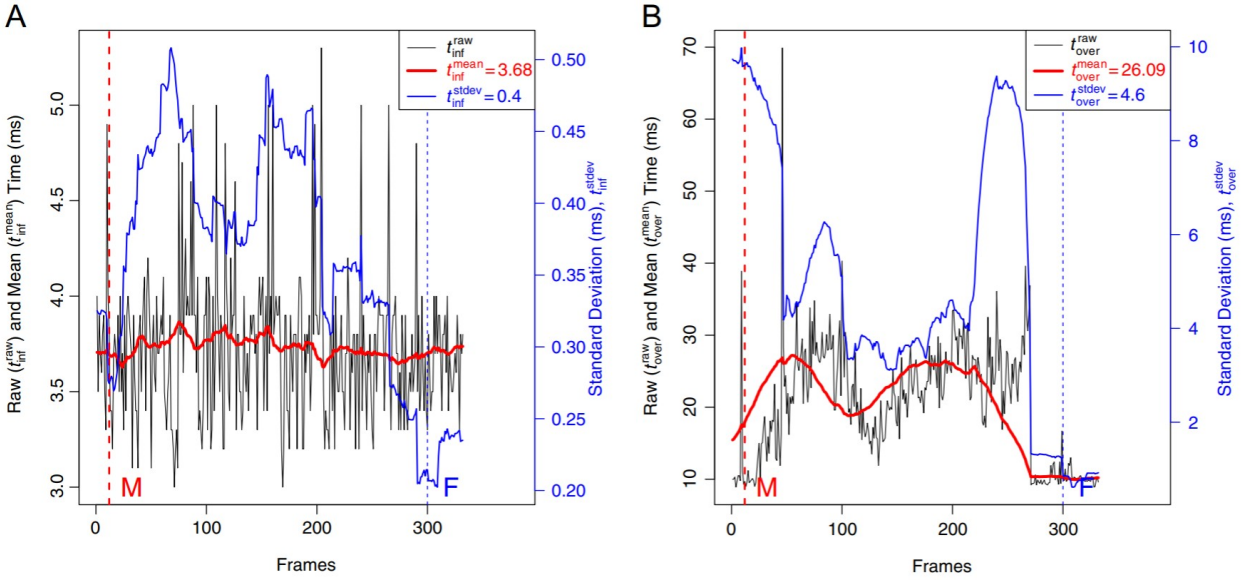

Рисунок 4.18 — Час виконання детекції (A) та повної ітерації (B) для *GPU*

Згідно рис. 4.17 та 4.18 повна ітерація, що включає в себе завантаження вхідних даних, їх попередню обробку, детекцію за допомогою моделі мережі та обробку результатів, займає більше часу ніж виконання детекції. Це справедливо для всіх типів прискорювачів. При цьому відсоток часу, що

витрачається на всю ітерацію, для більш швидких прискорювачів – *GPU*, *TPU* є набагато більшим у порівняні із *CPU*.

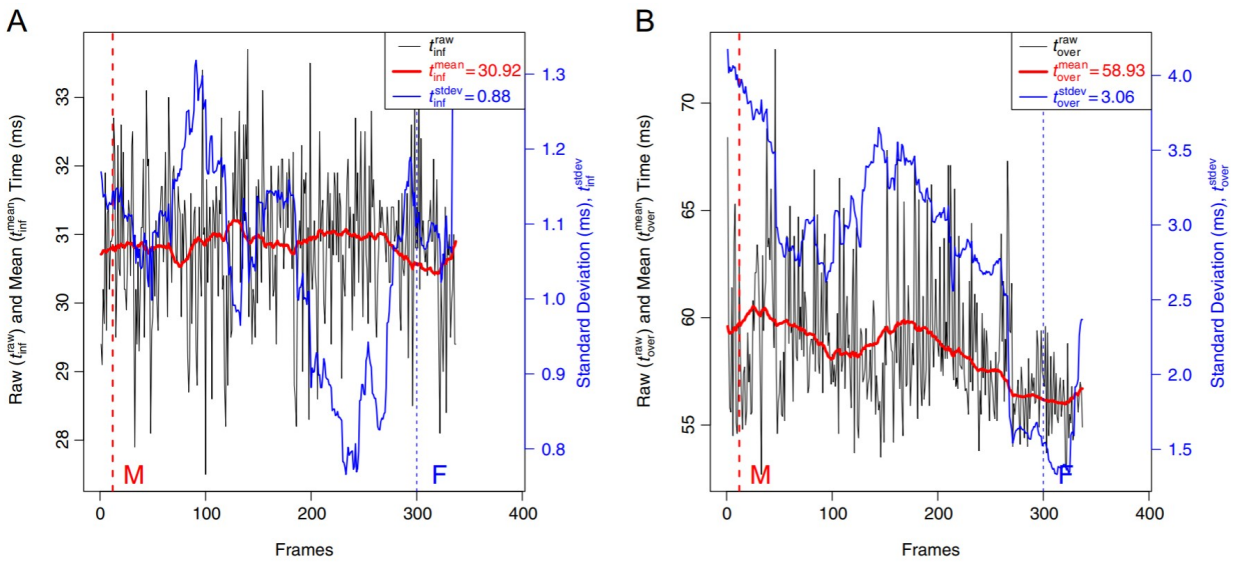

Рисунок 4.19 — Час виконання детекції (A) та повної ітерації (B) для *TPU – Coral Edge TPU*

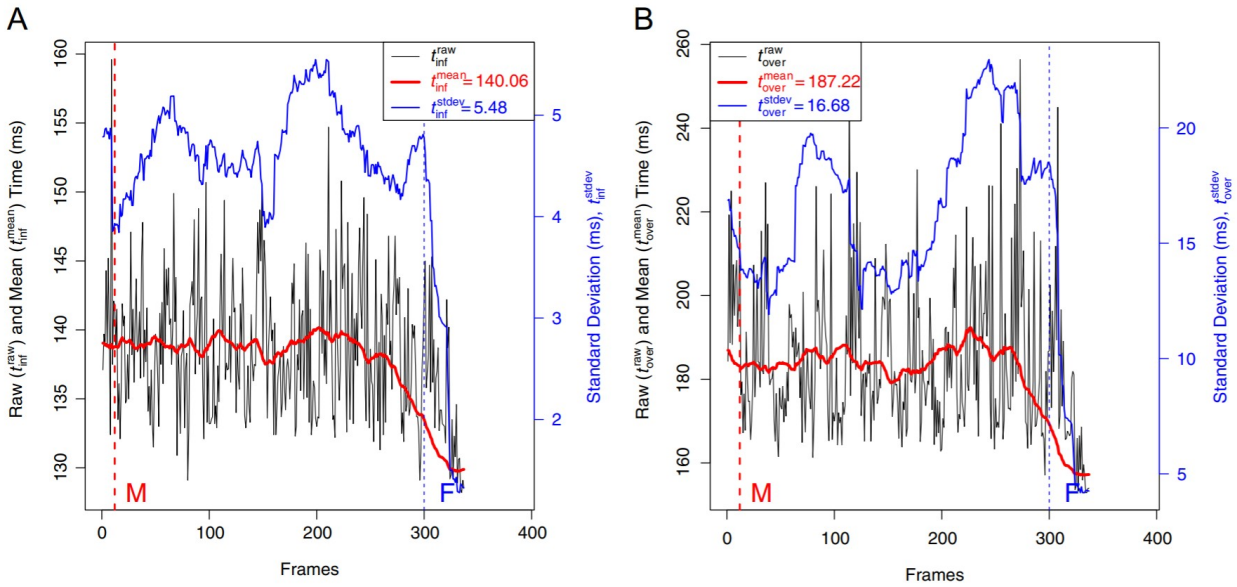

Рисунок 4.20 — Час виконання детекції (A) та повної ітерації (B) для *TPU – Horned Sungem*

Із порівняння графіків на рис. 4.18 та 4.19 можна зробити висновок, що, враховуючи усі накладні витрати часу, продуктивність роботи спеціалізованого прискорювача *Coral Edge TPU*, при певних умовах, не

уступає продуктивності *GPU – NVIDIA GTX 1080 Ti* в даній конкретній задачі. Середній час виконання детекції на *Coral Edge TPU* (рис. 4.19 А) є на 15% більший за час повної ітерації на *GPU* (рис. 4.18 В).

Для виявлення впливу зміни інтерфейсу підключення *TPU* прискорювача та операційної системи, на якій виконується програмне забезпечення моделі нейронної мережі, були проведені заміри часу при різних конфігураціях інфраструктури (рис. 4.21).

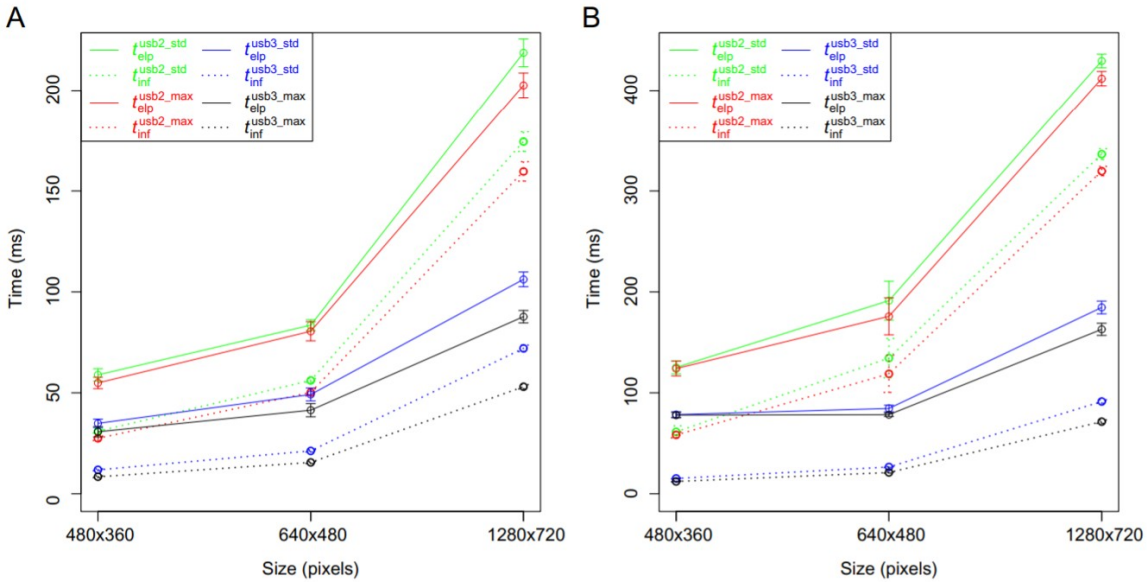

Рисунок 4.21 — Час виконання ітерації при використанні інтерфейсу підключення *USB2/USB3*, ОС *Linux* (А) та *Windows* (В) для прискорювача *Coral Edge TPU* в режимах роботи «стандартний» (*std*) та «*overclocking*» (*max*)

Із наведеного графіку видно, що зміна інтерфейсу підключення, на більш швидкий, призводить до зменшення часу ітерації. Також, ця закономірність справедлива і для використання більш швидкого режиму роботи прискорювача. Окремо слід відмітити, що виконання одного і того самого сценарію займає більше часу при використані прискорювача на ОС *Windows*. Це може пояснюватись меншою ефективність бібліотек виконання нейронних мереж та середовища виконання для спеціалізованого прискорювача під дану ОС.

Прискорення від використання ОС *Linux* у порівнянні з ОС Windows, а також вплив режимів «стандартний» (*std*) та «*overclocking*» (*max*) для *Coral Edge TPU*, показано на рис. 4.22. Узагальнений графік прискорення, що досягається при різних комбінаціях забезпечення інфраструктури, а саме зміна інтерфейсу підключення, ОС, режимів роботи прискорювача, наведено на рис. 4.23.

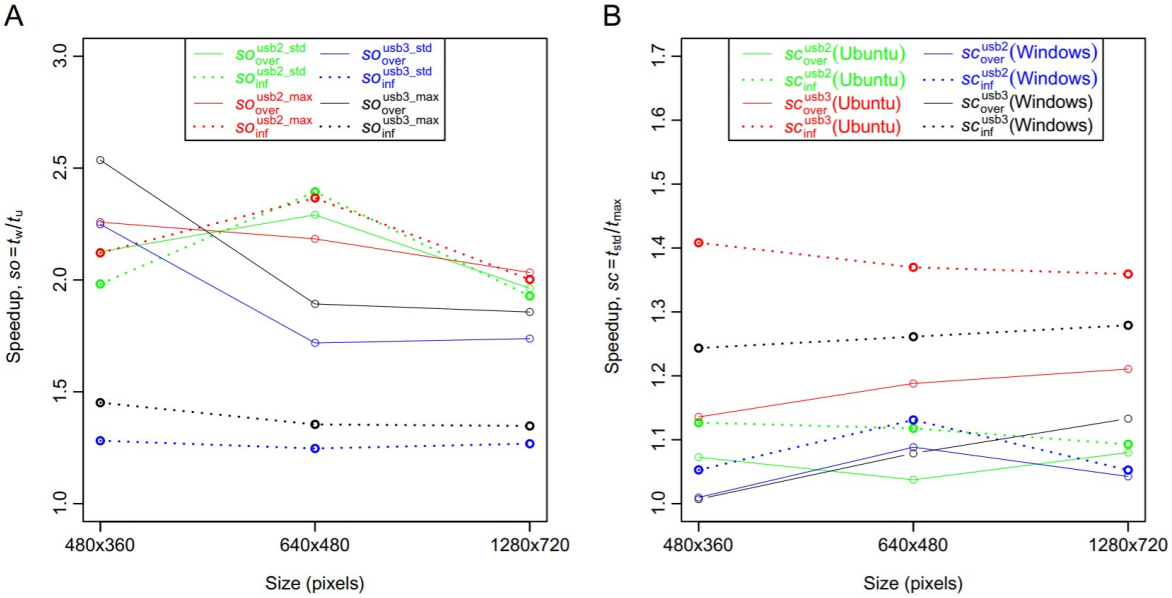

Рисунок 4.22 — Прискорення від використання ОС *Linux* у порівнянні з ОС *Windows* (A) та режимів «стандартний» (*std*) та «*overclocking*» (*max*) (B) для *Coral Edge TPU*

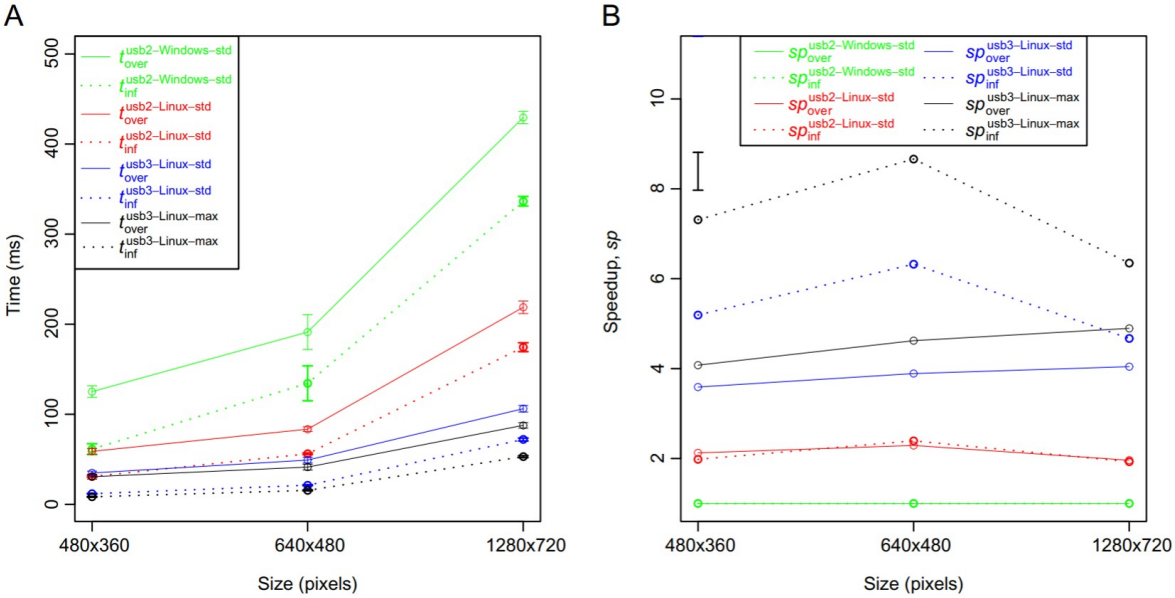

Рисунок 4.23 — Час виконання ітерації (A) та прискорення (B) для *Coral Edge TPU* при різних комбінаціях налаштування інфраструктури

Величини прискорення, що досягаються при різних комбінаціях апаратного та програмного забезпечення цільової інфраструктури, наведено у таблиці 4.3.

Таблиця 4.3 — Прискорення при різних конфігураціях інфраструктури

Configurations				Frame size		
OS	I/O	Overclocking	Running stage	Small (480 × 360)	Middle (640 × 480)	Large (1280 × 720)
Windows	USB 2.0	Std	Over	1	1	1
Windows	USB 2.0	Std	Inf	1	1	1
Ubuntu	USB 2.0	Std	Over	2.13	2.29	1.96
Ubuntu	USB 2.0	Std	Inf	1.98	2.39	1.93
Ubuntu	USB 3.0	Std	Over	3.59	3.89	4.04
Ubuntu	USB 3.0	Std	Inf	5.19	6.32	4.67
Ubuntu	USB 3.0	Max	Over	4.08	4.62	4.90
Ubuntu	USB 3.0	Max	Inf	7.31	8.66	6.35

В результаті аналізу факторів, що впливають на продуктивність цільової інфраструктури обробки даних глибокими нейронними мережами, було досягнуто суттєвих різниць в продуктивності при роботі з різними комбінаціями забезпечення цільової інфраструктури. Зміна операційної системи на *Linux* дозволяє отримати прискорення у 2,13 рази, подальша зміна інтерфейсу підключення на *USB3* дає додаткові 4,04, а також використання спеціального режиму підвищеної продуктивності для *Coral Edge TPU* надає додаткові 4. Загалом, було досягнуто прискорення в межах 6,35-8,66 при різних конфігураціях обчислювальної інфраструктури.

4.4 Аналіз продуктивності роботи спеціалізованого прискорювача в практичних умовах в медичному контексті

Використані в роботі спеціалізовані прискорювачі *TPU* мають невелику вартість, у порівняні з високопродуктивними графічними процесорами *GPU*, і потребують незначних змін у системі для їх використання. Вони працюють, як допоміжні процесори, і можуть бути підключеними до основної системи через універсальну послідовну шину *USB3*, що забезпечує пропускну здатність даних на рівні 5 Gbps. Враховуючи, що *Coral Edge TPU USB Stick* витрачає всього 2 Вт енергії, це дає змогу використовувати даний прискорювач у задачах периферійних обчислень – *edge computing*, де присутні високі обмеження на обчислювальні ресурси та енергоефективність. У таблиці 4.4 приведено порівняння продуктивності роботи *TPU* прискорювача *Coral Edge TPU USB Stick* та графічного процесора *NVIDIA A100* при використанні моделі нейронної мережі *ResNet50*.

Таблиця 4.4 — Порівняння продуктивності роботи *Coral Edge TPU USB Stick* та *NVIDIA A100*

Model	ResNet 50			
Accelerator	NVIDIA A100			Coral Edge TPU
Number format	FP32	FP16 (TC)	INT 8 (TC)	INT8
FPS	853,09 [116]	2357,09 [116]	-	15,15
TFLOPS	19,5 [117]	312 [117]	-	-
TOPS	-	-	624 [117]	4 [2]
TOPS/W	-	-	**2,4960**	2
TOPS/USD	-	-	0,0499	**0,0533**
FPS/W	3,4124	**9,4284**	-	7,5750
FPS/USD	0,0682	0,1885	-	**0,2020**
Interface	PCI-e [117]			USB 3.0 [2]
VRAM GB	40 [117]			0,0078
TDP	250 [117]			2 [2]
Price USD	12500,00 [118]			74,99 [2]

З порівняльної таблиці слідує, що *NVIDIA A100* має більші абсолютні показники продуктивності у порівняні з *Coral Edge TPU*. Абсолютна продуктивність *NVIDIA A100* для розрядності *FP16* та використанням

NVIDIA Tensor Cores в 155 разів більша у порівняні із продуктивністю *Coral Edge TPU* для розрядності *INT8*. При цьому, слід відмітити більші енерговитрати та ціну для *NVIDIA A100*, які більші у 125 разів та 166 разів відповідно. Аналізуючи відносну продуктивність прискорювачів слід зауважити, що маючи набагато меншу потужність, *Coral Edge TPU* у певних сценаріях має більшу ефективність ніж *NVIDIA A100*. Порівнюючи продуктивність пристроїв, враховуючи витрати енергії, *Coral Edge TPU* має всього на 24,5% меншу продуктивність ніж *NVIDIA A100*. Враховуючи низьку ціну *Coral Edge TPU*, то у порівнянні з *NVIDIA A100*, він має на 7,1% більшу відносну продуктивність. Як результат порівняння обох прискорювачів слідує, що для задач *Edge Computing* використання *Coral Edge TPU* є обумовленим через його портативність, високу енергоефективність, продуктивність та низьку ціну.

Для подальшого аналізу продуктивності роботи спеціалізованого прискорювача *Coral Edge TPU* було проведено заміри часу виконання ітерації в тестовій портативній програмі для діагностики легеневих аномалій. За результатами роботи програми були зібрані дані часу обробки кожного знімка в залежності від конфігурації обчислювальної системи:

- виконання на *CPU* – при відсутності прискорювача *Edge TPU* (рис. 4.24-4.27 ліві);
- виконання на *Edge TPU* (рис. 4.24-4.27 праві).

Враховуючи, що в ролі цільової системи, на якій виконується програма, виступає звичайний персональний комп’ютер лікаря, можлива ситуація, коли разом із програмою діагностики будуть паралельно виконуватись інші програми лікаря або лікар буде працювати із інтерфейсами вводу/виводу даних комп’ютеру. Це може впливати на загальну продуктивність роботи програми діагностики за рахунок того, що ресурси системи будуть розділятись диспетчером ОС між паралельними процесами або висока завантаженість інтерфейсами вводу/виводу таких, як *USB*, призведе до зниження швидкості обміну даними між прискорювачем *EdgeTPU*, що також

вплине на швидкість обробки рентген зображень. Для дослідження цього впливу було проведено додаткові заміри часу обробки зображень в у мовах різної завантаженості системи:

- відсутність іншого навантаження окрім роботи програми діагностики – *no load* (рис. 4.24);
- навантаження за рахунок роботи користувача із системою – 40% *user load* (4.25);
- виконання паралельної задачі – 60% *load* (рис. 4.26);
- виконання паралельної задачі – 100% *load* (рис. 4.27).

В ході аналізу отриманих результатів було відмічено підвищений час обробки 1-го знімку, що зумовлене додатковими службовими витратами на ініціалізацію моделі нейронної мережі та даних, характерних для прискорювачів зі спеціалізованою архітектурою. Тому, отримані графіки заміри часу обробки були додатково поділено:

- при наявності витрат ініціалізації при обробці 1-го знімку – *with overheads* (рис. 4.28);
- без витрат ініціалізації для знімків > 2-го – *no overheads* (рис. 4.24-4.27).

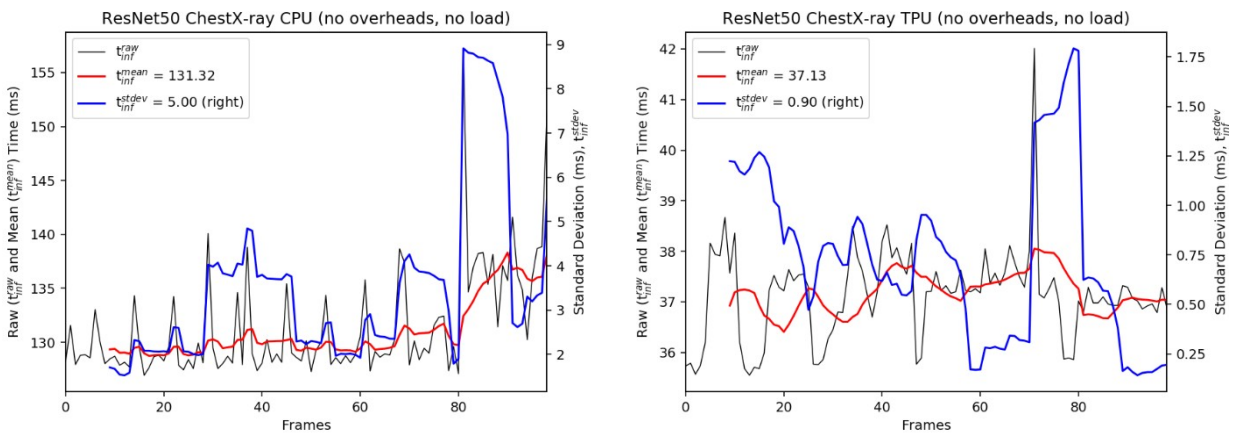

Рисунок 4.24 — Графік часу обробки рентген зображення на *CPU* (лівий) та *TPU* (правий) без витрат ініціалізації, без навантаження на систему

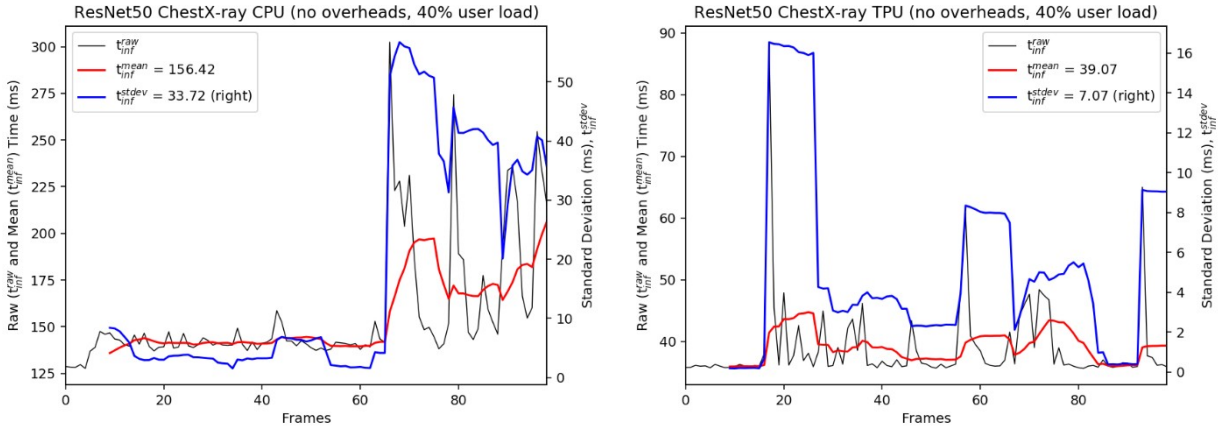

Рисунок 4.25. Графік часу обробки рентген зображення на *CPU* (лівий) та *TPU* (правий) без витрат ініціалізації, із навантаженням користувача на систему

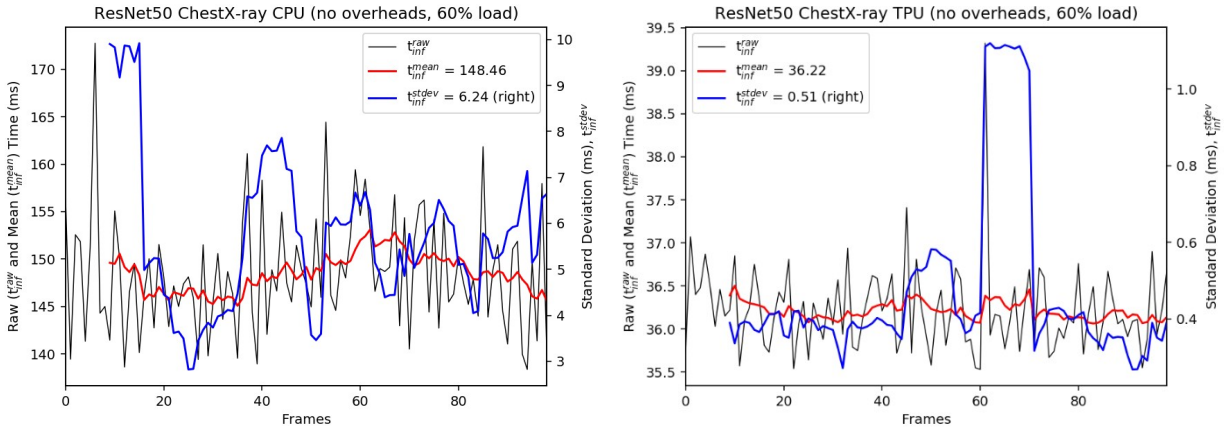

Рисунок 4.26 — Графік часу обробки рентген зображення на *CPU* (лівий) та *TPU* (правий) без витрат ініціалізації, із паралельним навантаженням 60% на систему

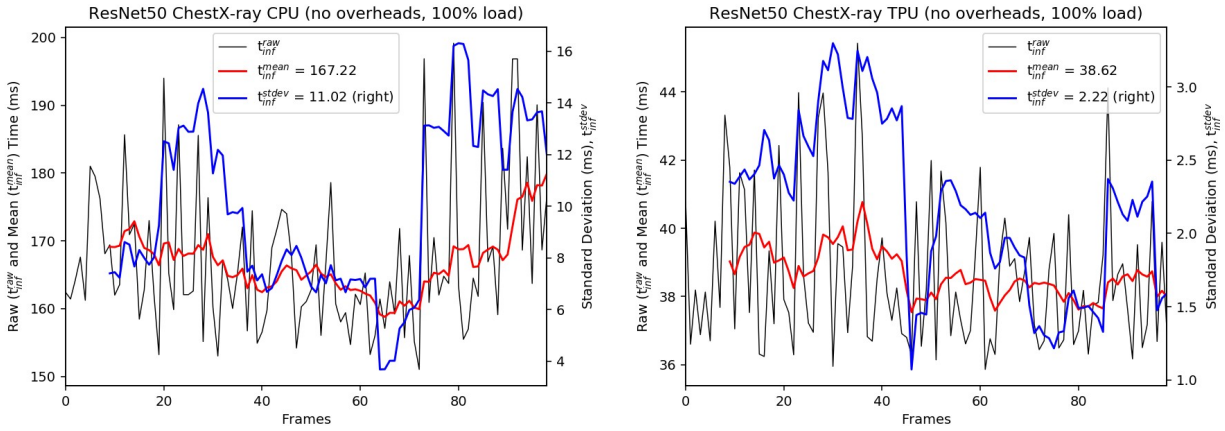

Рисунок 4.27 — Графік часу обробки рентген зображення на *CPU* (лівий) та *TPU* (правий) без витрат ініціалізації, із паралельним навантаженням 100% на систему

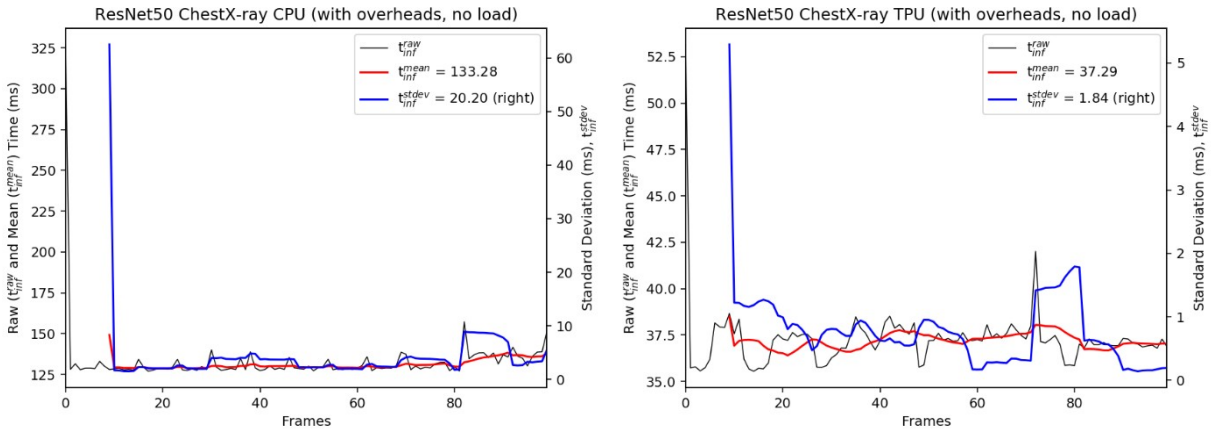

Рисунок 4.28 — Графік часу обробки рентген зображення на *CPU* (лівий) та *TPU* (правий) із витратами на ініціалізацію, без навантаження на систему

Для *CPU* час обробки першого кадру – першої ітерації, який включає витрати на ініціалізацію, в 2,49 рази більший за час оброки при наступних ітераціях. Збільшений час першої ітерації є характерним і для виконання програми на прискорювачі *EdgeTPU* – в 1,43 рази більший за час обробки подальших ітерацій. При роботі із системою користувач має більший пріоритет, тому ОС може виділяти більше ресурсів процесам користувача, що впливатиме на продуктивність програми діагностики. Для даного режиму відхилення часу обробки від середнього *stdev* було найбільшим як для конфігурації із *CPU*, так для конфігурації із *EdgeTPU*, і становило 33,72 мс та

7,07 мс відповідно. При паралельному навантаженні у 60% на систему, середній час, у порівняні з системою без навантаження, був на 13,2% більший при виконанні на CPU. При повному навантажені на систему, середній час збільшився на 27,3% та 4% відповідно, у порівнянні із системою без навантаження, для *CPU* та *TPU*. Узагальнені результати підтверджують, що паралельне навантаження на систему має мінімальний вплив на час обробки даних нейронною мережею на прискорювачі *EdgeTPU*, оскільки він виступає у ролі додаткового процесору для основної системи і виконує задачі нейронної мережі незалежно від процесора основної системи.

4.5 Висновки до розділу 4

В даному розділі було проведено аналіз результатів застосування методу адаптації глибоких нейронних мереж, що включає в себе адаптивний ітеративний прунінг, підвищення ефективності процесу обробки даних мережею та підвищення ефективності програмно-апаратної складової цільової інфраструктури.

За результатами застосування методу адаптивного ітеративного прунінгу було проаналізовано ефективність розробленого методу у порівняні зі звичайним методом прунінгу. Розроблений метод базується на адаптивній зміні гіперпараметрів таких, як швидкість навчання, розмір порції даних, кількість епох, рівень прунінгу. Це дозволяє компенсувати втрати точності за рахунок додаткового навчання мережі з вагами, що залишилися після чергової ітерації прунінгу. В ході тестування на архітектурі *VGG-16* для задачі класифікації на наборі даних *MNIST* було продемонстровано наступні показники:

- Після адаптивного ітеративного прунінгу досягається прискорення у 15,2 раз при втраті точності менше 3% (4 ітерації прунінгу);
- Після адаптивного ітеративного прунінгу досягається прискорення у 30,6 раз при втраті точності менше 4% (7 ітерацій прунінгу);

- Після адаптивного ітеративного прунінгу було досягнуто максимального прискорення 32,2 із точність роботи 96,2% (10 ітерацій прунінгу);
- Після звичайного прунінгу і навчання мережі було досягнуто прискорення 13 і точність роботи 98,6% (3 ітерації прунінгу та 63 епохи навчання).

За результатами аналізу технічних особливостей роботи спеціалізованих обчислювальних архітектур було виявлено, що значні показники прискорення при використанні *TPU*, у порівнянні з *GPU*, досягаються на пізніх ітераціях обробки даних моделями глибоких нейронних мереж, коли витрати на ініціалізацію не впливають на продуктивність. Відмінності в архітектурі *GPU* та *TPU* виражаються у різних затримках ініціалізації під час підготовки даних та компіляції моделі. На початкових 1-й та 2-й ітераціях *GPU* має меншу затримку, через що продуктивність *TPU* на початкових ітераціях не вища за продуктивність *GPU*. Тому, в ході підвищення ефективності процесу обробки даних, важливо враховувати дані особливості роботи спеціалізованих прискорювачів.

В результаті аналізу факторів, що впливають на продуктивність цільової інфраструктури для глибоких нейронних мереж було досягнуто видимих різниць в продуктивності при роботі з різними комбінаціями забезпечення цільової інфраструктури. Зміна операційної системи на *Linux* дозволяє отримати прискорення у 2,13 рази, подальша зміна інтерфейсу підключення на *USB3* дає додаткові 4,04, а також використання спеціального режиму підвищеної продуктивності для *Coral Edge TPU* надає додаткові 4. Загалом було досягнуто прискорення в межах 6,35-8,66 при різних конфігураціях обчислювальної інфраструктури.

Узагальнюючи отримані результати, можна зробити висновок, що використання розробленого методу адаптації глибоких нейронних мереж дозволяє зменшити час обробки даних за допомогою глибоких нейронних мереж на спеціалізованих обчислювальних архітектурах.

ВИСНОВКИ

В дисертаційній роботі розв’язане актуальне наукове завдання адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур. При цьому, отримано наступні наукові та практичні результати:

1. Проведено аналіз сучасних технологій модифікації глибоких нейронних мереж для підвищення продуктивності обробки даних на основі методів прунінгу та квантизації, що дозволяють зменшувати розмір та обчислювальну складність моделей глибоких нейронних мереж, при цьому зменшуючи втрати точності їх роботи у порівняні із повноцінними початковими моделями.
2. Досліджено фактори, що впливають на процес адаптації та підвищення ефективності обробки даних глибокими нейронними мережами із застосуванням спеціалізованих прискорювачів, до яких відносяться: висока обчислювальна складність глибоких нейронних мереж, що містять згорткові шари – *CNN*; необхідність враховувати внутрішню архітектуру спеціалізованих прискорювачів для нейронних мереж, щоб пришвидшити обробку даних мережею та покращити енергоефективність в цілому; додаткові накладні витрати на ініціалізацію моделі мережі, завантаження та обробку даних на спеціалізованих прискорювачах; конфігурація апаратної та програмної складової цільової обчислювальної інфраструктури на якій застосовується спеціалізований прискорювач.
3. Розроблено комплексний метод адаптації глибоких нейронних мереж для спеціалізованих обчислювальних архітектур.
4. Розроблено метод адаптивного ітеративного прунінгу для зменшення розміру моделей нейронних мереж, що базується у поступовому зменшені розміру мережі, шляхом видалення зайвих каналів у шарах, та додатковому навчанні отриманої зменшеної моделі для відновлення точності роботи. Відповідно до методу, гіперпараметри мережі

адаптивно змінюються, щоб компенсувати втрати точності після кожної ітерації прунінгу та досягти обраного критерію покращення такого, як прискорення.

5. Розроблено метод підвищення ефективності процесу обробки даних нейронними мережами на спеціалізованих обчислювальних архітектурах, що враховує технічні особливості обробки даних глибокими нейронними мережами на спеціалізованих прискорювачах, наприклад, ітерація виконання обчислень, та дозволяє визначити параметри такі, як розмір порції даних, що дає можливість збільшити продуктивність обробки даних мережею за рахунок зменшення впливу накладних витрат ініціалізації та передачі даних.

6. Розроблено метод підвищення ефективності обчислювальної інфраструктури для глибоких нейронних мереж, що дозволяє налаштувати програмну та апаратну складові такі, як ОС та інтерфейси підключення, що дозволяє збільшити продуктивність обробки даних за допомогою нейронних мереж на цільовій системі.

7. Розроблено програмний компонент діагностики легеневих аномалій за даними рентген знімків для дослідження продуктивності роботи спеціалізованого прискорювача *Coral Edge TPU USB* в медичному контексті. В якості архітектури глибокої нейронної мережі для даної задачі було обрано *ResNet50*, що було треновано на наборі даних *ChestXray* та адаптовано під спеціалізований прискорювач, відповідно до розробленого комплексного методу адаптації.

8. Проведено аналіз результатів застосування комплексного методу адаптації глибоких нейронних мереж, що включає в себе адаптивний ітеративний прунінг, підвищення ефективності процесу обробки даних мережею та підвищення ефективності програмно-апаратної складової цільової обчислювальної інфраструктури. За результатами застосування розробленого методу адаптивного ітеративного прунінгу було досягнуто прискорення у 32,2 рази із точність роботи 96,2% (10

ітерацій прунінгу). За результатами аналізу технічних особливостей роботи спеціалізованих обчислювальних архітектур було виявлено, що значні показники прискорення при використанні *TPU*, у порівнянні з *GPU*, досягаються на пізніх ітераціях обробки даних моделями глибоких нейронних мереж, коли витрати на ініціалізацію не впливають на продуктивність. Даний фактор треба враховувати при підвищенні ефективності процесу обробки даних за допомогою нейронних мереж на прискорювачах зі спеціалізованою архітектурою. В результаті аналізу факторів, що впливають на продуктивність цільової обчислювальної інфраструктури для глибоких нейронних мереж, було досягнуто значних різниць в продуктивності при роботі з різними комбінаціями забезпечення обчислювальної інфраструктури, при цьому досягнуте прискорення було в межах 6,35-8,66.

Розроблені методи є складовою комплексного методу адаптації глибоких нейронних мереж і дозволяють підготувати обрані моделі нейронних мереж для їх застосування на прискорювачах нейронних мереж зі спеціалізованою архітектурою. Це було перевірено в практичних умовах для медичного застосування.

ЛІТЕРАТУРА

1. NVIDIA Jetson Nano Developer Kit [Електронний ресурс]. – Режим доступу: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

2. Coral Edge TPU USB Accelerator [Електронний ресурс]. – Режим доступу: https://coral.ai/products/accelerator#description

3. Intel Neural Compute Stick 2 [Електронний ресурс]. – Режим доступу: https://ark.intel.com/content/www/us/en/ark/products/140109/intel-neural-compute-stick-2.html

4. Horned Sungem TPU [Електронний ресурс]. – Режим доступу: https://hornedsungem.github.io/Docs/en/product/

5. Hisilicon HiKey 970 [Електронний ресурс]. – Режим доступу: https://www.96boards.org/product/hikey970/

6. NVIDIA Tensor Cores [Електронний ресурс]. – Режим доступу: https://developer.nvidia.com/tensor-cores

7. Jouppi, N. P., et al. (2017, June). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture* (pp. 1-12).

8. Antonini, M., Vu, T. H., Min, C., Montanari, A., Mathur, A., & Kawsar, F. (2019, November). Resource characterisation of personal-scale sensing models on edge accelerators. In *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things* (pp. 49-55).

9. Berthelier, A., Chateau, T., Duffner, S., Garcia, C., & Blanc, C. (2021). Deep model compression and architecture optimization for embedded systems: A survey. *Journal of Signal Processing Systems*, *93*(8), 863-878.

10. Vasquez, K., Venkatesha, Y., Bhattacharjee, A., Moitra, A., & Panda, P. (2021, February). Activation density based mixed-precision quantization for

energy efficient neural networks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (pp. 1360-1365). IEEE.

11. Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

12. Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, *28*.

13. Mallya, A., & Lazebnik, S. (2018). Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (pp. 7765-7773).

14. Yu, R., Li, A., Chen, C. F., Lai, J. H., Morariu, V. I., Han, X., ... & Davis, L. S. (2018). Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 9194-9203).

15. Stamoulis, D., Chin, T. W., Prakash, A. K., Fang, H., Sajja, S., Bognar, M., & Marculescu, D. (2018, November). Designing adaptive neural networks for energy-constrained image classification. In *Proceedings of the International Conference on Computer-Aided Design* (pp. 1-8).

16. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436-444.

17. Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, *61*, 85-117.

18. Wang, H., & Raj, B. (2017). On the origin of deep learning. *arXiv preprint arXiv:1702.07800*.

19. Bengio, Y. (2013, July). Deep learning of representations: Looking forward. In *International conference on statistical language and speech processing* (pp. 1-37). Springer, Berlin, Heidelberg.

20. Lacey, G., Taylor, G. W., & Areibi, S. (2016). Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283*.

21. Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., ... & Boudoukh, G. (2017, February). Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays* (pp. 5-14).

22. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., & Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, *42*(1), 269-284.

23. Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., ... & Modha, D. S. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, *34*(10), 1537-1557.

24. Ienne, P. (1993). Architectures for neuro-computers: review and performance evaluation. *Computer Science Department Technical Report No. 93*, *21*.

25. Tensor Cores: Versatility for HPC & AI [Електронний ресурс]. – Режим доступу: https://www.nvidia.com/en-us/data-center/tensor-cores/

26. Cloud Tensor Processing Units (TPUs) [Електронний ресурс]. – Режим доступу: https://cloud.google.com/tpu/docs/tpus

27. Ben-Nun, T., & Hoefler, T. (2019). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, *52*(4), 1-43.

28. Shimoda, M., Sada, Y., Kuramochi, R., Sato, S., & Nakahara, H. (2020). SENTEI: Filter-Wise Pruning with Distillation towards Efficient Sparse

Convolutional Neural Network Accelerators. *IEICE TRANSACTIONS on Information and Systems*, *103*(12), 2463-2470.

29. Hoffmann, J., Agnihotri, S., Saikia, T., & Brox, T. (2021). Towards improving robustness of compressed CNNs. In *ICML Workshop on Uncertainty and Robustness in Deep Learning (UDL)*.

30. Jiang, C., Ojika, D., Patel, B., & Lam, H. (2021, May). Optimized fpga-based deep learning accelerator for sparse cnn using high bandwidth memory. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 157-164). IEEE.

31. Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019, July). Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 33, No. 01, pp. 4780-4789).

32. Sze, V., Chen, Y. H., Yang, T. J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, *105*(12), 2295-2329.

33. Tang, W., Wei, X., & Li, B. (2020). Automated Model Compression by Jointly Applied Pruning and Quantization. *arXiv preprint arXiv:2011.06231*.

34. Finlinson, A., & Moschoyiannis, S. (2020, October). Synthesis and Pruning as a Dynamic Compression Strategy for Efficient Deep Neural Networks. In *International Symposium: From Data to Models and Back* (pp. 3-17). Springer, Cham.

35. Li, Y., Pintea, S. L., & van Gemert, J. (2020). Less bits is more: How pruning deep binary networks increases weight capacity.

36. Luo, W. (2020). Improving Neural Network With Uniform Sparse Connectivity. *IEEE Access*, *8*, 215705-215715.

37. Hadifar, A., Deleu, J., Develder, C., & Demeester, T. (2021). Exploration of block-wise dynamic sparseness. *Pattern Recognition Letters*, *151*, 187-192.

38. Passov, E., David, E. O., & Netanyahu, N. S. (2021, September). Gator: Customizable Channel Pruning of Neural Networks with Gating. In *International Conference on Artificial Neural Networks* (pp. 46-58). Springer, Cham.

39. Huang, Z., Li, L., & Sun, H. (2020). Global Biased Pruning Considering Layer Contribution. *IEEE Access*, *8*, 173521-173529.

40. Haider, M. U., & Taj, M. (2021, September). Comprehensive online network pruning via learnable scaling factors. In *2021 IEEE International Conference on Image Processing (ICIP)* (pp. 3557-3561). IEEE.

41. Wang, E., Davis, J. J., Zhao, R., Ng, H. C., Niu, X., Luk, W., ... & Constantinides, G. A. (2019). Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Computing Surveys (CSUR)*, *52*(2), 1-39.

42. Buyya, R., & Srirama, S. N. (Eds.). (2019). *Fog and edge computing: principles and paradigms*. John Wiley & Sons.

43. Intel Movidius Myriad X [Електронний ресурс]. – Режим доступу: https://ark.intel.com/content/www/us/en/ark/products/125926/intel-movidius-myriad-x-vision-processing-unit-4gb.html

44. Wang, Y. (2015). Cloud-dew architecture. *International Journal of Cloud Computing*, *4*(3), 199-210.

45. Skala, K., Davidovic, D., Afgan, E., Sovic, I., & Sojat, Z. (2015). Scalable distributed computing hierarchy: Cloud, fog and dew computing. *Open Journal of Cloud Computing (OJCC)*, *2*(1), 16-24.

46. Gordienko, Y., Stirenko, S., Alienin, O., Skala, K., Sojat, Z., Rojbi, A., ... & Jervan, G. (2017, May). Augmented coaching ecosystem for non-obtrusive adaptive personalized elderly care on the basis of cloud-fog-dew computing paradigm. In *2017 40th International Convention on Information*

and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 359-364). IEEE.

47. Murshed, M. S., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., & Hussain, F. (2021). Machine learning at the network edge: A survey. *ACM Computing Surveys (CSUR)*, *54*(8), 1-37.

48. Zhang, B., Davoodi, A., & Hu, Y. H. (2020). CHaPR: Efficient Inference of CNNs via Channel Pruning. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)* (pp. 1-6). IEEE.

49. Miao, J., Sun, K., Liao, X., Leng, L., & Chu, J. (2020). Human segmentation based on compressed deep convolutional neural network. *IEEE Access*, *8*, 167585-167595.

50. Shih, H. T., & Chang, T. S. (2020, October). Zebra: Memory Bandwidth Reduction for CNN Accelerators with Zero Block Regularization of Activation Maps. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (pp. 1-5). IEEE.

51. Li, G., Wang, J., Shen, H. W., Chen, K., Shan, G., & Lu, Z. (2020). Cnnpruner: Pruning convolutional neural networks with visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, *27*(2), 1364-1373.

52. Pei, S., Wu, Y., & Qiu, M. (2020, October). Neural network compression and acceleration by federated pruning. In *International Conference on Algorithms and Architectures for Parallel Processing* (pp. 173-183). Springer, Cham.

53. Toliupa, S., Tereikovska, L., Tereikovskyi, I., Doszhanova, A., & Alimseitova, Z. (2020, October). Procedure for adapting a neural network to eye iris recognition. In 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T) (pp. 167-171). IEEE.

54. Кулаков, Ю. О., Терейковська, Л. О., & Терейковський, І. А. (2019). Спосіб застосування згорткової нейронної мережі для розпізнавання особи і емоцій користувача за клавіатурним почерком. Правове, нормативне та метрологічне забезпечення системи захисту інформації в Україні, (2 (38)), 9-17.

55. Tereykovskaya, L., Ivanchenko, Y., Pogorelov, V. (2019). Метод адаптації глибокої нейронної мережі до розпізнавання комп’ютерних вірусів. COMPUTER-INTEGRATED TECHNOLOGIES: EDUCATION, SCIENCE, PRODUCTION, (35), 198-205.

56. Яковина, В. С., Симець, І. І. (2021). Прогнозування дефектів програмного забезпечення ансамблем нейронних мереж. Науковий вісник НЛТУ України, 31(6), 104-111.

57. Vesela, M., Klymenko, I., Melnikova, Y. (2021). ВИКОРИСТАННЯ НЕЙРОННИХ МЕРЕЖ В АДАПТИВНОМУ УПРАВЛІННІ ЕЛЕКТРОМОБІЛЕМ. Транспортні системи і технології, (38), 213-222.

58. Kuzmych, V., Novotarskyi, M. (2022). Accelerating Simulation of the PDE Solution by the Structure of the Convolutional Neural Network Modifying. In The International Conference on Artificial Intelligence and Logistics Engineering (pp. 3-15). Springer, Cham.

59. Бодянський, Є. В., Антоненко, Т. Є. (2019). Глибока нео-фаззі нейронна мережа та її навчання. Біоніка інтелекту, 1(92), 3-8.

60. Безсонов, О. О., Руденко, О. Г., Олійник, К., Романюк, О. (2020). Оптимізація процесу навчання штучних нейронних мереж.

61. Kochura, Y., Stirenko, S., Alienin, O., Novotarskiy, M., Gordienko, Y. (2017, September). Comparative analysis of open source frameworks for machine learning with use case in single-threaded and multi-threaded modes. In 2017 12th International Scientific and Technical Conference on

Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 373-376). IEEE.

62. Григоров, О. В., Аніщенко, Г. О., Стрижак, В. В., Петренко, Н. О., Турчин, О. В., Окунь, А. О., Пономарьов, О. Е. (2019). Штучний інтелект. Машинне навчання.

63. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

64. Neill, J. O. (2020). An overview of neural network compression. *arXiv preprint arXiv:2006.03669*.

65. Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016, October). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision* (pp. 525-542). Springer, Cham.

66. Parajuli, S., Raghavan, A., & Chai, S. (2018). Generalized Ternary Connect: End-to-End Learning and Compression of Multiplication-Free Deep Neural Networks. *arXiv preprint arXiv:1811.04985*.

67. Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*.

68. Nahshan, Y., Chmiel, B., Baskin, C., Zheltonozhskii, E., Banner, R., Bronstein, A. M., & Mendelson, A. (2021). Loss aware post-training quantization. *Machine Learning*, *110*(11), 3245-3262.

69. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., ... & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2704-2713).

70. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

71. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

72. Zhao, H., Shi, J., Qi, X., Wang, X., & Jia, J. (2017). Pyramid scene parsing network. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2881-2890).

73. LeCun, Y., Cortes, C., & Burges, C. (2010). MNIST handwritten digit database.

74. Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009, June). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (pp. 248-255). Ieee.

75. Cordts, M. (2017). Understanding Cityscapes: Efficient Urban Semantic Scene Understanding. Doctoral dissertation, TechnischeUniversität.

76. Irvin, J., Rajpurkar, P., Ko, M., Yu, Y., Ciurea-Ilcus, S., Chute, C., ... & Ng, A. Y. (2019, July). Chexpert: A large chest radiograph dataset with uncertainty labels and expert comparison. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 33, No. 01, pp. 590-597).

77. TensorFlow – end-to-end open source platform for machine learning [Електронний ресурс]. – Режим доступу: https://www.tensorflow.org/

78. ML for mobile and edge devices – TensorFlow Lite [Електронний ресурс]. – Режим доступу: https://www.tensorflow.org/lite

79. PyTorch an open source machine learning framework [Електронний ресурс]. – Режим доступу: https://pytorch.org/

80. Caffe2 a new lightweight, modular, and scalable deep learning framework [Електронний ресурс]. – Режим доступу: https://caffe2.ai/

81. Apache MXNet a flexible and efficient library for deep learning [Електронний ресурс]. – Режим доступу: https://mxnet.apache.org

82. Intel® Movidius™ Neural Compute SDK [Електронний ресурс]. – Режим доступу: https://movidius.github.io/ncsdk/tools/tools_overview.html

83. Intel Distribution of OpenVINO Toolkit [Електронний ресурс]. – Режим доступу: https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html

84. NVIDIA TensorRT [Електронний ресурс]. – Режим доступу: https://developer.nvidia.com/tensorrt

85. Zen Deep Neural Network Library [Електронний ресурс]. – Режим доступу: https://developer.amd.com/zendnn/

86. AMD ROCm [Електронний ресурс]. – Режим доступу: https://www.amd.com/en/graphics/servers-solutions-rocm

87. Coral Edge TPU Compiler [Електронний ресурс]. – Режим доступу: https://coral.ai/docs/edgetpu/compiler/

88. Horned Sungem SDK [Електронний ресурс]. – Режим доступу: https://github.com/HornedSungem/SungemSDK

89. Huawei HiAI SDK [Електронний ресурс]. – Режим доступу: https://developer.huawei.com/consumer/en/hiai/

90. Pruning in Keras example [Електронний ресурс]. – Режим доступу: https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras

91. Oyama, Y., Nomura, A., Sato, I., Nishimura, H., Tamatsu, Y., & Matsuoka, S. (2016, December). Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)* (pp. 66-75). IEEE.

92. Viebke, A., Memeti, S., Pllana, S., & Abraham, A. (2019). CHAOS: a parallelization scheme for training convolutional neural networks on Intel Xeon Phi. *The Journal of Supercomputing*, *75*(1), 197-227.

93. Yan, F., Ruwase, O., He, Y., & Chilimbi, T. (2015, August). Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1355-1364).

94. Qi, H., Sparks, E. R., & Talwalkar, A. (2016). Paleo: A performance model for deep neural networks.

95. Demmel, J., & Dinh, G. (2018). Communication-optimal convolutional neural nets. *arXiv preprint arXiv:1802.06905*.

96. Chen, J., & Ran, X. (2019). Deep learning with edge computing: A review. *Proceedings of the IEEE*, *107*(8), 1655-1674.

97. Varghese, B., Wang, N., Bermbach, D., Hong, C. H., de Lara, E., Shi, W., & Stewart, C. (2020). A survey on edge benchmarking. *arXiv preprint arXiv:2004.11725*.

98. He, Y., Zhang, X., & Sun, J. (2017). Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision* (pp. 1389-1397).

99. Hu, H., Peng, R., Tai, Y. W., & Tang, C. K. (2016). Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

100. Kruglov, A. (2018). Channel-wise pruning of neural networks with tapering resource constraint. *arXiv preprint arXiv:1812.07060*.

101. Molchanov, P., Tyree, S., Karras, T., Aila, T., & Kautz, J. (2016). Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*.

102. Wang, H., Zhang, Q., Wang, Y., & Hu, H. (2017). Structured probabilistic pruning for convolutional neural network acceleration. *arXiv preprint arXiv:1709.06994*.

103. Zhu, M., & Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.

104. Bolukbasi, T., Wang, J., Dekel, O., & Saligrama, V. (2017, July). Adaptive neural networks for efficient inference. In *International Conference on Machine Learning* (pp. 527-536). PMLR.

105. Gao, X., Zhao, Y., Dudziak, Ł., Mullins, R., & Xu, C. Z. (2018). Dynamic channel pruning: Feature boosting and suppression. *arXiv preprint arXiv:1810.05331*.

106. Nan, F., & Saligrama, V. (2017). Adaptive classification for prediction under a budget. *Advances in neural information processing systems*, *30*.

107. Wang, X., Yu, F., Dou, Z. Y., Darrell, T., & Gonzalez, J. E. (2018). Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 409-424).

108. Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

109. Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

110. Smith, S. L., Kindermans, P. J., Ying, C., & Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

111. You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.

112. Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., ... & Jue, J. P. (2019). All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, *98*, 289-330.

113. Puliafito, C., Mingozzi, E., Longo, F., Puliafito, A., & Rana, O. (2019). Fog computing for the internet of things: A survey. *ACM Transactions on Internet Technology (TOIT)*, *19*(2), 1-41.

114. Zhang, W., Li, S., Liu, L., Jia, Z., Zhang, Y., & Raychaudhuri, D. (2019, April). Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications* (pp. 1270-1278). IEEE.

115. Puliafito, C., Mingozzi, E., Vallati, C., Longo, F., & Merlino, G. (2018, June). Virtualization and migration at the network edge: An overview. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)* (pp. 368-374). IEEE.

116. NVIDIA A100 Deep Learning Benchmarks for TensorFlow [Електронний ресурс]. – Режим доступу: https://www.exxactcorp.com/blog/Benchmarks/nvidia-a100-deep-learning-benchmarks-for-tensorflow

117. NVIDIA A100 TENSOR CORE GPU [Електронний ресурс]. – Режим доступу: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf

118. Відеокарта PNY Tesla A100 TCSA100M-PB [Електронний ресурс]. – Режим доступу: https://ek.ua/ua/PNY-TESLA-A100-TCSA100M-PB.htm

ДОДАТОК А

Частина програмного коду

Лістинг А.1 — Модуль coral_edgetpu_main.py

```
from gui.coral_edgetpu_ui import *
from gui.coral_edgetpu_model import *


def close():
    exit(0)


def main():

    main_ui = MainUI()
    main_model = MainModel()

    main_model.register_ui(main_ui)
    main_ui.register_model(main_model)

    # main_ui.create_main_window('medic.ico', 'EdgeTPU X-ray Lung Abnormality Classification Demo')
    main_ui.create_main_window('medic.ico', 'EdgeTPU X-ray Демо Класифікації Аномалей Легеней')
    main_model.wait_for_flash_drive()

    main_ui.root.protocol("WM_DELETE_WINDOW", close)
    main_ui.root.mainloop()
```

```
if __name__ == "__main__":
    main()
```

Лістинг А.2 — Модуль coral_edgetpu_utils.py

```
import platform
import os.path
import tflite_runtime.interpreter as tflite

import numpy as np
import time

from PIL import Image

MODEL_DRIVE_LETTERS = 'ABDEFGHIJKLMNOPQRSTUVWXYZ'

TFLITE_MODEL = '/chest_resnet50_grayscale_fp32.tflite'
EDGETPU_MODEL = '/chest_resnet50_grayscale_uint8_edgetpu.tflite'

IMAGE_DIR = ['D:/representative-train/abnormal',
'D:/representative-train/normal']
NUM_CLASSES = len(IMAGE_DIR)

EDGETPU_SHARED_LIB = {
  'Linux': 'libedgetpu.so.1',
  'Darwin': 'libedgetpu.1.dylib',
  'Windows': 'edgetpu.dll'
}[platform.system()]


def check_flash_drive():
```

```
    drives = ['%s:' % d for d in MODEL_DRIVE_LETTERS if os.path.exists('%s:'
% d)]
    for d in drives:
        if os.path.exists(os.path.join(d, TFLITE_MODEL)):
            return d
    return None


def make_tpu_interpreter(flash_drive_model):
    model_file, *device = flash_drive_model.split('@')
    try:
        interpreter = tflite.Interpreter(
            model_path=model_file,
            experimental_delegates=[
                tflite.load_delegate(EDGETPU_SHARED_LIB,
                                     {'device': device[0]} if device else {})
            ])

        return interpreter, 0

    except ValueError:
        return None, None


def make_cpu_interpreter(flash_drive_model):
    try:
        interpreter = tflite.Interpreter(flash_drive_model)
        return interpreter, 1

    except ValueError:
```

```
        return None, None


def load_images(image_dir, dtype=None):

    image_array = []

    for i in os.listdir(image_dir):
        image_path = os.path.join(image_dir, i)
        image = Image.open(image_path)
        image = image.resize((128, 128))
        # convert image to numpy array
        x = np.asarray(image)
        x = np.expand_dims(x, axis=(0, -1))
        if dtype:
            x = x.astype(dtype)
        image_array.append((image_path, x))

    return image_array


def image_generator(subset_len, dtype=None):

    dataset = []
    index = []
    for dir in IMAGE_DIR:
        dataset.append(load_images(dir, dtype))
        index.append(0)

    for _ in range(subset_len):
```

```
        for k in range(NUM_CLASSES):
            data = (dataset[k][index[k]], k)

            index[k] += 1
            index[k] = index[k] % len(dataset[k])

            yield data


def run_inference(interpreter, operation_mode, images, ui, show_image=False):

    if operation_mode == 0:
        # dtype = None
        num_images = 50
    else:
        # dtype = 'float32'
        num_images = 50

    t1 = time.perf_counter()

    interpreter.allocate_tensors()

    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()

    fps_sum = 0
    fps_min, fps_max = float('+inf'), float('-inf')
    counter, avg_counter = 0, 0
    tp = 0
    label_counter = [0, 0]
```

```
for i in images:
    counter += 1

    interpreter.set_tensor(input_details[0]['index'], i[0][1])

    tinf1 = time.perf_counter()
    interpreter.invoke()
    tinf2 = time.perf_counter()

    inference_time = (tinf2 - tinf1) * 1000
    elapsed_time1 = (tinf2 - t1) * 1000
    framerate = 1000 / inference_time

    prediction = interpreter.get_tensor(output_details[0]['index'])[0]
    prediction_label = np.argmax(prediction)
    prediction_percentage = prediction[prediction_label]
    if operation_mode == 0:
        prediction_percentage /= 255

    if prediction_label == i[1]:
        tp += 1
    label_counter[prediction_label] += 1

    if counter > 2:

        avg_counter += 1

        if framerate < fps_min:
            fps_min = framerate
```

```
            if framerate > fps_max:
                fps_max = framerate

            fps_sum += framerate

            if not show_image:
                ui.set_fps("min / avg / max\n{:.2f} / {:.2f} / {:.2f}".format(fps_min, fps_sum / avg_counter, fps_max))

        if not show_image:
            ui.set_procesing_frame((counter, label_counter[0], label_counter[1]))
        else:
            ui.update_demo_window(i[0][0], prediction_label, prediction_percentage)
        # print('FPS:', framerate)
        print("{:.2f},{:.2f},{:.2f}".format(framerate, inference_time, elapsed_time1))
        # print(i[0][0])

        # get initialization overheads
        t1 = time.perf_counter()

    print('Accuracy', tp / (num_images * NUM_CLASSES))

def adaptive_pruning(model, pruning_iterations, prune_layer_name):
    for i in range(0, pruning_iterations):
        for pln in prune_layer_name:
            # model pruning
            layer = model.get_layer(name=pln)
            apoz = identify.get_apoz(model, layer, x_val)
            high_apoz_channels = identify.high_apoz(apoz, method="std",
                                                    cutoff_std=0.1)
```

```
        model = delete_channels(model, layer, high_apoz_channels)
        model.compile(optimizer='adam',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        # get total params after pruning
        params_after_prining=np.sum([K.count_params(p) for p in
                                     set(model.trainable_weights)])
        prune_level.append(1-(params_after_prining/original_model_params))
        # get pruning level
        print("Pruning level: "+str(1-(params_after_prining
                                       /original_model_params)))
        # test pruned model "before training"
        test_loss, test_accuracy = model.evaluate(x_val,
                                  y_val,
                                  batch_size=batch_size,
                                  verbose=2)
        # re-traing pruned model
        results = model.fit(chest_dataset,
                            epochs=num_epochs,
                            batch_size=batch_size,
                            verbose=training_verbosity,
                            validation_data=(x_val, y_val),
                            callbacks=[early_stopping, reduce_lr])
        # test pruned model "after re-training"
        test_loss, test_accuracy = model.evaluate(x_val,
                                  y_val,
                                  batch_size=batch_size,
                                  verbose=2)
    return model
```

```
def model_training():
    create ResNet 50 with custom input shape

                resnet50 = ResNet50(weights='imagenet', include_top=False,
input_tensor=Input(shape=(image_size, image_size, 3)))
                        # resnet50 = ResNet50(include_top=False,
input_tensor=Input(shape=(image_size, image_size, 3)))

    # Chest MNIST classifier
    classifier = resnet50.output
    classifier = GlobalAveragePooling2D()(classifier)
    ###
    # classifier = MaxPooling2D(pool_size=(1,1))(classifier)
    classifier = Flatten()(classifier)
    # classifier = Dropout(0.5)(classifier)
    # classifier = Dense(128, activation='relu')(classifier)
    ###
    classifier = Dense(num_classes, activation='softmax')(classifier)

    # freeze ResNet
    resnet50.trainable = False

    # stack to original ResNet 50
    chest_resnet = Model(inputs=resnet50.input, outputs=classifier)

    # train model
    checkpoint = ModelCheckpoint("resnet50_imagenet_grayscale_best",
                        monitor='loss', verbose=1, save_best_only=True,
                        mode='auto', save_freq=100)
    opt = tf.keras.optimizers.Adam(learning_rate=0.01)
```

```
    chest_resnet.compile(optimizer=opt, loss='categorical_crossentropy',
                         metrics=["accuracy"])
    chest_resnet.fit(chest_dataset, batch_size=batch_size, epochs=num_epochs,
                     callbacks=[checkpoint])

def quantize_model():



    converter = tf.lite.TFLiteConverter.from_keras_model(chest_resnet)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    converter.representative_dataset = representative_dataset



                                        converter.target_spec.supported_ops =
[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
    converter.inference_input_type = tf.uint8  # or tf.int8
    converter.inference_output_type = tf.uint8  # or tf.int8
    tflite_model = converter.convert()

    # save tflite model
    with open(project_drive+'chest_resnet50_grayscale_uint8.tflite', 'wb') as f:
        f.write(tflite_model)

def representative_dataset():
    BATCH_SIZE = 1
    num_classes = 2

    train_path = "D:/representative-train"
```

```
    chest_dataset = tf.keras.preprocessing.image_dataset_from_directory(train_path, batch_size=BATCH_SIZE, image_size=(image_size, image_size), color_mode='grayscale', label_mode='categorical')
    chest_dataset = chest_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

    for i in chest_dataset:
        yield [i[0]]
```

Лістинг А.3 — Модуль coral_edgetpu_model.py

```
from concurrent.futures import ThreadPoolExecutor
import threading
import os
import random

import gui.coral_edgetpu_utils as utils


class MainModel:

    ui = None

    update_sec = 1.0

    auto_update = None
    executor = ThreadPoolExecutor(max_workers=1)

    flash_drive = None

    cpu_interpreter = None
```

```
tpu_interpreter = None

cpu_image_data = None
tpu_image_data = None

def register_ui(self, ui):
    self.ui = ui

def wait_for_flash_drive(self):
    self.executor.submit(self.load_model)

def wait_for_tpu(self):

    # load tpu interpreter
    interpreter, operation_mode = utils.make_tpu_interpreter(
        os.path.join(self.flash_drive, utils.EDGETPU_MODEL))

    if interpreter is not None:
        self.tpu_interpreter = interpreter
        self.ui.set_accelerator_state(2)
        self.ui.set_fps('')
        self.ui.set_procesing_frame((', ', '), text=[', ', '])
    else:
        self.ui.set_accelerator_state(1)
        threading.Timer(self.update_sec, self.wait_for_tpu).start()

def load_model(self):
    flash_drive = utils.check_flash_drive()

    if flash_drive is not None:
```

```
        self.flash_drive = flash_drive

        # load image data
        self.tpu_image_data = list(utils.image_generator(50))
        self.cpu_image_data = list(utils.image_generator(50, dtype='float32'))

        # load cpu interpreter
        interpreter, operation_mode = utils.make_cpu_interpreter(
            os.path.join(self.flash_drive, utils.TFLITE_MODEL))
        if interpreter is not None:

            self.cpu_interpreter = interpreter

            self.ui.set_flash_drive_connected()

            # wait for tpu connection in background
            self.executor.submit(self.wait_for_tpu)

    else:
        self.ui.set_flash_drive_disconnected()
        self.ui.set_accelerator_state(3)
        self.ui.set_fps('')
        self.ui.set_procesing_frame((', ', '), text=[', ', '])

        threading.Timer(self.update_sec, self.load_model).start()

def button_action1(self):
    self.executor.submit(self.run_demo)

def button_action2(self):
```

```
        self.executor.submit(self.run_image_demo)

    def run_demo(self):

        # self.view.set_accelerator_state(0)
        self.ui.set_fps('')
        self.ui.set_procesing_frame(('', '', ''), text=['', '', ''])

        # try tpu first
        if self.tpu_interpreter is not None:
            # self.view.set_accelerator_state(2)
            self.ui.set_button_state('disabled')
            utils.run_inference(self.tpu_interpreter, 0, self.tpu_image_data, self.ui)
            self.ui.set_button_state('normal')

            return

        # fallback to cpu
        # self.view.set_accelerator_state(1)
        if self.cpu_interpreter is not None:
            self.ui.set_button_state('disabled')
            utils.run_inference(self.cpu_interpreter, 1, self.cpu_image_data, self.ui)
            self.ui.set_button_state('normal')

    def run_image_demo(self):

        random_image = random.randint(0, len(self.cpu_image_data)-5)

        # try tpu first
        if self.tpu_interpreter is not None:
```

```
            self.ui.set_button_state('disabled')
                                                utils.run_inference(self.tpu_interpreter,     0,
self.tpu_image_data[random_image:random_image+1], self.ui, show_image=True)
            self.ui.set_button_state('normal')

            return

        # fallback to cpu
        if self.cpu_interpreter is not None:
            self.ui.set_button_state('disabled')
                                                utils.run_inference(self.cpu_interpreter,     1,
self.cpu_image_data[random_image:random_image+1],                    self.ui,
show_image=True)
            self.ui.set_button_state('normal')
```

Лістинг А.4 — Модуль coral_edgetpu_ui.py

```
import tkinter as tk
from PIL import Image, ImageTk


class MainUI:
    but1 = None
    but2 = None
    but3 = None
    label1 = None
    label2 = None
    label310 = None
    label311 = None
    label320 = None
    label321 = None
```

```
label330 = None
label331 = None
label4 = None
root = None
frame1 = None
frame2 = None
frame3 = None
frame4 = None
main_win_name = None

model = None

auto = False

window_icon = None

main_window_x = None
main_window_y = None

demo_window = None
demo_file = None
demo_image = None
demo_description = None

# flash_drive_state = ['Waiting for flash drive with model ...',
#                      'ResNet50 model connected!']

flash_drive_state = ['Очікую флеш-накопичувач із моделлю ...',
                     'Модель ResNet50 підключено!']
```

```
# accelerator_state = ['Waiting for EdgeTPU ...',
#                      'EdgeTPU not connected. Running on CPU.',
#                      'Coral EdgeTPU connected.',
#                      '']

accelerator_state = ['Очікую прискорювач EdgeTPU ...',
                     'EdgeTPU не підключено. Виконання на CPU.',
                     'Coral EdgeTPU підключено.',
                     '']

# prediction_description = ['Abnormality detected\n{:.2f}',
#                           'No abnoramlities detected\n{:.2f}']

prediction_description = ['Виявлено аномалію легеней\n{:.2f}',
                          'Не виявлено аномалій легеней\n{:.2f}']

# processing_description = ['Total', 'Normal', 'Abnormal']

processing_description = ['Всього', 'Без аномалій', 'Із аномаліями']

def register_model(self, model):
    self.model = model

def but1_command(self):
    self.model.button_action1()

def but2_command(self):
    self.model.button_action2()
```

```
def update_demo_window(self, image_path, prediction_label, prediction_percentage):
    if self.demo_window is None:
        self.create_demo_window(image_path, self.prediction_description[prediction_label].format(prediction_percentage))
    else:
        self.update_demo_image(image_path, self.prediction_description[prediction_label].format(prediction_percentage))

def close_demo_window(self):
    self.demo_window.destroy()
    self.demo_window = None

def create_main_window(self, win_ico, win_name):
    win_width = 600
    win_hight = 400
    label_width = 50
    button_width = 10
    lab_padx = (win_width-(label_width+button_width))//2
    self.main_win_name = win_name
    root = tk.Tk()
    root.resizable(0, 0)
    box_value = tk.StringVar()
    self.window_icon = win_ico
    root.iconbitmap(self.window_icon)
    root.title(win_name)
    w = root.winfo_screenwidth()
    h = root.winfo_screenheight()
    w = w//2
    h = h//2
```

```
self.main_window_x = w - win_width//2
self.main_window_y  = h - win_hight//2
root.geometry('{}x{}+{}+{}'.format(win_width, win_hight, self.main_window_x, self.main_window_y))
# root.minsize(width=600, height=100)
root.state("normal")
self.root = root
frame1 = tk.Frame()
self.frame1 = tk.LabelFrame(frame1, width=win_width, height=win_hight//4, text='Deep Neural Network', font=("Times New Roman", 12, "bold"), labelanchor="n")
self.frame2 = tk.LabelFrame(frame1, width=win_width, height=win_hight//4, text='Статус DNN прискорювача', font=("Times New Roman", 12, "bold"), labelanchor="n")
self.frame3 = tk.LabelFrame(frame1, width=win_width, height=win_hight//4, text="Обробка рентген знімків", font=("Times New Roman", 12, "bold"), labelanchor="n")
self.frame4 = tk.LabelFrame(frame1, width=win_width, height=win_hight//4, text="FPS", font=("Times New Roman", 12, "bold"), labelanchor="n")
self.label1 = tk.Label(self.frame1, text=self.flash_drive_state[0], width=label_width-5)
self.label2 = tk.Label(self.frame2, text='', width=label_width)

subframe0 = tk.Frame(self.frame3)
subframe1 = tk.Frame(subframe0)
subframe2 = tk.Frame(subframe0)
subframe3 = tk.Frame(subframe0)

self.label310 = tk.Label(subframe1, text='', width=15)
self.label311 = tk.Label(subframe1, text='', width=15)
```

```
self.label320 = tk.Label(subframe2, text='', width=15)
self.label321 = tk.Label(subframe2, text='', width=15)
self.label330 = tk.Label(subframe3, text='', width=15)
self.label331 = tk.Label(subframe3, text='', width=15)


self.label4 = tk.Label(self.frame4, text='', width=label_width)
self.but1 = tk.Button(self.frame1, text="Запуск", width=button_width)
self.but2 = tk.Button(self.frame1, text="Результат", width=button_width)
# self.but3 = tk.Button(self.frame3, text="But3", width=button_width)
self.label1.pack(side='left', pady=5)
self.but1.pack(side='left', padx=5, pady=5)
self.label2.pack(pady=5)
self.but2.pack(side='left', padx=5, pady=5)
self.label310.pack(pady=5)
self.label311.pack(pady=5)
self.label320.pack(pady=5)
self.label321.pack(pady=5)
self.label330.pack(pady=5)
self.label331.pack(pady=5)
# self.but3.pack(side='left', padx=10, pady=5)
self.label4.pack(pady=15)
self.but1['command'] = self.but1_command
self.but1['state'] = 'disabled'
self.but2['state'] = 'disabled'
self.but2['command'] = self.but2_command
frame1.pack()
self.frame1.pack_propagate(False)
self.frame2.pack_propagate(False)
self.frame3.pack_propagate(False)
```

```
        self.frame4.pack_propagate(False)
        self.frame1.pack(padx=2)
        self.frame2.pack(padx=2)
        subframe0.pack()
        subframe1.pack(side='left')
        subframe2.pack(side='left')
        subframe3.pack(side='left')
        self.frame3.pack(padx=2)
        self.frame4.pack(padx=2)

    def create_demo_window(self, image_path, text_description):
        demo_window = tk.Toplevel(self.root)
        demo_window.resizable(0, 0)
        demo_window.title("Результат")
        demo_window.iconbitmap(self.window_icon)
                                                demo_window.geometry("300x400+{}+
{}".format(self.main_window_x+600, self.main_window_y))

            label_frame1 = tk.LabelFrame(demo_window, width=300, height=400,
text='X-ray {}'.format(image_path[image_path.rfind('\\')+1:-4]), font=("Times
New Roman", 12, "bold"), labelanchor="n")

        image_file = Image.open(image_path)
        image_file = image_file.resize((300, 300))
        image_file1 = ImageTk.PhotoImage(image_file)
        # image = canvas.create_image(0, 0, anchor='nw', image=image_file1)
                    image_label = tk.Label(label_frame1, height=300, width=300,
image=image_file1)
        image_label.image = image_file1
        image_label.pack()
```

```
        description_label = tk.Label(label_frame1, text=text_description, width=250)
        description_label.pack(pady=10)

        label_frame1.pack_propagate(False)
        label_frame1.pack(padx=2)

                                demo_window.protocol("WM_DELETE_WINDOW",
self.close_demo_window)
        self.demo_window = demo_window
        self.demo_file = label_frame1
        self.demo_image = image_label
        self.demo_description = description_label

    def update_demo_image(self, image_path, text_description):
            self.demo_file['text'] = 'X-ray {}'.format(image_path[image_path.rfind('\\')
+1:-4])
        image_file = Image.open(image_path)
        image_file = image_file.resize((300, 300))
        image_file1 = ImageTk.PhotoImage(image_file)
        self.demo_image['image'] = image_file1
        self.demo_image.image = image_file1
        self.demo_description['text'] = text_description

    def set_flash_drive_connected(self):
        self.label1['text'] = self.flash_drive_state[1]
        self.but1['state'] = 'normal'
        self.but2['state'] = 'normal'

    def set_flash_drive_disconnected(self):
```

```
    self.label1['text'] = self.flash_drive_state[0]
    self.but1['state'] = 'disabled'
    self.but2['state'] = 'disabled'

def set_button_state(self, state):
    self.but1['state'] = state
    self.but2['state'] = state

def set_accelerator_state(self, label_index):
    self.label2['text'] = self.accelerator_state[label_index]

def set_procesing_frame(self, data, text=None):

    if text is None:
        text = self.processing_description

    self.label310['text'], self.label311['text'] = text[0], data[0]
    self.label320['text'], self.label321['text'] = text[1], data[2]
    self.label330['text'], self.label331['text'] = text[2], data[1]

def set_fps(self, data):
    self.label4['text'] = data
```

ДОДАТОК Б

Список публікацій здобувача

1. Taran, V., Gordienko, Y., Rokovyi, O., Alienin, O., Kochura, Y., Stirenko, S. (2022). Edge Intelligence for Medical Applications under Field Conditions. Lecture Notes on Data Engineering and Communications Technologies, 135, 71-80. Springer. ISSN 23674520. 10.1007/978-3-031-04809-8_6 (Scopus).

2. Alienin, O., Rokovyi, O., Gordienko, Y., Kochura, Y., Taran, V., Stirenko, S. (2022). Artificial Intelligence Platform for Distant Computer-Aided Detection (CADe) and Computer-Aided Diagnosis (CADx) of Human Diseases. Lecture Notes on Data Engineering and Communications Technologies, 135, 91-100. Springer. ISSN 23674512. DOI 10.1007/978-3-031-04809-8_8 (Scopus).

3. Gordienko, Y., Kochura, Y., Taran, V., Gordienko, N., Rokovyi, O., Alienin, O., Stirenko, S. (2021). “Last Mile” Optimization of Edge Computing Ecosystem with Deep Learning Models and Specialized Tensor Processing Architectures. Advances in Computers, 122, 303-341. Academic Press Inc. ISSN 00652458. DOI 10.1016/bs.adcom.2020.10.003 (Scopus).

4. Gordienko, Y., Kochura, Y., Taran, V., Gordienko, N., Rokovyi, O., Alienin, O., Stirenko, S. (2020). Scaling Analysis of Specialized Tensor Processing Architectures for Deep Learning Models. Studies in Computational Intelligence, 866, 65-99. Springer Verlag. ISSN 1860949X. DOI 10.1007/978-3-030-31756-0_3 (Scopus) Q1.

5. Gordienko, Y., Kochura, Y., Taran, V., Gordienko, N., Bugaiov, A., Stirenko, S. (2019). Adaptive Iterative Pruning for Accelerating Deep Neural Networks. Procedings of XIth International Scientific and Practical Conference on Electronics and Information Technologies, 8892346, 173-178. IEEE. ISBN 9781728135618. DOI 10.1109/ELIT.2019.8892346 (Scopus).

6. Taran, V., Gordienko, Y., Rokovyi, A., Alienin, O., Stirenko, S. (2020). Impact of Ground Truth Annotation Quality on Performance of Semantic Image Segmentation of Traffic Conditions. Advances in Intelligent Systems and Computing, 938, 183-193. Springer Verlag. ISSN 21945357. DOI 10.1007/978-3-030-16621-2_17 (Scopus) Q3.

7. Kochura, Y.; Gordienko, Y.; Taran, V.; Gordienko, N.; Rokovyi, A.; Alienin, O.; Stirenko, S. (2020). Batch Size Influence on Performance of Graphic and Tensor Processing Units During Training and Inference Phases. Advances in Intelligent Systems and Computing, 938, 658-668. Springer Verlag. ISSN 21945357. DOI 10.1007/978-3-030-16621-2_61 (Scopus) Q3.

8. Taran, V., Gordienko, Y., Gordienko, N., Rokovyi, A., Kochura, Y., Alienin, O., Stirenko, S. (2018). Performance evaluation of deep learning networks for semantic segmentation of traffic stereo-pair images. ACM International Conference Proceeding Series, 73-80. Association for Computing Machinery. ISBN 9781450364256, DOI 10.1145/3274005.3274032 (Scopus).