	Jetson AGX Orin series			NVIDIA RTX 30 Series	Nvidia's RTX 5000-series	Jetson Orin Nano series		Raspberry Pi
	Jetson AGX Orin Developer Kit	**Jetson AGX Orin 64GB**	**Jetson AGX Orin 32GB**	**Nvidia RTX 3090**	**Nvidia RTX A5000**	**Jetson Orin Nano Developer Kit**	**Jetson Orin Nano 8GB**	**Raspberry Pi 5**
Networking*	RJ45 connector with up to 10 GbE	1x GbE 1x 10GbE		RJ45 connector with up to 10 GbE	1x GbE, 1x 10 GbE	1xGbE Connector	1x GbE	1x GbE
Display	1x DisplayPort 1.4a (+MST) connector	1x 8K60 multi-mode DP 1.4a (+MST)/eDP 1.4a/HDMI 2.1		1x HDMI 2.1, 1x DP 1.4a, 1x eDP 1.4	1x 4K30 multi-mode DP 1.2 (+MST)/eDP 1.4/HDMI 1.4a	1x DisplayPort 1.2 (+MST) connector	1x 4K30 multi-mode DP 1.2 (+MST)/eDP 1.4/HDMI 1.4**	1x DisplayPort 1.2 (+MST) connector
Other I/O	40-pin header (UART, SPI, I2S, I2C, CAN, PWM, DMIC, GPIO) 12-pin automation header 10-pin audio panel header 10-pin JTAG header 4-pin fan header 2-pin RTC batter backup connector microSD slot DC power jack Power, Force Recovery, and Reset buttons	4x UART, 3x SPI, 4x I2S, 8x I2C, 2x CAN, PWM, DMIC & DSPK, GPIOs		4x UART, 3x SPI, 4x I2S, 8x I2C, 2x CAN, PWM, DMIC & DSPK, GPIOs	4x UART, 3x SPI, 4x I2S, 8x I2C, 2x CAN, PWM, DMIC & DSPK, GPIOs	40-Pin Expansion Header(UART, SPI, I2S, I2C, GPIO) 12-pin button header 4-pin fan header microSD Slot DC power jack	3x UART, 2x SPI, 2x I2S, 4x I2C, 1x CAN, DMIC & DSPK, PWM, GPIOs	
Power	15W - 60W		15W - 40W	350W	230 W	7W - 15W		5V/3A
Mechanical	110mm x 110mm x 71.65mm (Height includes feet, carrier board, module, and thermal solution)	100mm x 87mm 699-pin Molex Mirror Mezz Connector Integrated Thermal Transfer Plate		313 mm x 138 mm x 55 mm	267 mm x 112 mm x 40 mm (Height includes feet, carrier board, module, and thermal solution)	100 mm x 79 mmx 21 mm (Height includes feet, carrier board, module, and thermal solution)	69.6mm x 45mm 260-pin SO-DIMM connector	85 mm x 56 mm

Summer Research Fellowship Programme, 2024

Final Report

Optimization of Training Time and Generalizability for DNN Training on Edge Devices.

Kedar Dhule
ENGS2190
National Institute of Technology, Karnataka

Guide: Prof. Yogesh Simmhan
Indian Institute of Science, Banglore

Table of Contents **Page No.**

Abstract

This project explores the optimization of training time and power consumption for deep neural network (DNN) workloads on Nvidia Jetson edge devices. A unique feature of these compact devices is their fine-grained control over CPU, GPU, memory frequencies, and active CPU cores, allowing for efficient power management within constrained settings. We used a method called multi-dimensional binary search to identify optimal power modes that minimise training time while adhering to specified power budgets. Additionally, we conducted performance comparisons across various hardware devices, highlighting differences in compute capabilities and efficiency. Results demonstrate the significant impact of hardware specifications on DNN training performance, with insights into the effectiveness of different power modes and configurations. For instance, while Nvidia RTX 3090 excels in computational speed due to its higher CUDA core count. The study further extends to evaluating the generalizability of training models across different Jetson generations, including the latest Jetson Orin Nano and found our methods to be effective across different Jetson generations. This project contributes to advancing DNN training methodologies on edge devices.

CHAPTER-1 INTRODUCTION

My project focuses on optimising the training time and power consumption for deep neural network (DNN) workloads (model + dataset), on Nvidia Jetson edge devices. This project is an extension of a previous study published in Sigmetrics where we examined DNN training on Jetson devices. In that study, we analysed how various factors, such as I/O pipelining, parallelism, storage media, mini-batch sizes, and power modes, affect CPU and GPU utilisation, fetch stalls, training time, and energy usage. Using these results, we built a simple model to predict the training time and energy consumption per epoch for any given DNN across different power modes .

We use Accelerated edge devices like NVIDIA's Jetson series and five popular DNN workloads (ResNet, MobileNet, YOLO, BERT, LSTM) for training and inference . Jetson devices have unique characteristics such as low power usage, a slower RAM shared between GPU and CPU, support for diverse storage media such as SD cards, eMMC and NVME SSDs . A unique feature of Nvidia's Jetson edge devices is their fine-grained control over CPU, GPU and memory frequencies and active CPU cores, enabled by dynamically setting their power modes .

At the beginning of the project, I learned to execute training of DNN workloads under various power modes (adjusting CPU/GPU/memory frequencies and core counts) using shell scripts and PyTorch on a remote machine accessed via SSH in Visual Studio Code. During the runs, I became familiar with Tmux (terminal multiplexer), a command-line tool that allows you to manage multiple terminal sessions within a single window or remote terminal session. With Tmux, I effectively operated two Orin AGX devices, Orin3 and Orin4, simultaneously, running different workloads on each.

Additionally, we have worked on a minor revision of our previous project, PowerTrain, which was submitted to Elsevier's Future Generation Computing Systems (FGCS) . In this minor revision, the editors and reviewers provided comments and suggestions for improving the manuscript. We have addressed all the comments in this revision, incorporating feedback and corrections into the paper . Our work involved incorporating feedback and corrections from the reviews into the paper and submitting detailed responses to address the reviewers' comments .

CHAPTER - 2 OBJECTIVE

2.1) Optimal Power Mode Identification

- Training needs to maximise throughput and Inference needs to minimise latency to ensure fast responses . We develop a Multi-dimensional Binary Search (BS) strategy that identifies the best power mode and inference batch size configuration to meet user defined power and inference latency constraints and maximise the training throughput while reducing the number of power modes that are experimentally profiled.
- The goal of our optimization is to identify the optimal power mode for the DNN training and inference workloads that minimises the overall time while staying within a specified power budget .

2.2) PowerTrain Generalizability

- We aim to evaluate the generalizability of PowerTrain by transferring a reference workload trained on Jetson devices of one generation to workloads running on different generations.But reviewers questioned the extent to which the model can be transferred .
- So our objective is to extend our evaluation of PowerTrain to yet another less-powerful Jetson device in the latest Jetson generation, Jetson Orin Nano , compared to Jetson AGX Orin and Jetson Xavier AGX we evaluated in the initial submission.
- Reviewer also commented as to what if the difference between the user device and Orin AGX is even greater. Specifically, they asked whether a pretrained model on the Orin AGX could be applied to a more powerful GPU (e.g., A100) or a less powerful device (e.g., Raspberry Pi).
- Therefore, our next objective is to contrast compute performance of various edge and server devices against the Nvidia Jetson Orin device used in our experiments .

CHAPTER - 3 Experimental Setup

3.1) Hardware Devices Setup

We perform our experiments on 3 different device types and contrast their compute performance against the Nvidia Jetson Orin device used in our experiments.

- NVIDIA RTX 3090 :

It is a high-end professional graphics card used for building a GPU server.Its 10496 CUDA cores provide significant parallel processing power, crucial for accelerating deep learning computations as well as its 24GB of GDDR6X memory is useful to train larger and more complex models that require a lot of data storage during processing. One of the key advantages of the RTX 3090 cards is that it can handle large batch sizes during training. This is very beneficial in case training complex models as it reduces training time

- NVIDIA RTX A5000 :

It is a high-end professional graphics card designed for advanced GPU workstations. It's built on the latest NVIDIA Ampere architecture. It is built on the latest NVIDIA Ampere architecture. The A5000 has 64 Streaming Multiprocessors (SMs), each containing 128 CUDA cores totaling 8,192 CUDA cores. It has 512 GB CPU and 24GB of GDDR6 (GPU) memory.Also it has 256 Tensor cores specifically designed to accelerate deep learning which is useful for GPU to process large volumes of data in parallel and significantly speed up training tasks.

- Raspberry Pi 5 :

The Raspberry Pi 5 is the latest iteration of the popular single-board computer series developed by the Raspberry Pi Foundation. It is the latest generation (non-accelerated) edge device. It has ARM Cortex-A76 64-bit CPU and 8 GB LPDDR4X RAM .It is powered via a USB Type-C port, which requires a charger that can output 5 volts and 5 amps. I have used a 128 GB micro SD card ,USB card reader and installed the Raspberry Pi (64-bit) operating system. It has Gigabit Ethernet and Wi-Fi 6 (802.11ax) for faster wireless connectivity .

So for running DNN workload on the new raspberry pi5 , I need to flash the SD card with the OS. I have downloaded and installed the official Raspberry Pi Imager. I have used a 64 GB microSD card ,a card reader and installed the Raspberry Pi (64 bit) operating system using RPi Imager on Windows . After writing the Raspberry Pi OS to SD card ,I booted Rpi by inserting SD card into it and connecting Rpi to monitor ,keyboard,mouse for accessability ,Ethernet cable for internet and Plug the Rpi in to power it on .

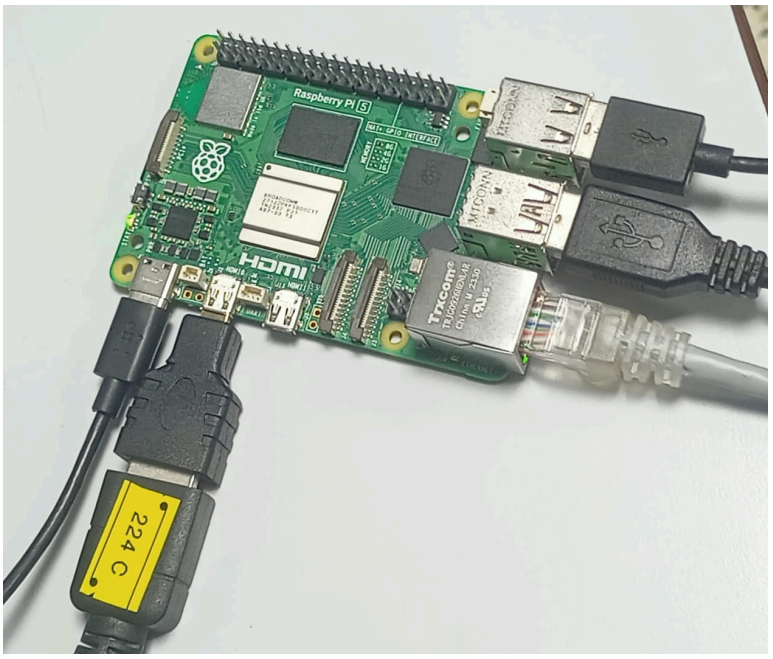


Fig : Raspberry Pi 5 setup

I had used the Raspberry Pi Imager settings to create a username and password,so I was able to go straight into the desktop environment.We have used nmcli linux command utility for managing network connections in Rpi .We setup a Static IP Address via nmcli to connect with LAN and seamlessly work in remote also .For Ex:

```
vboxuser@itslinuxfoss:~$ sudo nmcli con mod "Wired connection 1" ipv4.addresses 192.
168.1.200/24 ipv4.gateway 192.168.1.0 ipv4.dns 0.0.0.0 ipv4.method manual
vboxuser@itslinuxfoss:~$
```

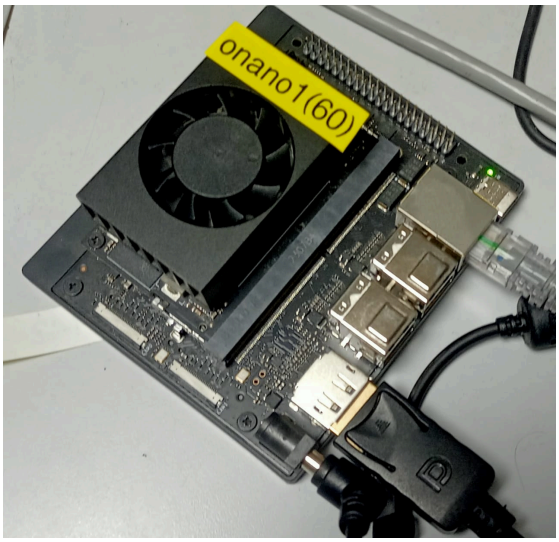
```
$ nmcli con show
```

```
vboxuser@itslinuxfoss:~$ nmcli con show
NAME                UUID                                  TYPE      DEVICE
Wired connection 1  188904bd-0b93-4759-a0ca-8a04827cc226  ethernet  enp0s3
lo                  77dde0a3-eaa9-44a4-b728-bba0b36e5db2  loopback  lo
vboxuser@itslinuxfoss:~$
```

- Jetson AGX Orin :

The NVIDIA Jetson AGX Orin Developer Kit and all Jetson Orin modules share one System-on-Chip (SoC) architecture. This enables the developer kit to emulate any of the modules and makes it easy to start developing the next AI-powered product. Jetson AGX Orin modules deliver up to 275 TOPS of AI performance with power configurable between 15W and 60W. This gives up to 8X the performance of Jetson AGX Xavier in the same compact form factor. Jetson AGX Orin is available in 64GB ,32GB, and Industrial versions.

- Jetson Orin Nano:



The NVIDIA Jetson Orin Nano Developer Kit sets a new standard for creating entry-level AI-powered robots, smart drones, and intelligent cameras,it also simplifies the process of starting with the Jetson Orin Nano series. Jetson Orin Nano series modules deliver up to 40 TOPS of AI performance in the smallest Jetson form-factor, with power options between 7W and 15W. This gives you up to 80X the performance of NVIDIA Jetson Nano. Jetson Orin Nano is available in 8GB and 4GB versions.

Hardware Device Specifications Comparison

	Jetson AGX Orin series			NVIDIA RTX 30 Series	Nvidia's RTX 5000-series	Jetson Orin Nano series		Raspberry Pi
	Jetson AGX Orin Developer Kit	Jetson AGX Orin 64GB	Jetson AGX Orin 32GB	Nvidia RTX 3090	Nvidia RTX A5000	Jetson Orin Nano Developer Kit	Jetson Orin Nano 8GB	Raspberry Pi 5
AI Performance	275 TOPS		200 TOPS	285 TOPS	200 TOPS	40 TOPS		-
GPU	2048-core NVIDIA Ampere architecture GPU with 64 Tensor Cores		1792-core NVIDIA Ampere architecture GPU with 56 Tensor Cores	10496--core NVIDIA Ampere architecture GPU with 328 Tensor Cores	8192 --core NVIDIA Ampere architecture GPU with 256 Tensor Cores	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores		Broadcom VideoCore VII (Graphics Only)
GPU Max Frequency	1.3 GHz		930MHz	2115 MHz	1695 MHz	625MHz		800 MHz
CPU	12-core Arm® Cortex®-A78AE v8.2 64-bit CPU 3MB L2 + 6MB L3		8-core Arm® Cortex®-A78 AE v8.2 64-bit CPU 2MB L2 + 4MB L3	16 Core 12th GEn Intel i9-12900K	32 Core AMD EPYC 7532	6-core Arm® Cortex®-A78AE v8.2 64-bit CPU 1.5MB L2 + 4MB L3		4 ARM coretex A76
CPU Max Frequency	2.2 GHz		2.2 GHz	5.2 GHz	2.4 GHz	1.5 GHz		2.2 GHz
DL Accelerator	2x NVIDIA v2		2x NVIDIA v2	Tensor Cores (Third Generation)	Tensor Cores (Third Generation)	-		-
DLA Max Frequency	1.6GHz		1.4GHz	1.6 GHz	1.1 GHz	1.0 GHz		-
Vision Accelerator	1x PVA v2					-		
Safety Cluster Engine	-				-	-		
Memory	64GB 256-bit LPDDR5 204.8GB/s		32GB 256-bit LPDDR5 204.8GB/s	24 GB GDDR 6X 102.4GB/s	824 GB GDDR6 90 .4GB/s	8GB 128-bit LPDDR5 68 GB/s		8 GB
Storage	64GB eMMC 5.1			-	PCIe NVMe SSD (optional	- (SD Card Slot & external NVMe via M.2 Key M)	- (Supports external NVMe)	microSD, eMMC

	Jetson AGX Orin series			**NVIDIA RTX 30 Series**	**Nvidia's RTX 5000-series**	**Jetson Orin Nano series**		**Raspberry Pi**
	Jetson AGX Orin Developer Kit	**Jetson AGX Orin 64GB**	**Jetson AGX Orin 32GB**	**Nvidia RTX 3090**	**Nvidia RTX A5000**	**Jetson Orin Nano Developer Kit**	**Jetson Orin Nano 8GB**	**Raspberry Pi 5**
Video Encode	2x 4K60 (H.265) 4x 4K30 (H.265) 8x 1080p60 (H.265) 16x 1080p30 (H.265)		1x 4K60 (H.265) 3x 4K30 (H.265) 6x 1080p60 (H.265) 12x 1080p30 (H.265)	2x NVENC, H.264/H.265 (8K AV1 Encode with RTX IO)	2x NVENC, H.264/H.265	1080p30 supported by 1-2 CPU cores		H.264, H.265
Video Decode	7x 4K30 (H.265) 11x 1080p60 (H.265) 22x 1080p30 (H.265)		4x 4K30 (H.265) 9x 1080p60 (H.265) 18x 1080p30 (H.265)	1x NVDEC H.264, H.265, AV1, VP9	1x NVDEC H.264, H.265, AV1	5x 1080p60 (H.265) 11x 1080p30 (H.265)		H.264, H.265
CSI Camera	16-lane MIPI CSI-2 connector	Up to 6 cameras (16 via virtual channels) 16 lanes MIPI CSI-2 D-PHY 2.1 (up to 40Gbps) | C-PHY 2.0 (up to 164Gbps)		-	-	2x MIPI CSI-2 22-pin Camera Connectors	Up to 4 cameras (8 via virtual channels***) 8 lanes MIPI CSI-2 D-PHY 2.1 (up to 20Gbps)	2x MIPI CSI
PCIe*	x16 PCIe slot supporting x8 PCIe Gen4 M.2 Key M slot with x4 PCIe Gen4 M.2 Key E slot with x1 PCIe Gen4	Up to 2 x8 + 1 x4 + 2 x1 (PCIe Gen4, Root Port, & Endpoint)		1x PCIe 4.0 x16	1x PCIe 4.0 x16	M.2 Key M slot with x4 PCIe Gen3 M.2 Key M slot with x2 PCIe Gen3 M.2 Key E slot	1 x4 + 3 x1 (PCIe Gen3, Root Port, & Endpoint)	1x PCIe 2.0 x1
USB*	USB Type-C connector: 2x USB 3.2 Gen2 USB Type-A connector: 2x USB 3.2 Gen2, 2x USB 3.2 Gen1 USB Micro-B connector: USB 2.0	3x USB 3.2 Gen2 (10 Gbps) 4x USB 2.0		USB Type-C connector: 2x USB 3.2 Gen2, USB Type-A connector: 2x USB 3.2 Gen2, 2x USB 3.2 Gen1, USB Micro-B connector: USB 2.0	3x USB 3.2 Gen2 (10 Gbps), 4x USB 2.0	USB Type-A Connector: 4x USB 3.2 Gen2 USB Type-C Connector for UFP	3x USB 3.2 Gen2 (10 Gbps) 3x USB 2.0	USB Type-C Connector: 4x USB 3.2 Gen2, USB Type-C Connector for UF

3.2) Software and Libraries :

- The NVIDIA RTX A5000, NVIDIA RTX 3090 runs CUDA v12.1 and cuDNN v8.9.6 .As Raspberry Pi5 has only CPU cores for compute it doesn't have CUDA while Jetson AGX Orin and Jetson Orin Nano runs CUDA v11.4 and cuDNN v8.6.0.
- We have used the latest stable version of PyTorch v2.3.0 with torchvision v0.18.1 on the A5000, 3090, and Raspberry Pi 5, and PyTorch v2.0.0 with torchvision v0.15.0 on the AGX Orin and Orin Nano as the training and inference framework.
- We use the same library versions for all A5000,3090 and Raspberry pi 5 (LTS versions) But these are newer than the versions in Orin AGX due to Nvidia JetPack's slower update cycle.
- We used the YOLO v8 DNN workload with Ultralytics v8.2.25, the latest stable version, to avoid version compatibility issues on the A5000, 3090, and Raspberry Pi 5, while Ultralytics v8.0.124 was used on the AGX Orin and Orin Nano.
- To analyse the results of the executed runs, it is necessary to represent the data graphically. To create static visualisations of this data, I used Python libraries such as Matplotlib, Seaborn, NumPy, and Pandas. These libraries helped me create various types of plots, including line plots, bar plots, box plots, violin plots, and heat maps. These plots were very helpful for identifying trends, comparing data, and summarising the results.

3.3) Models and Datasets

We have chosen five popular computer vision DNN architectures and datasets that can train within the available resource of hardware devices . Out of these five DNNS, two are used for image classification tasks, one for object detection, one for question answering and another for next word prediction.

1) ResNet-18

It is a widely-used convolutional neural network (CNN) architecture designed for image classification tasks .It consists of 18 layers, including convolutional layers, batch normalisation layers, ReLU activations, and fully connected layers. The key feature of ResNet architectures is the introduction of residual connections (or skip connections) which help to alleviate the vanishing gradient problem in deep networks . We train ResNet-18 on the validation subset of ImageNet, which consists

of 50,000 images and occupies 6.7GB on disk .It consists of 11.7 million parameters and 1.8 billion FLOPs (Floating Point Operations) .

2) MobileNet v3

MobileNet is a family of convolutional neural networks designed specifically for mobile and embedded vision applications where computational power and memory are limited. Our workload trains MobileNet using high-resolution images from the Google Landmarks Dataset v2 (GLD-23k), which consists of 23,080 photos of both human-made and natural landmarks, meticulously categorised into 203 different classes. The network consists of 5.5 million parameters and 225.4 million FLOPs.

3) YOLO v8n

YOLO (You Only Look Once) is a family of acclaimed real-time object detection models . It supports vision AI tasks, including detection, segmentation, pose estimation, tracking, and classification . We use the smallest of its family, to train on a subset of the MS COCO dataset, COCO minitrain, which has 25,000 images which take up 3.9GB on disk . It consists of 53 layers , 3.2 million parameters and 8.7 billion FLOPs .

4) BERT

BERT (Bidirectional Encoder Representations from Transformers) is a groundbreaking natural language processing (NLP) model developed by Google. It is based on the powerful Transformer architecture. We use this architecture with the SQuAD V2.0 Train dataset, which has 130K samples with a 40MB size. SQuAD (Stanford Question Answering Dataset) is a popular benchmark dataset for evaluating machine reading comprehension and question answering models. BERT consists of 12 layers, 110 million parameters, and 11.5 teraflops of computation.

5) LSTM

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) designed to overcome the vanishing gradient problem in standard RNNs.We use this workload for next word prediction with Wikitext training Dataset which has 36000 samples with 17.8 MB size.The Wikitext dataset is a collection of articles extracted from Wikipedia , commonly used for training and evaluating language models . LSTM consists of 2 layers , 8.6 million parameters and 3.9 billion FLOPs .

3.4) Performance Metrics

Several system parameters such as CPU and GPU utilisation, RAM usage and power (sampled every 1s) are measured and reported. The JTOP Python module (a wrapper around the tegrastats utility from NVIDIA) is used to measure CPU and GPU utilisation and power except in Raspberry pi 5. We profile a power mode for MAXN, we run the DNN training for all mini batches as we train for 5 epochs and record the minibatch time .

Fetch stall, GPU compute and end-to-end times for every training minibatch is measured. Fetch stall time is the time taken for fetching and preprocessing data that does not overlap with the GPU compute, and results in the GPU being idle. GPU compute time is measured using torch.cuda.event with synchronise to accurately capture the minibatch's execution time on the GPU.It includes the kernel launch time, and the forward and backward passes of training.The logging overhead is reported as fetch time, compute time, and epoch time to separate csv files. I have attached a code snippet in the implementation section .

We measure and report epoch time as the end-to-end (E2E) time required to process all mini-batches of each epoch, including fetch stall time, GPU compute time, and any framework overheads. In previous experiments, we excluded epoch 0 due to its bootstrap overheads and reported an average over only epochs 1 to 5 (referred to as 1+). However , in this experiment , we were unable to run 5 epochs on the Raspberry Pi 5 as it was shutting down due to undervoltage. Consequently, we ran only for epoch 0. To maintain consistency in experiments across all devices, we now include epoch 0 along with epochs 1 to 5 (1+) .

We note that the very first minibatch's training takes a long time to train, possibly due to PyTorch performing an internal profiling to select the best possible kernel implementation in this phase. Therefore, we discard the first minibatch profiling entry.1 epoch has a lot of batches (depends on dataset and batch size).1 epoch means one full iteration through all samples or all batches .If want to run for multiple epochs ,we can break the execution after particular batch.Since we have few epochs we are iterate through all the batches .

The Jetson Orin Nano powermodes span 20 CPU frequencies, 5 GPU frequencies, 3 memory frequencies and 6 CPU cores for a total of 18, 000 possible power modes .We randomly sample 180 of the available 1800 power modes for Orin Nano and collect profiling.data for ResNet and MobileNet workload

CHAPTER - 4 Implementation

4.1) Standalone Training ,Inference and Interleaving

Training is the process of teaching a neural network model using a dataset and a data loader, while inference is the process of making predictions using a trained model on new, unseen data. I executed training and inference runs of DNN workloads on Jetson AGX Orin3 and Orin4 simultaneously, running different workloads on each using Tmux. We adjusted CPU/GPU/Memory frequencies and core counts to generate different power modes for running. We used the “nvpmodel -m” tool to set the power mode configuration to MAXN. For example:

```
sudo nvpmodel -m 0
```

To run DNN workloads on any device, we need to install necessary libraries. To avoid conflicts that may arise between project-specific libraries, I created a new environment using the Anaconda platform. We needed to copy the dataset from another device on which we were training the neural network model, so I used the SCP (Secure Copy Protocol) command in the Linux system, which transfers files between the local host and the remote host, or between two remote hosts, in a secure way as

```
scp -r user@hostname:Source_folder Destination_folder
```

follows.

Since we are storing the dataset on an SSD, we need to mount it to make the file system accessible.

```
sudo mount /dev/nvme0n1p1 /media/ssd
```

Finally, we are ready to run the script file for training, inference, and interleaving.

Interleaving involves running multiple processes or tasks in overlapping time periods. In our implementation, interleaving means running training and inference concurrently so that the GPU is continuously utilised, reducing idle time and potentially speeding up both tasks. I also executed interleaved runs for training and inference tasks on different DNN workloads, such as ResNet training and MobileNet inference.

4.2) Multi-dimensional Binary Search

In this method, we extend the idea of a traditional binary search to deal with the multiple dimensions of CPU, GPU, memory, and cores that determine the optimal power mode. First, we find the middle value of the power mode range. Then we profile the power mode to get power consumption and training time. Based on the observed power, we decide which half to eliminate, similar to the traditional binary search. If the profiled power is under the budget, we compare the observed training time with the current best training time. If the observed training time is less than the best training time, we update the best one for further comparison and eliminate the lower half of the current dimension values. If the profiled power is over the budget, we eliminate the higher half of the current dimension values.

This process is repeated as long as the size of the search space is greater than 1. In this way, we identify the optimal power mode that minimises the overall time while staying within a specified power budget. During each iteration, the search space reduces by half, returning a solution in approximately (log N) profiling tries, where N is $cores \times cpu\,f \times gpu\,f \times memf$.

4.3) Performance comparison of various hardware

The training implementation varies depending on the specific hardware. For example, non-Jetson devices such as the 3090, A5000, and RPi5 do not require Jetson-specific tools like “nvpmodel -m” and “jtop”, which are used for the Orin AGX and Nano. For these non-Jetson devices, jtop logging needs to be removed.

On the Raspberry Pi, training must be adjusted to use the CPU, as there is no GPU available, and logging must be changed from CUDA events. Additionally, the need to mount external devices depends on where the dataset is stored. If the dataset is stored in the home directory, mounting may not be necessary.

As we discussed in the Performance Metrics section,below is the implemented Pseudocode snippet of training function for a given DNN workload in which Fetch stall, GPU compute and end-to-end times for every training minibatch is measured.

```
def train(model: torch.nn.Module, train_loader: DataLoader, epoch_fname: Any, fetch_fname: Any,
compute_fname: Any, vmtouch_fname: Any, dataset_outer_folder: str, reference_time: float, epochs:
int) -> None:

    model.train()
    criterion = CrossEntropyLoss()  # type: ignore
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)  # type: ignore
    vmtouch_dir = join_path(dataset_outer_folder, 'vmtouch_output/')  # type: ignore
    start1, end1, start2, end2, start3, end3 = create_cuda_events()  # type: ignore

    start_time = get_current_time()  # type: ignore
    epoch_count = 0
    img_idx = batch_size - 1  # type: ignore
    stabilization_time = 15
    num_epochs = 5
    for epoch in range(num_epochs):
        print('Epoch:', epoch, 'Begins')
        record_event(start1)
        record_event(start2)
        epoch_start_time = get_current_time()
        batch_count = 0

        for batch_idx, (images, labels) in enumerate(train_loader):
            current_time = get_current_time()
            print('Current time:', current_time)
            if current_time - epoch_start_time > stabilization_time:
                batch_count += 1
            if batch_count == MAX_BATCH_COUNT:
                break

            print('Batch index =', batch_idx)
            images, labels = move_to_device(images, DEVICE), move_to_device(labels, DEVICE)
            record_event(end2)
            record_event(start3)

            optimizer.zero_grad()
            loss = criterion(model(images), labels)
            loss.backward()
            optimizer.step()
            record_event(end3)
            synchronize_cuda()
            record_event(end1)
            synchronize_cuda()
            fetch_time = elapsed_time(start2, end2)
            compute_time = elapsed_time(start3, end3)
            epoch_time = elapsed_time(start1, end1)

            log_info(fetch_fname, epoch, batch_idx, fetch_time, current_time - reference_time)
            log_info(compute_fname, epoch, batch_idx, compute_time, current_time - reference_time)
            log_info(epoch_fname, epoch, epoch_time, current_time - reference_time)

            record_event(start1)
            record_event(start2)
            img_idx += batch_size
        print('Epoch:', epoch, 'Ends')
        epoch_count += 1
```

4.4) Generalizability to another device

The Jetson devices come with a number of predefined power modes that users can choose from.We use Unix utility used to read information about the frequency scaling .Specifically, it lists the available frequencies for the first CPU (cpu0) on a Linux system for Jetson Device . We configured a custom power mode by specifying the number of CPU cores enabled, and the frequencies of CPU, GPU and RAM (External Memory Controller (EMC)). The power mode can be changed on the fly, without any system downtime.

To save results for each power mode, we iteratively change the CPU frequency rather than running all power modes in one execution. This way, if an error occurs, we can identify which CPU frequency caused the error, discard it, and restart with that particular CPU frequency again.Below I have provided a sample script file as pseudo code for training.

```
# Define parameters
prefetch_factor = 2
no_workers = [4]
external_device = "nvme0n1"
batch_size = 16

# Remove previous log and result files
remove_files("mn_nw*")
remove_files("log_file_*")

#Available CPU frequencies fro next run
#available_frequencies = [c2,c3,c4,c4,c5,c6,c7,c8,c9.c10]

# Define CPU, GPU, and memory frequency values
cpu_cores_values = [2, 4, 6]
cpu_frq_values = [c1]
gpu_frq_values = [g1, g2, g3]
mem_frq_values = [m1, m2]

# Iterate over all combinations of CPU, GPU, and memory frequencies
for cpu_frq in cpu_frq_values:
    for gpu_frq in gpu_frq_values:
        for cpu_cores in cpu_cores_values:
            for mem_frq in mem_frq_values:

                # Generate and apply the power mode configuration
                generate_power_mode(cpu_cores, cpu_frq, gpu_frq, mem_frq)
                apply_power_mode()
                enable_fan_and_clocks()
                log_current_configuration()

                # Run the training script and log the output
                run_training_script(no_workers, prefetch_factor, external_device)
                kill_running_python_processes()

                # Save results and logs for the current configuration
                save_results_and_logs(cpu_cores, cpu_frq, gpu_frq, mem_frq)
```

CHAPTER - 5 Result and Analysis

5.1) Performance Analysis of Hardware Devices

Each DNN workload runs on all Hardware devices except BERT on Raspberry pi5. The minibatch time,fetch time and GPU compute time are reported . Every training workload is run for 5 epochs except raspberry pi 5 run for epoch 0 (we discussed the reason in the performance metrics section).We calculated Average epoch times across all devices as a summation of minibatch time for all epochs divided by number of epochs and report the average epoch times .Since we were running few epochs we were iterating through all the batches.We configure PyTorch's Dataloader to run with 4 worker processes to enable pipelined and parallel data fetch and pre-processing . We use a batch size of 16 for all training workloads .

We observe that the NVIDIA RTX 3090 is significantly faster than the NVIDIA RTX A5000, which can be explained by the 3090 having more CUDA cores of the same Ampere generation (10, 496 vs. 8, 192). The increased number of CUDA cores in the 3090 allows for more parallel processing power.

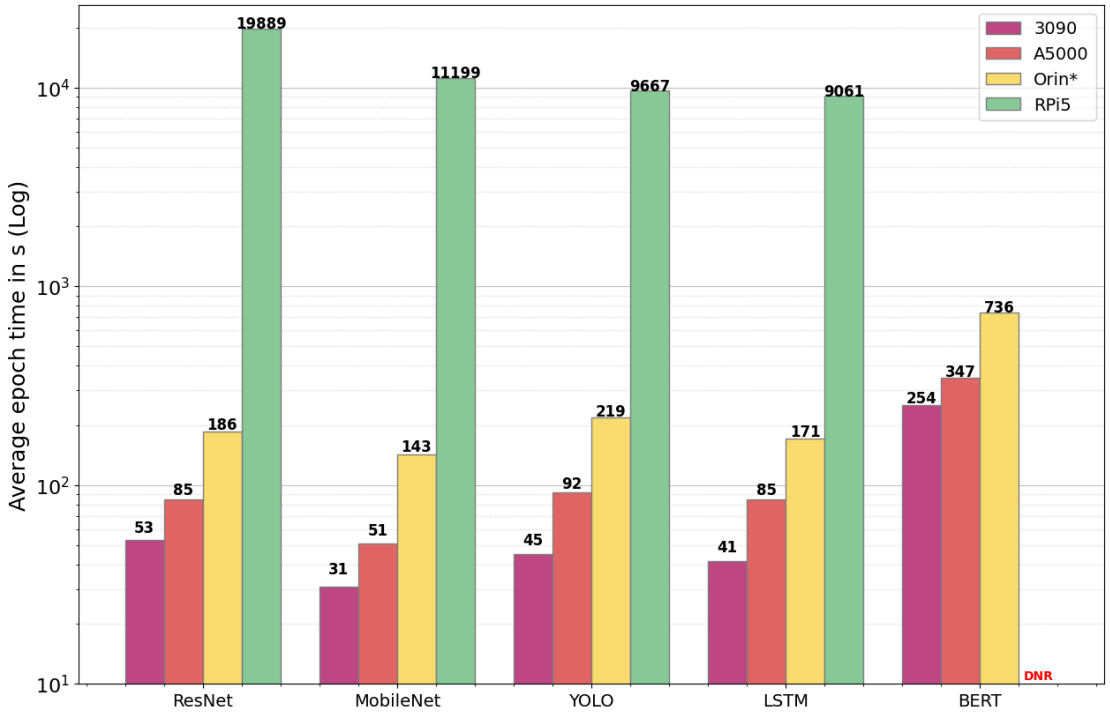


Fig: Average epoch times across various edge and server devices.

The NVIDIA Jetson AGX Orin is slower than both the A5000 and the 3090 as this is reflected in its longer average epoch time shown in figure. Despite being

based on the same Ampere architecture, the reduced core count 2048 of Orin AGX , resulted in lower overall performance .

The Raspberry Pi 5 has the lowest average epoch time among all the hardware devices tested.It trains using just the ARM CPU cores, and is two orders of magnitude slower than the Orin. BERT was unable to run on the RPi5 (DNR in Figure) as it ran out of memory in 8GB of memory available in the RPi .

	3090	A5000	RPi5	Orin AGX
CPU	16 core 12th Gen Intel i9-12900K	32 core AMD EPYC 7532	4 core ARM Cortex-A76	12 core ARM A78AE
Max CPU frequency (MHz)	5200	2400	2400	2200
GPU(arch/CUDA cores)	Ampere ,10496	Ampere,8192	VideoCore VII (graphics only)	Ampere , 2048
Max GPU frequency (MHz)	2115	1695	800	1300
RAM (GB)	128 (CPU) + 24 (GPU)	512 (CPU) + 24 (GPU)	8	64 (shared)
Peak Power (W)	350	230	27	60

Table: Specifications of Server GPUs and Edge Devices

5.2) Training on Jetson Orin Nano

We ran two DNN workloads ResNet and MobileNet on Jetson Orin Nano. We configured a custom power mode by specifying the number of CPU cores enabled, and the frequencies of CPU, GPU and RAM (External Memory Controller (EMC)). We choose 180 of the available 1800 power modes for profiling data for ResNet and MobileNet workloads .

Using the ResNet workload trained on Orin AGX as the reference model for prediction, we transfer-learn to Orin Nano by retraining using 50 random power modes for ResNet (or MobileNet).We validate the predictions of the transferred model using all 180 power modes we profile for the workload.we observe low median time and power errors, with MAPEs of 7.85% and 5.96% for ResNet, and 8.98% and 4.72% for MobileNet. This confirms the strong generalizability of PowerTrain to even Jetson accelerators that are 6.9× less powerful than the Orin AGX.

CHAPTER - 6 Conclusion And Future Work

In this comprehensive report, we have conducted a detailed study on the contrasting compute performance of DNN training on various edge and server devices. We have also extended our evaluation of PowerTrain to another less-powerful Jetson device in the latest Jetson generation, the Jetson Orin Nano. Additionally, we propose an efficient traversal technique to optimise the performance of inference, training, and concurrent workloads. We developed a multi-dimensional binary search strategy through which we identify the optimal power mode that effectively minimises the overall time while staying within a specified power budget.

In the future, we plan to explore variants like time and power prediction for concurrent training and inference workloads. We also plan to look into choosing separate power modes for inference and training, and dynamically switching between the power modes during interleaving. Training batch size is another important variable we plan to include in our comprehensive study.

I learned many things in both theoretical and practical aspects through this project. My guide and all the other students of Dream Lab provided me with constant support and help whenever needed. I am also thankful to the Indian Academy of Science for providing me with this opportunity.