UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Sciences

Department of Electronics and Computer Science

A project report submitted for the award of
Meng Electronic Engineering w/Industrial Studies

Supervisor: Geoff Merrett
Examiner: Kees De Groot

Implementation of multi-layered perceptron for computer vision on edge computing

by Sourish Mukherjee

3rd April 2022

Acknowledgements

I would like to thank Dr Jia Bi, Lei Xun and Professor Geoff Merrett for all the help and guidance provided throughout this project.

I would also like to thank friends and family for their support provided.

Abstract

Convolutional neural networks are the most commonly used models for computer vision, due to their historically high accuracy. However, in recent times, another model which has gained momentum in usage for vision, is the transformer. Due to its competitive accuracy rates and also being less memory-heavy than CNN, it is being considered. A new proposed model, MLP-Mixer, which consists of only multi-layered perceptions(mlp), has been shown to have similar accuracy rates as CNN, whilst also being smaller in size and faster. This makes it ideal for implementing into hardware-constraint devices, such as the Nvidia Jetson Nano. Moreover, when considering the limitations of the hardware- memory usage as well as energy consumption, the MLP-Mixer is the better option, where the throughput is generally higher, hence lower latency. In this progress report, the three networks will be introduced and explained, as well as the project itself- including milestones and the progress of the project.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.
If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.
You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.
We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced**.**

The material in the report is genuine, and I have included all my data/code/designs. We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment**.**

I have not submitted any part of this work for another assessment. If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Contents

Chapter 1 Introduction

1.1 Problem Statement:

In modern deep learning models for computer vision tasks, there are two main architectures used, Convolutional Neural Networks (CNNs) and Vision Transformer (ViT) [11]. To learn features in different spatial locations and channels of input images, CNNs use multiple building blocks (e.g., convolutional layers, pooling layers, and fully connected layers), which has led to high accuracies, hence has become standard for computer vision. However, ViT has become a competitive alternative to CNN that utilizes multiple attention mechanisms to weigh the importance of each part of the input data differently. To achieve high accuracy, however, there is an issue involving CNN and ViT models. Both models require a high computational cost and large memory requirements due to their complex architecture, which becomes a limiting factor on embedded computers with a lot more limited performance capability.

Therefore, other models are required to be researched, tested, and compared with for vision on embedded. Currently, a novel alternative to both architectures has been proposed by [12], coined 'MLP-Mixer', which does not use convolutions or self-attention. Instead, the architecture of MLP-Mixer is completely based on the multi-layer perceptron (MLP), which is repeatedly applied in the spatial location and feature channels [12]. MLP-Mixer, with a simple structure, can achieve competitive performance and higher throughput than CNN and ViT on a host computer. Due to its higher throughput and its capability to run with less computations, it is worth considering deploying MLP-Mixer on embedded platforms for the model inference process.

1.2 Objective:

The main goal of this project is to successfully run the MLP-Mixer model on an embedded computer and measure the performance capabilities. The path flow of the project starts with model being trained, with a chosen dataset, on a host desktop until comparable accuracy has been achieved, before being implemented into the embedded computer, the Nvidia Jetson Nano. The hardware specifications of the Jetson Nano include 4GB unified ram, 16GB eMMC storage []. Although, there are some options in terms of dataset, a pre-existing mlp-mixer model has been pre-trained on the Imagenet-21k model, and the simplest implementation would be to fine-tune the model to the CIFAR-10 as it is one of the most commonly used and understood dataset, hence measuring and comparing accuracy would be easier.

Furthermore, as there are memory limitations, issues regarding computational requirements may arise. Therefore, consideration for the model to undergo further modification, mainly compression required to satisfy the hardware limitations may arise. Once the constraints are satisfied, the model will be implemented and compared with a CNN model and Vit model of similar sizes, based on the evaluation matrix. The evaluation matrix involves four parameters, including test accuracy (which measures how accurate the model is at classifying), test latency (the time is taken to achieve a classification of the test data), FLOPs and the number of model parameters. Additionally, observation of the relationship between FLOPs and latency of MLP will be done (e.g., whether smaller FLOPs can reduce the latency of MLP on Jetson). An additional target will be to find a solution for controlling the FLOPs or parameters if the MLP-Mixer model is over resource limitations on Jetson by static solution and/or dynamic solution.

To summarise, the outcome of this project is to:

- Train Mixer model on Cifar-10 dataset, and successfully run it on host computer using GPU hardware accelerator.
- Run the model onto the Jetson Nano, whether by undergoing model compression to a suitable size.
- Compare model's performance on Nano against suitable CNN and Vit models using described evaluation matrix.
- Undergo further improvements by either controlling FLOPs or parameters of the model.

The Evaluation Matrix =

Number of Parameters
Model Accuracy
Latency
FLOPs

Chapter 2: Background and report of literature review

2.1 Computer Vision

Computer vision is a subsection of artificial intelligence which uses mathematical computation to process images [1]. Its main use is to detect key features which are vital in classification of the input visual data. A feature is a piece of information which describes the image. Features in images can be classified into three main forms: spectral, geometric and texture [2]. Spectral features mainly compromise of information of the pixels itself. Pixels contain information of colour and brightness intensity, without any regard of the pixel's position in the image matrix. Geometric features include detection of different shapes in the image. There are various different algorithms which are used to detect shapes such as edge detection or region detection. Both involve finding the differences between consecutive pixel values, hence position of pixels in the matrix is vital for this feature extraction [3]. Texture feature involves using colour and intensity of pixels as well as its arrangement to detect different regions of the image. There are multiple ways to use these features extracted to perform classification, although CNNs are the most common models used to do so. In the following sections, the most common methods used in computer vision will be compared.

2.1.1 Multi-Layered Perceptron (MLP)

MLP is a simple feedforward neural network which undergoes a set of operations on the input data to provide an output. It undergoes a training process where an iteration of training data batches is inserted as input and for each iteration, the loss function (difference between the model prediction of output and the true output) is reduced until an acceptable accuracy rate is reached [4]. It is to be noted that loss function can only be calculated for supervised learning.

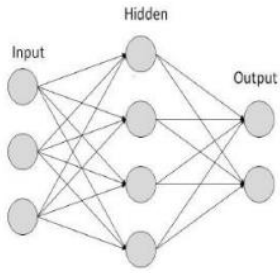


Figure 1: Simple 3-layer MLP.[4].

A perceptron is a simple mathematical calculation which takes in multiple inputs and assign weight coefficients to each and bias values to it and perform summation. An activation function is the performed onto the weighted sum. An activation function is a non-linear operation (for nn) performed on the sum to normalise or transfer the output onto a range. Multiple perceptions make up a feed forward neural network.

Back-propagation is a method used to reduce loss function such that the accuracy of the model increases, during training of the model. As the output prediction y is dependent on the weights

and previous inputs, where those inputs are again dependant on previous weights and previous inputs, until the first layer is reached. As the initial inputs are constants that can't be varied, the loss function is dependent on weights, hence the loss function is derivate and which propagates back through each layer to change the weight coefficients [5].

$$W_{ij} = W_{ij} + \partial W_{ij}$$

Equation 1 : How weights are updated [5].

2.1.2 Convolutional Neural Network (CNN)

Convolutional neural network [6] follows the mlp in that it involves feed forward model with back-propagation. However, the architecture of a CNN model varies such that successful feature extraction is possible. The architecture contains three main layers – convolutional, pooling and fully-connected.

Convolutional layer as the name suggest performs convolution function between a kernel and the image. A kernel (filter) is a n × n 'window' which contains trainable values that slides across the image pixel matrix and performs element-wise multiplication which the results then undergo some mathematical operation (addition, mean value, etc) to give a resultant value placed on a destination pixel. The values in the kernels are trained such that the output image displays more clearly the specific features detected on the original input. For CNN, channels refer to the different number of kernels used in a layer on the image.

For coloured images, it is slightly more complex. Since coloured images in fact have three-pixel mappings- red, blue and green, the channels for CNN kernel correspond to that. Furthermore, there are 2 main types of convolutions used: spatial and depth-wise.

Spatial convolution [7] is more straightforward, with n × n × c kernel size. Spatial convolution follows normal 2d convolution, where the kernel slides throughout the input image, performing multiplication to output a m × n × 1 output image, where m and n depends on the dimensions of the input image and the kernel dimensions. To change the number of channels of the output image, say to O, a simple kernel of dimensions n × n × c × O is required.

Depth-wise convolution [8] involves spatial convolution, but rather than doing across all the channels together, convolution is done across each channel separately. Once the convolutions are done, each channel is stacked back together to give 3-d image. The next step involves pointwise convolution, where the stacked channels undergo convolution but the kernel size is 1 × 1 × 3, which results in a m × n × 1 image.

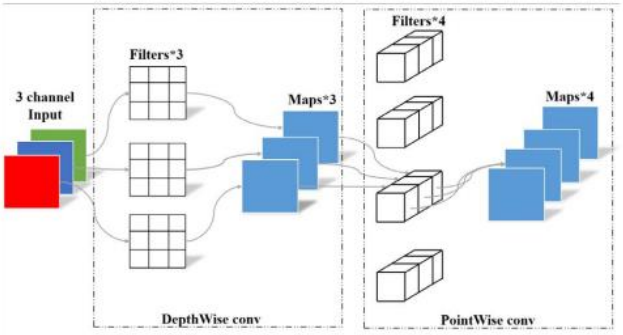


Figure 2: Showing how depth-wise and point-wise convolution works. [8].

The pooling layer's main purpose is to reduce the size of the image. A kernel, which is not trainable, again slides through the image performing element-wise multiplication, and the maximum argument is positioned into the destination pixel. Finally, the fully-connected layer is similar in architecture to mlp.

One of the reasons as to why CNN is a popular network used for image classification is due to the fact that due to its ability to retain spatial location data [9]. Each pixel contains information of its position, which although via pooling and convolution, is slightly lost, still helps with feature extraction. This is not possible with a basic mlp as it can't retain any spatial information since the input data is flattened.

2.1.3 Transformer

Transformer [10] is an attention based neural network which although was first suggested for natural language processing (nlp). However, attention neural networks are also widely used alternative for vision. Attention is a function where three input elements: query and key-value pair undergo manipulations to provide a vector output.

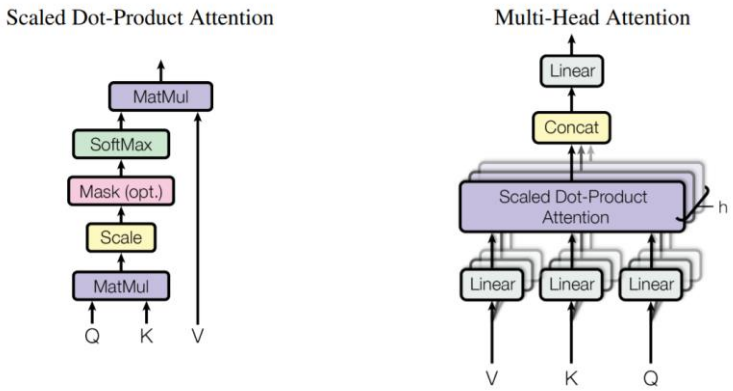


Figure 3: Two attention modules used in transformer model. The scalar Dot-Product Attention is used in multi-Head.[10].

The diagram shows multi-head attention used in transformer. It involves scaled dot-product

attention, which functionality can be seen in dot-product diagram. The multi-head attention takes in h number of values, keys and queries and concatenates the results.

Another key aspect of the transformer is positional encoding. Positional encoding allows the model to retain information on the relative or absolute positioning of the input sequence. This encoding can either be learned or fixed depending on the model and is used in encoder and decoder stacks.

For vision, Vision transformer (Vit) [11] is a very popular model used. The input image is split into n number of patches, with each patch having p^2 dimensions (where n = original height x original length / p^2). Positional encoding is done on the patches such that spatial information can be retained. The patches pass through a transformer encoder, which contains multi-head attention, amongst other components.

The result of smaller Vit-L/16 model [11] validates its use for vision as it outperforms the Resnet CNN model in terms of accuracy for different datasets, including cifar10 (99.74 vs 99.63) whilst requiring less computational resources.

2.1.4 MLP-Mixer

MLP-Mixer[12] is an architecture designed completely from mlp, and doesn't require convolution or attention. Similar to Vit, the input images are split into patches which is first passed through a projection layer outputting a table, which the paper refers to as **X,** where the output of the fully connected for each patch is stacked on top. **X** is then inserted into the mixer layer. The mixer layer contains 2 mlp blocks, the first one being token mixing and the second one channel mixing.

Token mixing inserts the columns of the table **X** as input to the mlp, as shown in the diagram (via transformation). The purpose of token mixing is for "mixing features at different spatial locations". [12]. This process is analogous to 1 channel depth-wise convolution that takes place in CNN as various spatial information across the whole image is taken as input to mlp, and 1 channel depth-wise convolution is basically as 2D convolution. The result of token mixing is added to the initial table via skip-connections, which is then inserted to channel mixing.

Channel mixing, in comparison inserts the rows to the newly edited table to the mlp which results as again added. This mixing focuses more on feature mixing per spatial location. Whilst token mixing is similar to depth-wise convolution, channel mixing is similar to spatial convolutions with kernel dimension 1x1, since each mlp performs multiplication across the values in each patch. However, the advantage of the mixer models. These procedures allow image classification where singular mlp would struggle to achieve high accuracy, and both procedures are also present in CNNs and Vit.

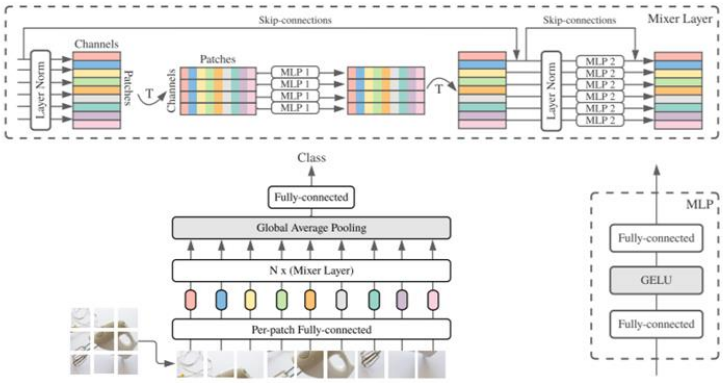


Figure 4 : A top-level layout of the proposed MLP-Mixer model.[12]

The results of the architecture provided in [12] are comparable to that of CNN and Vit. In fact, looking at pre-trained on JFT-300M, the same dataset used in the transformers section, it is visible that the Mixer model has very similar accuracy (94.77-Mixer vs 95.21-Vit), whilst having greater throughput and requiring less computational power than the other two models.

2.2 Model Compression

As stated in the project goals, the hardware specifications of the Jetson Nano means that not only memory, but also computational power and energy supply are very limited, meaning that to successfully implement the model, model compression is required. In this report, the main compression techniques discussed are deep compression and knowledge distillation.

2.2.1 Deep Compression

Deep compression [13] actually involves three stages to reduce either the number of parameters in the model and/or reduce the total memory the parameters require, which can lead to between a total of 36x to 49x reduction. The stages in chronological are pruning, quantization and Huffman coding.

Pruning is technique that has been widely used in CNN but can also be implemented in any architecture. It involves training the model such that it is connected. Then low-valued weight parameters, below a certain threshold, are masked (removed) and the model is re-trained. Structured pruning involves removal of whole layers or groups of parameters at a time whereas unstructured pruning iterates through each parameter, comparing to threshold value. Due to simplicity and flexibility in the level of sparsity level, which basically is the percentage of zero values in a matrix, unstructured pruning is preferred. The final values are stored in compact sparse row or compact sparse column.

Quantization involves decreasing the bit size of each weight by essentially making the values into integers. In addition, weight sharing is also possible via k-means clustering and through back-propagation, the “centroïdes”- which are the shared weights, are updated. Huffman Coding is a lossless compression technique which “works from leaves to the root in the opposite direction.” [15]. [15] details the method of how Huffman coding works.

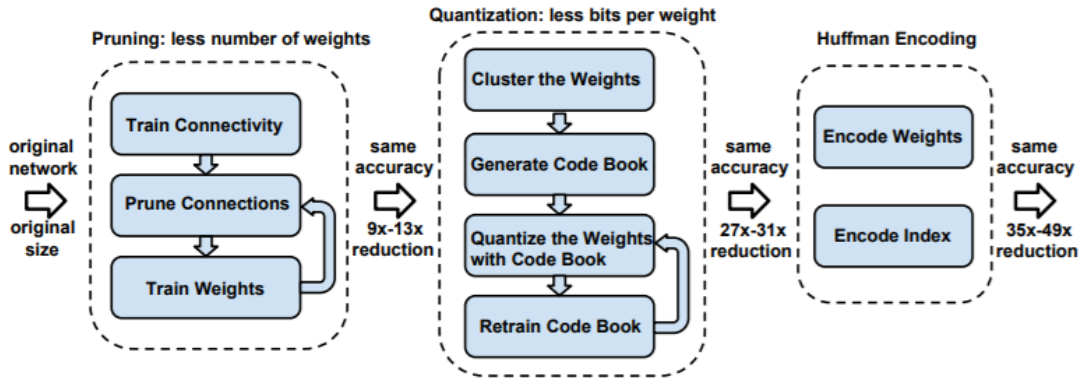


Figure 5: Shows the three phases used in deep compression.[13]

2.2.3 Penalty based pruning

Regularization [21] is a technique used in machine learning and AI, which involves reducing the size of the weights, such that overfitting does not take place. Since the magnitude of the weights are reduced, it can also be used as a form of pruning by weight decay to induce sparse weight tensors, since the smallest weight values will be forced to zero. The two most common form of regularization are norm 1(L1) and norm 2(L2).

L1 norm is described as a way to measure the absolute distance between two points. If point A is $[x_1,y_1]$ and point B is $[x_2,y_2]$:

$$L1\ Norm = |x_1-x_2| + |y_1-y_2|.$$

Equation 2: L1 Norm

In machine learning, it is used to add to the loss function such that the amount of loss is increased.

$$L_{new} = L + |w|.$$

Equation 3: Loss function after L1 regularisation

On the other hand, L2 norm is the Euclidian distance between two points.

$$L2\ Norm = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Equation 4: L2 Norm

For regularization, the loss function is described as:

$$L_{new} = L + |w|^2.$$

Equation 5: Loss function after L2 regularisation

L1 norm is more commonly used for sparsity, since looking at the term $\frac{dL}{dw}$,, the differentiation of the norm 1 term gives a constant value, hence the weight values are reduced in equal amounts each time regardless of its magnitude. L2 on the other hand is a polynomial function and so its

derivative is a linear function. Therefore, as the weight values decrease, the value of the term $\frac{dL}{dw}$, reduces each iteration, hence more steps are required.

2.2.2 Sensitivity pruning

Sensitivity based pruning involves removal of weights or neurons based on how much of an effect it has on the loss function. Initially, connection sensitivity analysis [18] is done of the network. There are different methods and implementations that can be done to perform sensitivity analysis, change in one parameter with respect to another, however, [19] provides a simplistic approach taken to measure sensitivity. The approach involves identifying the overall change in loss value when a single neuron change- specifically, before the activation layer. Mathematically, the term is defined as:

$$S_{n,i}(\boldsymbol{y}_N, p_{n,i}) = \frac{1}{C}\sum_{k=1}^{C}\left|\frac{\partial y_{N,k}}{\partial y_{n,i}} \cdot \frac{\partial y_{n,i}}{\partial p_{n,i}}\right|.$$

Equation 6: Sensitivity measurement of a neuron [19]

where y_n is the output of the neuron, y_N is the predicted output per label C, and p_n is a function coined as post synaptic potential, which is the weighted sum prior undergoing through activation layer. Furthermore, to implement into the loss function during backpropagation, an insensitivity parameter introduces, which is described as max{0,1-sensitivity}. The implementation is to the loss function is similar to that of L1 regularisation, but the main difference is that the insensitivity parameter is also included such that only weights connecting to the removal neurons are reduced to zero.

$$w_{n,i,j}^{t+1} = w_{n,i,j}^{t} - \eta\frac{\partial L}{w_{n,i,j}^{t}} - \lambda w_{n,i,j}^{t}\overline{S}_{n,i}$$

Equation 7: Weight update with sensitivity parameter [19]

2.2.4Knowledge Distillation

Knowledge Distillation [14] requires a fully-trained teacher model, from which knowledge is gathered and applied to a smaller untrained student model. The teacher model is usually trained for hard targets (one hot encoding), but if SoftMax is done for the logits [14] with certain value of temperature (denominator variable), it provides soft targets, which are used for training of the student model.

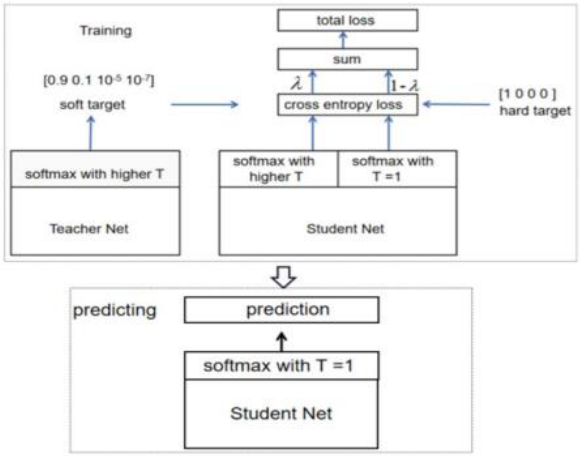


Figure 6: Shows the process behind knowledge distillation. Reference: C. Xi, X. Zhiqiang and C. Yuyang, "Introduction to Model Compression Knowledge Distillation," 2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP), 2021, pp.

2.2.5 Precision

Usually, models in pytorch operate at single precision, which means that all the parameters are saved as 32 bits floating points(fp32). However, as the models are trained and as accuracy increases, computational costs and memory usage increases with it [31]. Therefore, a simple solution would be to use half-precision(fp16), which halves the memory requirements to run the model.

However, when fp16 is used during training, when weights are multiplied with the learning rate, the value may be less than 2^{-24} becomes zero, hence some weights don't get updated, which leads to high loss of accuracy. Therefore, to tackle such issues, mixed precision is introduced. Mixed precision includes both single point and half-precision, where half-precision is used for calculation, and a single point copy is kept such that during update, it can be used, as to avoid no zero-weight update.

2.3 Conclusion of Literature Review

Although, CNN is the most common type of network used for computer vision, as it performs at high accuracy rates, there have been emerging alternative architectures such as Vit and more recently mlp-mixer which have performed competitively. Additionally, due to its high throughput, the mlp-mixer is deemed as adequate model to use for a device such as the nano with low ram specifications, where inference latency is a major deciding factor. However, when considering implementation of a model on an embedded device, the issue of insufficient memory requirements (CPU and GPU ram) must be considered, hence model compression techniques must be compared for suitability.

The literature covers two main methods for compression- knowledge distillation and deep compression. Although both techniques are valid techniques, when implementing the mlp-

mixer into a hardware constraint device, deep compression is the preferred option here due to the fact that for knowledge distillation, two models are required- a trained parent model and a smaller student model. This entails more steps and requirements compared to deep compression since a larger model is required to be first trained to a sufficient level first, from which the information can extracted (SoftMax) and used to train the smaller model to a high degree of accuracy. Furthermore, compression rate can be controlled and adjusted using hyperparameters which would be useful in understanding further latency and model size relationship on the Nano, whereas with knowledge distillation, the model reduction would be

Implementing deep compression focuses mainly on pruning and quantisation. There are two main forms of pruning considered in this literature, saliency pruning and penalty-based pruning. Generally, saliency-based pruning such as magnitude pruning (pruning weights/neurons below certain threshold) or sensitivity pruning, is preferred as they result in higher accuracy when implemented. However, penalty-based pruning should also be considered due to its simple implementation and as it's a very common technique use in AI. Additionally, depending on the criteria of saliency pruning, penalty-based pruning can provide faster training duration. Finally, the last consideration is precision and quantization. As training is done on host compute only, mixed precision training is not required. However, when considering compression, quantising parameters to either fp16 or even int8 should be considered for inference, as computational requirements decrease. This can therefore be considered for further reduction if necessary.

Chapter 3: Training on Host PC

3.1 Training model on TPU

Initially, the code for the implementation of mlp-mixer model size B-16, retrieved from GitHub [29], was used to run the model to measure accuracy, and memory usage. The model uses pretrained (with Imagenet21k) parameters to undergo transfer learning [20] and training on cifar10 dataset. As the model was very large, both TPU and GPU hardware accelerators were used to run to achieve faster measurement. When running on the Collab TPU (), the maximum accuracy achieved by the model was 97.1% after 4 validation epochs, which reflects the findings of the paper [12], and within comparison of the Vision transformer model designed to have accuracy around 97%.

Validation epoch	Accuracy	Total time elapsed (mm:ss)
1	0.964	00:55
2	0.967	00:56
3	0.969	00:55
4	0.971	00:56

Table 1:Displays accuracy and total time elapsed per validation epoch

3.2 Training model on GPU

When running training and validation of the model on Collab with the GPU accelerator, there was error during validation, whilst also being unstable requiring multiple restarts to Collab kernel to operate, meaning validation was not possible on it. Therefore, the best solution was to instead use another source of code [3] based on Google's implementation, designed to run with GPU accelerator for cifar10 dataset. Due to its huge ram requirements, the model was trained on the ECS Alpha cluster [22]. Similar to the previous implementation, this also loaded the ImageNet pretrained parameters onto the model, train and run validation. The results of running 100 epochs of training and validation were maximum 93% accuracy. With 1000 epochs of training, the maximum accuracy reached was 95%, which is comparable to the results achieved by Google using the TPU. Furthermore, the time elapsed for each validation run was 44 seconds, with 32 batch size.

The model was comparatively accurate, whilst running just validation, the total memory usage was 4.379 GB (appendix c), more than the maximum available ram on the Nano. As anticipated, the total memory usage needed to be reduced so that the model could run on the Nano and even more constraint devices. Therefore, model compression techniques needed to be implemented.

Chapter 4: Model Compression

4.1 Unstructured Magnitude Pruning

As the dense training involved validation after 100 training steps, the initial method was to implement pruning just before validation occurred. Additionally, a user defined hyperparameter would determine the sparsity increment steps, such that after certain number of validations runs, the total sparsity would be around 90%. However, after pruning, due to loss of connections, there would be a drop in accuracy, which means that fine-tuning is required Therefore, to keep accuracy as high as possible, pruning would be done after each training epochs, rather than every 100.

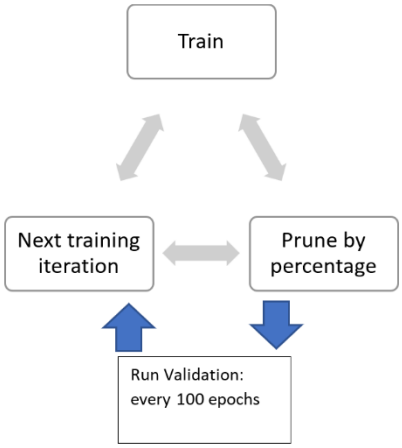


Figure 7: The design chosen of the pruning process.

When implementing magnitude pruning, pytorch contains class and methods which performs unstructured magnitude pruning, hence the initial plan was to implement pytorch l1 pruning method [23] into the code. However, what was not considered was the structure of the mlp-mixer code. The modelling follows three layers, described in modelling.py as classes, where the lowest layer performs forward propagation, hence contained the parameters (bias and weights). Accessing the weight parameters as named was not possible just by iteration, due to the change in name of each weight tensor defined by modelling.py. This would be a time-consuming process and may result in failure. Hence, another method of adding norm 1 magnitude-based pruning was attempted.

4.2 Alternative approach to implementing L1 pruning

An alternative method was used to implement L1 pruning, displayed in figure 8, which followed an intuitive understanding of how pruning occurs on a model.

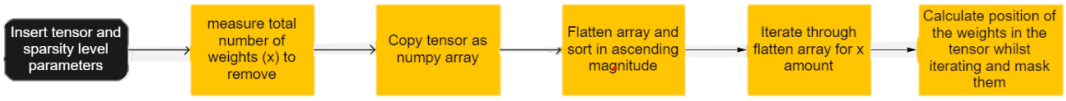


Figure 8: Flowchart of how L1 designed algorithm operates.

The hyperparameter sparsity level controlled the number of weights to remove. The algorithm successfully removed the smallest weights. However, it was not possible to integrate it into the mlp mixer code due to its time complexity. Even though the design only involved a single iteration, when implementing into the Mlp-mixer, it would require an iteration of the weight parameters such that each tensor could be inserted as argument. This transformed the design into a nested for loop, its time complexity is described as O (n^2), which was unusable, especially when the mlp-model has approximately 59 million parameters.

This algorithm used brute force to measure Manhattan distance and prune, which was why the complexity was high. Other algorithms, such as graph methods [24] were also considered due to more efficient complexity, but as it required prior understanding and practice, and to refrain from deviating from original target, other pruning implementations were then researched on. Reviewing literature, penalty-based regularisation was then implemented as there had been well-documented examples of the method.

4.3 Penalty based Pruning

Norm-1 (equation 3) was the preferred method for forming sparse connections in neural network. This involved introducing a regularisation coefficient parameter, λ, which controls how strongly the weights would be penalised. The code was retrieved from [32]. Additionally, as the parameters were in fp32, a function was introduced which would force negligible values to zero, depending on what the user defined as the decimal point. This was introduced such that the total sparsity level of the model could be measured and incremented, by negligible parameters. The accuracy of the model was measured for every 100 epochs of training.

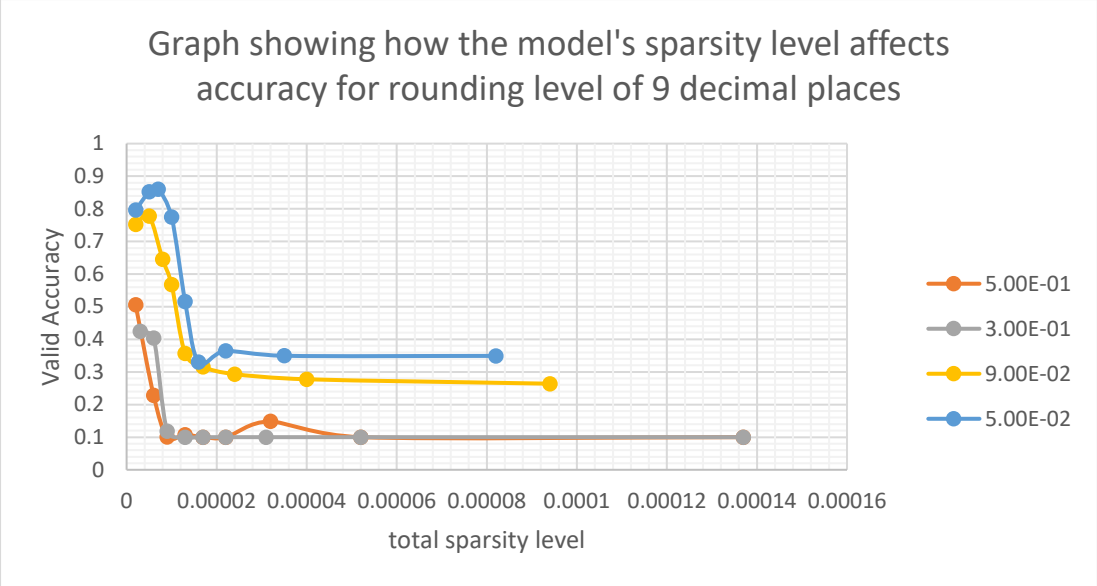


Figure 9: Graph showing the accuracy achieved per sparsity level with different regularization coefficient values for precision of 10 decimal places.

Figure 9 showed with higher values of λ (0.5), the accuracy dropped to around 10%, resulting in a bad model. This was thought to be due to higher penalisation of all of the weights, resulting in high levels of underfitting. Furthermore, there was an unexpected difference between the first and second measurements of accuracies. Smaller values of λ were tested with to determine whether there was a pattern. The result for smaller values of λ were overall higher accuracies, with the highest accuracy for higher sparsity levels being 34.9%. However, the issue huge accuracy degradation persisted, indicating that this may have been an issue in implementation. Since the degree of accuracy was proportional to λ, it suggested that the regularisation function worked properly. Therefore, the issue pointed towards the rounding function. To test further, and since the total sparsity level was much less than 1%, the next test involved 7 decimal places. An important note was that although the maximum sparsity level reached was much less than 1%, sensitivity to rounding must be considered, as accuracy dropped rapidly.

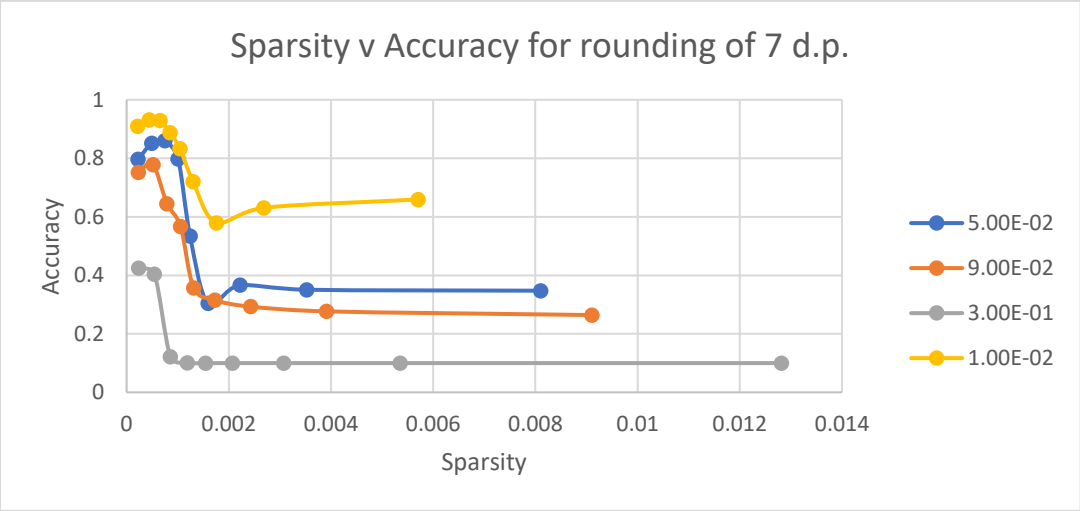


Figure 10: Graph showing sparsity level with corresponding accuracy per lambda for precision 7 d.p.

As shown in figure 10, the sparsity level became greater (between 79- 115.5 x) with 7 d.p. rounding, whilst having a maximum drop in accuracy of 0.13%, meaning that smaller λ could be used, which would have less effect on accuracy, whilst also creating larger amounts of sparse connections. This meant was that for the higher values of accuracy, the goal of at least 51% sparsity was more achievable. Examining the graph showed that although that was the maximum accuracy reached, unlike for other λ, where the accuracy had levelled to a certain amount for increasing sparsity, for λ = 0.01, the accuracy was incrementing, meaning that the model was yet to reach maximum accuracy, which would require more training epochs.

Understanding further about pytorch provided an insight to what led to the error. When performing rounding, the weights underwent copying whilst setting gradient flag to be false, meaning all calculation required for gradient descent was not copied back to the weight tensors, which affected the next weight parameters negatively, leading to higher loss values. The code was modified to allow gradient copy. Additionally, sparsity levels were increased by rounding to 4 decimal places and measured.

To further test how the model's accuracy varied with sparsity level, as well as finding optimal hyperparameter values, a test was done which involved investigating how the pruning affected the model for λ= 1e-2. The final accuracies were collected after 1000 training epochs with corresponding sparsity levels for different rounding of parameters. The rounding was done after every 100 training steps, similar to the previous sections.

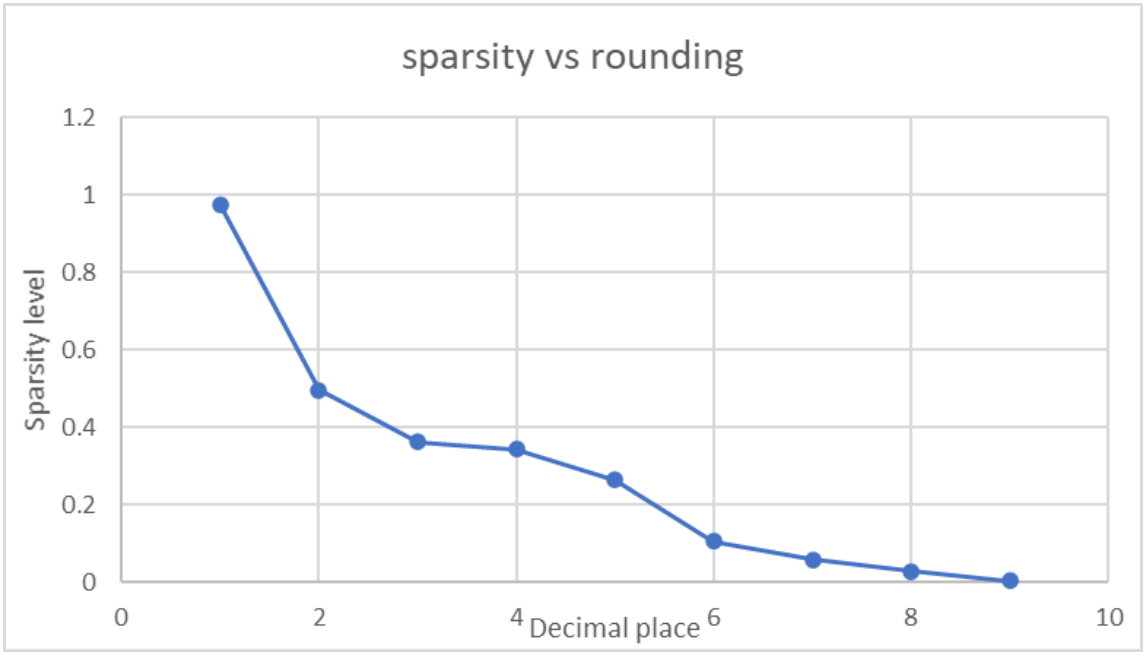


Figure 11: Final sparsity level achieved after 10 validation epochs per decimal place precision.

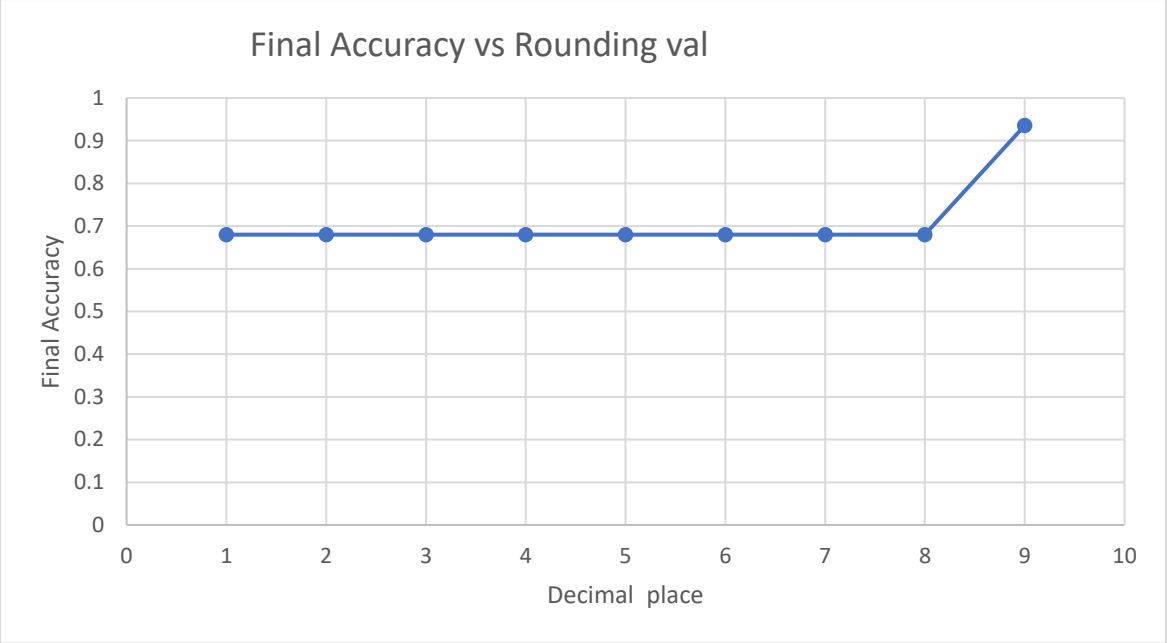


Figure 12: Final accuracy achieved after 10 validation epoch per decimal place precision.

Figure 10 showed that although the weight parameter precision decreased, the accuracy of the model was not affected to a significant level. However, there was a significant drop in accuracy by 25% when transitioning form 9 decimal places to 8 decimal places. The main conclusion drawn from these measurements was that although the sparsity level was increased from 2 % to 97.5 %, the final accuracy after 1000 training epochs remained the same, hence the rounding hyperparameter was set to 1 for maximum sparsity level.

Although sufficient sparsity level was achieved, the model had not been compressed yet since the parameters were still expressed in dense tensors, hence the same computational power was still required for the zero elements in the tensors. [12] had stated the use of compressed matrix storage (coo) after undergoing pruning and quantisation of model to store the sparse parameters. Although this was possible with pytorch, the issue arose as the model was built upon nn. linear [33], which only allowed dense multiplication of tensors.

There were two possible solutions which would allow the use of sparse matrix multiplication, torch. sparse.admm[25] and sparselinear[26]. With torch. sparse.admm, the design required to explicitly specify the sparse weight tensor, as well as the input tensor and bias tensor to perform forward propagation. In contrast, with sparselinear, the existing arguments of the current model could be used for the function. Furthermore, the method performed operations in coo format. The added benefit of sparselinear was that the user could define the sparse connections whilst also allowing random pruning. This meant that both penalty-based pruning could be implemented and compared with randomised pruning in terms of latency. Even though both allow efficient sparse dense matrix multiplication, sparselinear was more practical for this implementation, due to its similarity to nn. Linear.

4.4 Quantization

Quantization was considered to reduce further compute requirements of the model such that the model could run on the Nano. This the model from fp32 to fp16. This would half the compute requirements which in turn would speedup calculations. Since this was implemented for only validation, post-training quantization was required. Hence, the model was trained in single point precision, and when loading both the parameters and the inputs, they were converted to fp16. Furthermore, more compression was possible as pytorch allowed for the use of int8, which would reduce the model by 4x [34]. However, the feature was only to be used for CPU run. Additionally, the Jetson Nano does not support int8 precision.

Initially, quantization to fp16 was done. Due to pytorch available packages, only required to implement torch. half() when loading the parameters and loading the input. Initially, both training and validation was tested with fp16, with maximum accuracy resulting in 93%, whilst GPU usage reduced to about 4.39 GB. Running only validation, the total memory usage was 3.935 GB (),

4.5 implementation of Sparselinear

Initially, the model was initially run on the Jetson Nano, due to complications of downloading and running on the alpha cluster, untrained and initialised at 99% sparsity level and quantized to fp16, with batch size of 32, to test whether the model ran on the Jetson Nano successfully. Figure 11 shows how the steps taken to implement sparselinear.

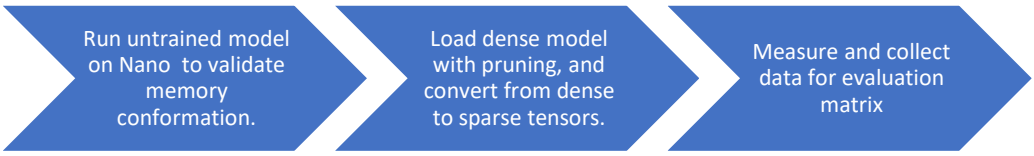


Figure 13: Plan for implementing sparselinear on the Nano

Even though the number of parameters reduced to around 1.3M, validation run was killed by the OS, due to insufficient ram. Hence, to increment memory usage on the Nano, Swapfile was then considered to be implemented. The main disadvantage of using swap file is that it sends ram storage to read only memory storage temporarily, hence the validation time period increments due to allocating and retrieving temporary pages. However, it was mainly considered as a immediate implementation solution to test whether the model would run.

A swap file of 3GB was set using [35]. The result was that there was still insufficient memory to run forward propagation of the model. The next step was to test with different evaluation batch sizes. Reducing batch size would reduce throughput, hence less processing would be required per batch.

The batch size was set to 3 such that the model would run. Although, around 45% of the validation was completed, the process ultimately ended due to runtime error. [36] provided an insight to the root of the error, indicating at the fact that the kernel of the device required more than 5 seconds, leading to a watchdog timeout. The crash along with the time elapsed led to more analysis requirement of the sparselinear method.

Testing was done on the functionality of sparselinear to understand why low number of parameters still resulted in huge latency and be able to satisfy the memory constraint, the first test was done to measure how changing sparsity level of parameters tensors affected the time elapsed to do forward propagation. This test implementation involved following the demonstration provided in [26], but the test compared sparsity levels and time elapsed. The time taken and GPU usage were collected for each sparsity level. Figure 14 shows the result when a large and wide network was selected (10,000 in features, 100 out). As expected, with increased sparsity level, time elapsed to perform forward propagation decreased linearly. Furthermore, with increased sparsity, the computational requirements decreased, with a massive difference between 90% and 99% sparsity in terms of GPU usage, measured as percentage of Collab allocated GPU.

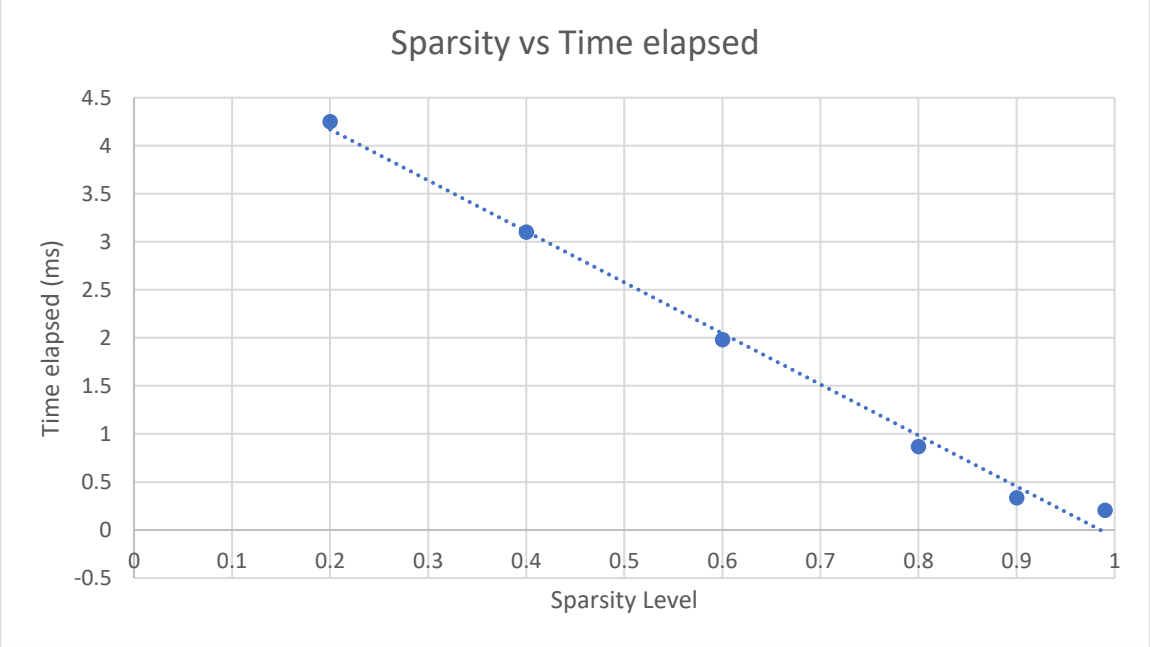


Figure 14: Relation between sparsity level and time elapsed for sparselinear connection

Furthermore, the test was also performed on a smaller network to observe whether the relationship between elapsed time and sparsity level upheld, along with difference in GPU usage. The second test setup included: in features = 100, out = 10. The results were captured

in table 2. The values confirm that for much smaller models, the time elapsed did not vary much, with the standard deviation of the mean time elapsed being 2.2E-3 ms. The GPU usage varied more from 10 % to 18%, which was a considerable difference. However, there was no pattern visible between GPU used and sparsity level, and seemed to fluctuate between that range for all sparsity level. Therefore, the logical conclusion from this test was that as the untrained model already was running at maximum available ram, keeping the highest level of sparsity possible was needed such that inference time would be minimum and the computational requirements are met.

in features	out features	sparsity	gpu usage(run1)	gpu usage(run2)	gpu usage(run3)	GPU Usage range	GPU usage average(%)	time elapsed	time elaps	time elapsed(ru	average time elapsed(ms)
100	10	0.2	11	17	10	7	12.66666667	0.198	0.201	0.2	0.199666667
100	10	0.4	16	16	17	1	16.33333333	0.2	0.201	0.194	0.198333333
100	10	0.6	16	12	16	4	14.66666667	0.205	0.208	0.2	0.204333333
100	10	0.8	16	15	16	1	15.66666667	0.205	0.203	0.2	0.202666667
100	10	0.9	13	14	14	1	13.66666667	0.201	0.204	0.201	0.202
100	10	0.99	18	18	16	2	17.33333333	0.202	0.198	0.208	0.202666667

Table 2: Showing calculated gpu usage and time elapsed per sparsity level

The second test involved comparing a dense linear network and a sparse network where the number of non-zero weights was equal to the total number of weights of the dense. The time elapsed and memory utilisation were again measured and compared. The results proved that although the sparse linear did use equal or less GPU, the elapsed time was longer, with the sparse connection requiring a maximum of 3.5 x more time.

			sparselinear			nn.linear	
number of elements	15	sparsity	time elapsed(microsec)	gpu usage		time elapsed(microsec)	gpu usage
		99	209	21	run1	60.4	21
		90	206	20	run2	59.8	20
		80	213	17	run3	60.4	19
		40	197	21	average	60.2	
	10000	99	207	18	run1	60.6	42
		90	205	20	run2	60.9	43
		80	211	37	run3	60.4	38
		60	203	37	average	60.63333333	

Table 3: Showing collected data of nn.linear and sparselinear

Loading and running the dense checkpoint into the sparse model was not possible, as it caused runtime errors. To fix the bugs would require to re-design code in modelling.py to allow allocation of the sparse weights with respect to their positions and connections. Another solution would be to train the sparse model without fine tuning, and loading the sparse weights. However, considering the error produced running validation with 99% sparsity, rather than training sparse model, it was considered more efficient to train and run a smaller model. Hence the model was reduced via static methods.

4.6 Static model reduction

The model was reduced via static method to the s/16 model. Unlike the b/16, it did not contain any pretrained parameters, hence it was first required to undergo training, rather than fine tuning. The configurations were modified to fit the model description.

Initially, the model was trained for 100 training epochs, with validation done after every training epoch. This was done such that there would be more updates on the estimate of the

accuracy of the model, so that more data would be collected for grasping out how rapidly does the model improve with given hyperparameters. The main hyperparameter changed was the learning rate, which affected how much gradient descent would the loss function undergo. With low learning rate, the model would take longer time to reach optimal solution, whereas with a very high learning rate, the descent would overshoot and the loss would increase. Table 4 shows the results for different learning rates. The optimal learning rate was chosen to be 1.10E-01.

learning rate	best accuracy
3.00E-02	0.3187
5.00E-02	0.3357
7.00E-02	0.3377
9.00E-02	0.3375
1.10E-01	0.3415
1.30E-01	0.3389
3.00E-01	0.3319

Table 4: Learning rate and corresponding accuracy

Once the hyperparameters were tuned such that the training efficiency was maximised, the number of training epochs was increased to 7500 training epochs, with validation run after every 100 epochs. The result is seen in graph 15.

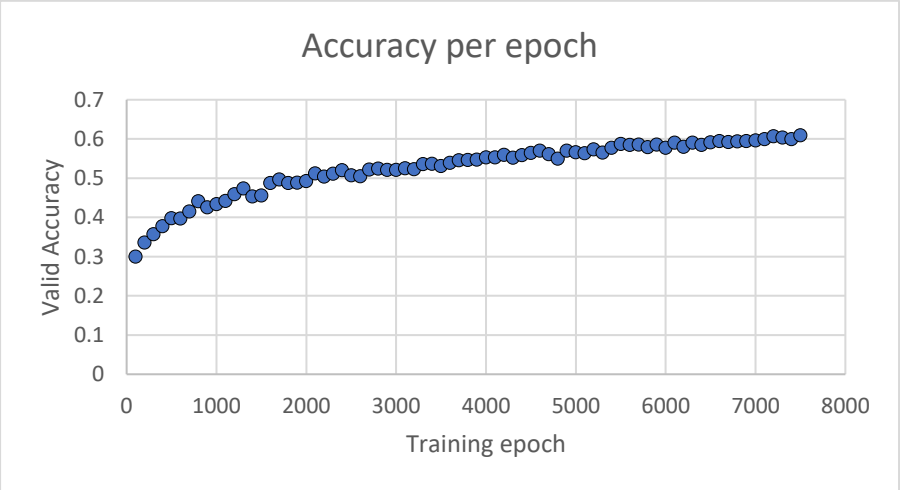


Figure 15: Valid accuracy of the s/16 model with image size 224 for 7500 training epochs

The maximum accuracy measured was 60.9%. That meant the accuracy difference from the fine-tuned B16 model by 34.1%. Since the model dropped by a significant amount, accuracy was required to be increased. Incrementing number of training epochs could increase the accuracy, but the change in accuracy needed to be considered for effectiveness. If the model was reaching optimal accuracy, incrementing training epochs would not have much effect. Hence, the gradient of the valid loss was measured and interpreted to make sure the change in loss was sufficient.

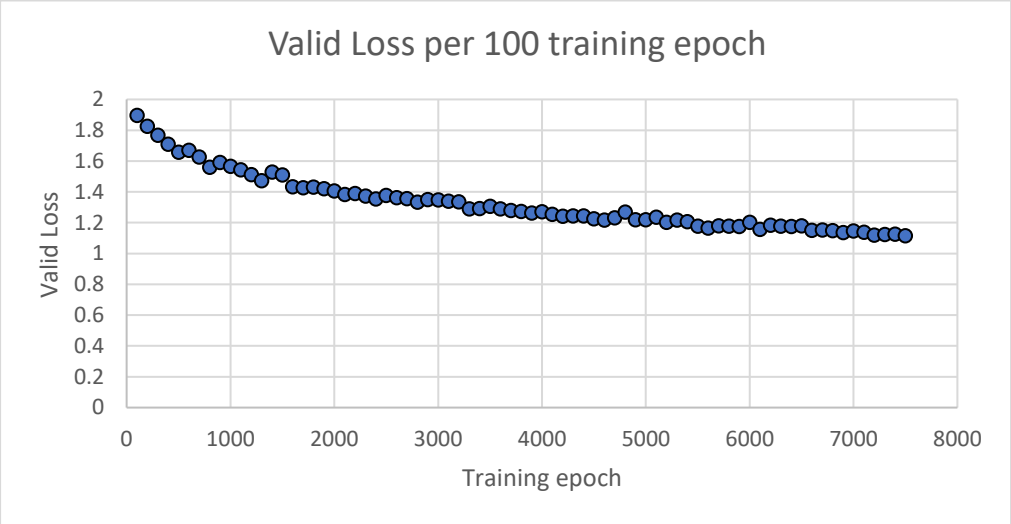


Figure 16: Valid loss of the model for image size 224 for 7500 training epochs

The gradients were measured between 5 consecutive points, as well as the change in gradient, which is displayed in table 5. As viewed the magnitude change in gradient decreased from 3.5E-2 to 2.0E-3, which was an order of 10 difference. Hence incrementing training epoch would only slightly increase the accuracy, not enough to reach above 90%.

points	Gradient	change in gradient
1_5	-0.05939	NA
6_10	-0.02417	-0.035221
11_15	-0.00554	-0.018634
16_20	-0.00605	0.000515
21_25	-0.00485	-0.001199
26_30	-0.00369	-0.001168
31_35	-0.01094	0.007253
36_40	-0.00568	-0.005256
41_45	-0.00589	0.000203
46_50	-0.00065	-0.005231
51_55	-0.01095	0.010298
56_60	0.007229	-0.018182
61_65	0.003971	0.003258
66_70	-0.00201	0.005985
71_75	-0.00402	0.002005

Table 5: Showing the gradient and change in gradient measured of loss

The B/16 model loaded the weights pretrained on ImageNet, where the dimension was 224 x 224 x3, meaning the input images were resized to match those specifications. However, as s/16 was being trained on CIFAR10 dataset, resizing was not needed. Therefore, the model was also trained for smaller image size. As the image size decreased, yet the patch sizes remained as 16 x16, the number of patches reduced considerably.

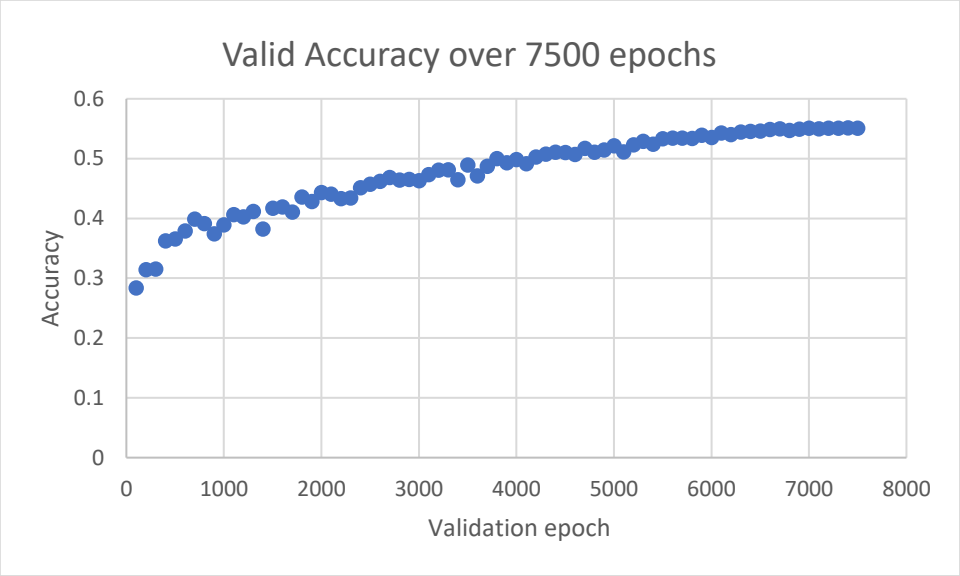


Figure 17: Valid accuracy for image size 32

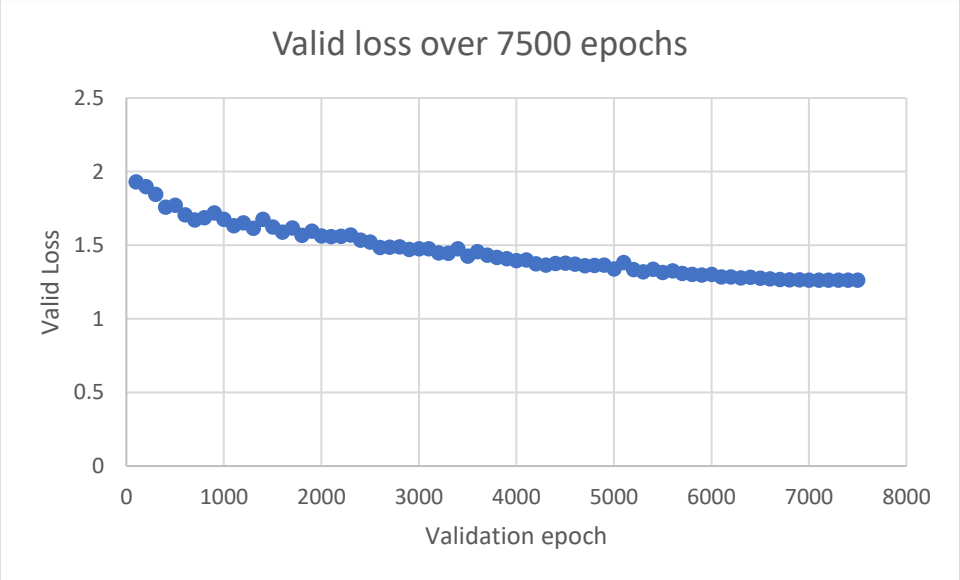


Figure 18: Valid loss for image size 32

The maximum accuracy achieved was 55.1%, which was 5.8% less than with image size 224. The change in gradient was again measured as in the previous test to verify if optimal accuracy was found. The table shows the result of gradient and each change of gradient. The change in gradient became a magnitude of order 10 less, being 3.9% of initial change. However, as the

change of gradient was not zero, optimal solution was not found. Similar to 224 model, more training steps would be required, but the ratio of gradient to the loss suggested that it would not provide the model with 90%+ accuracy.

gradient	change in gradient
-0.045703	NA
-0.000988	0.044715
0.000703	0.001691
-0.00717	-0.007873
-0.009989	-0.002819
-0.003085	0.006904
-0.00746	-0.004375
-0.014452	-0.006992
-0.00435	0.010102
-0.005902	-0.001552
-0.009609	-0.003707
-0.00565	0.003959
-0.002237	0.003413
0.000206	0.002443
-0.001539	-0.001745

Table 6: Change in gradient and gradient for loss

Apart from accuracy, the memory usage of the model was also measured to validate the model conformed to limitations, and further compression was not required, with memory used being 3.85 GB(appendix E).

4.7 CNN implementation

The pruned CNN model was retrieved from [38], which contained various implementations of pruning. Initially, Norm 1 pruning was selected for resnet56, as the number of parameters and FLOPs stated were closest to the mixer model. Accuracy of 93 % was achieved with the pruned model, and FLOPs measured was 0.22GMAC. However, the total memory usage measured shown in appendix F was larger than 4GB, hence swapfile was implemented. However, the OS killed the run. Therefore, it could not be implemented in time.

Chapter 5: Results & Analysis

When measuring evaluation matrices, the methods for measuring were first considered. The code had already contained method for the measurement of accuracy and the total number of parameters Hence, latency and FLOPs were required to be calculated. As total validation time elapsed was measured, $Latency(ms) = \frac{Total\ time\ elpased}{total\ number\ of\ batches \times batch\ size} \times 100$.

(Equation 8: Latency calculation using variables measured)

Each activation layer and transformation had different measurements of FLOPs, so the package [38] was used due to its feature of measuring GELU layer. The measurement was performed in GMACs, which is half the measurement of GFLOPs.

In terms of measurement consideration, the effect of the hyperparameters on the evaluation matrices was further looked into. The main hyperparameters considered were floating point precision, evaluation batch size, and image size. Precision was considered as it reduced memory usage by half, hence could impact validation latency, whilst also having an impact on accuracy. Batch sizes affect the throughput of the architecture, with higher batch sizes giving higher throughputs, and finally image size varies each patch size, hence the number of inputs per mlp layer also varies, which affects all evaluation matrices.

Initially, due to requirement match, the b/16 model was run using half-precision. However, as observed in yyy, this caused a watchdog timeout. Hence, the smaller models were considered. The results in table 7 shows the outcome when the s/16 model was loaded for fp32, when the images were resized to 3 x 224 x 224. An important note was that for batch size 32, the range of time elapsed was 01:25, which was noticeable fluctuation. This was most likely due to inconsideration of the performance variations due to high temperature. For 64 batch size, the range was 30 seconds, which was insignificant compared to elapsed time (4%). When measuring latency, increasing batch size reduce latency, as expected.

s/16 — image size = 224 — fp 32

batch size	accuracy	time1	accuracy 2	time 2	accuracy 3	time 3	average accuracy	average time(mm:ss)	latency(ms)	FLOPs(GMACs)	N. of Params. (Millions)	Range
32	0.6596	11:54	0.6596	10:41	0.6596	10:29	0.6596	11:01	7.430305755	3.781	18	01:25
64	0.6596	10:55	0.6596	11:02	0.6596	11:25	0.6596	11:07	6.68	3.781	18	00:30
128	Not enough Ram									3.781	18	

Table 7: Result collected for s/16 image -224

Table 8 shows for fp16.The comparison between fp32 and fp16 showed that although the model size decreased, the change in latency was not affected, which was not expected. Further investigation would be required.

s/16 — image size = 224 — fp 16

batch size	accuracy	time1	accuracy 2	time 2	accuracy 3	time 3	average accuracy	average time(mm:ss)	latency(ms)	FLOPs(GMACs)	N. of Params. (Millions)	Range
32	0.6594	10:57	0.6594	11:12	0.6594	11:22	0.6594	11:10	7.53	3.781	18	00:25
64	0.6594	11:11	0.6594	10:59	0.6594	11:10	0.6594	11:06	6.63	3.781	18	00:11
128	NoT enough ram									3.781	18	

Table 8: Results collected for s/16-image 224 fp16

The model without resized image input was run on the Nano, with results shown in table 9. The number of parameters decreased by 0.8M, which in turn decreased the FLOPs massively. The latency also decreased by a magnitude of order 10. This seemed to suggest that lower FLOPs resulted in lower latency. Additionally, the range also reduced form around 15-30 seconds to 5 seconds, although the impact doubled to a maximum of around 8.8 %.

s/16

image size 32

fp 32

batch size	accuracy	time1	accuracy 2	time 2	accuracy	time 3	average accuracy	average time(mm:ss)	latency(ms)	FLOPs(GMACs)	N. of Params. (Millions)	Range
32	0.5513	01:03	0.5513	01:03	0.5513	01:08	0.5513	01:04	0.72	0.078	17.2	00:05
64	0.5513	01:10	0.5513	01:09	0.5513	01:07	0.5513	01:08	0.68	0.078	17.2	00:03
128	0.5513	01:12	0.5513	01:06	0.5513	01:07	0.5513	01:08	0.66	0.078	17.2	00:06

Table 9: Results collected for s/16 image 32

To further test the effect of FLOPs on latency, the model was scaled by 4:5, such that it was larger than the s/16 model, but smaller than the b/16 model. The training of the new model followed the procedure as the s/16 model, with hyperparameter tuning and training for both image sizes. The testing also followed a same procedure as s/16, without using half-precision. The results were collected in the table 10. The larger model provided a more accurate result, at the cost of higher latency. This caused problem for input image size 224, as when the batch size was increased to 64/128, more throughput was required, which the Nano could not handle and provided a launch timeout runtime error.

m/16

image =32

fp 32

batch size	accuracy 1	time1	accuracy 2	time 2	accuracy	time 3	average accuracy	average time(mm:ss)	latency(ms)	FLOPs(GMACs)	N. of Params. (Millions)	Range
32	0.5783	01:45	0.5783	01:43	0.5783	01:40	0.5783	01:42	1.15	0.152	33.4	00:05
64	0.5783	01:48	0.5783	01:40	0.5783	01:46	0.5783	01:44	1.04	0.152	33.4	00:08
128	0.5783	01:52	0.5783	01:52	0.5783	01:43	0.5783	01:49	1.08	0.152	33.4	00:09

M/16

image =224

fp 32

batch size	accuracy 1	time1	accuracy 2	time 2	accuracy	time 3	average accuracy	average time(mm:ss)	latency(ms)	FLOPs(GMACs)	N. of Params.	Range
32	0.7004	19:05	0.7004	19:04	0.7004	19:03	0.7004	19:04	12.86	7.329	34.6M	00:02
64	Launch timeout											
128	Launch timeout											

Table 10: Results collected for new model with both image size

The results suggested a positive correlation between FLOPs and the latency of the model, so to test, the results were plotted on a scatter graph per batch size with FLOPs being the independent variable. However, due to lack of enough plot points and distance between each plot point, it was hard to determine the relationship between the two variables. Hence, a log-linear graph was then plotted, as seen in figure 19, for maximum throughput for each FLOPs. The trend of the graph displayed was that the latency increased with FLOPs in a non-linear method. However, due to lack of further data, it was hard to determine the manner in which the latency increases, i.e., whether it is exponential.

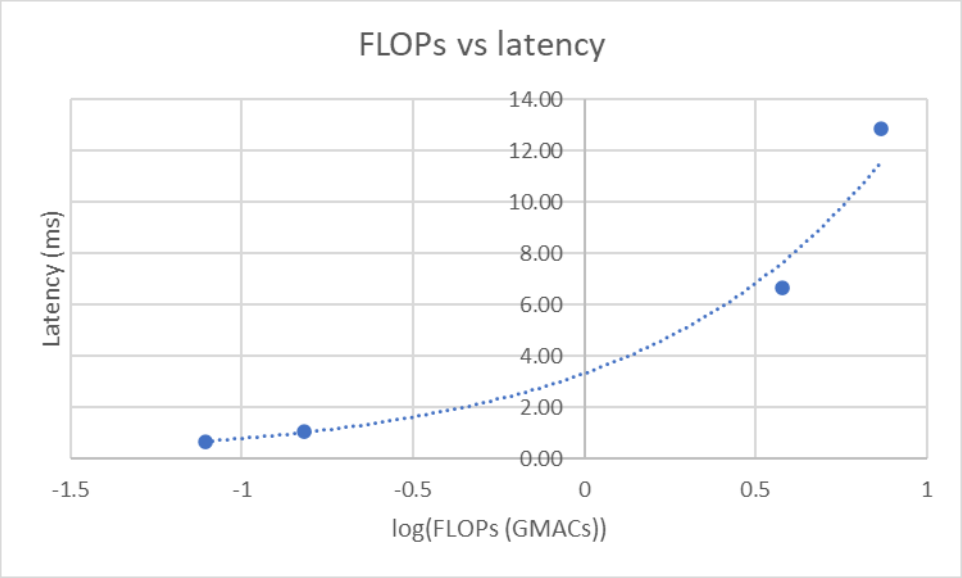


Figure 19: Showing the relation between FLOPs and latency

Chapter 6 Conclusions and Evaluation

7.1 Conclusion

In conclusion, the results obtained measured the test elements of the evaluation matrix for the MLP-Mixer module. The results obtained from such result clearly displayed a positive correlation between the FLOPs and latency of the model. However, a numerical equation was not estimated, due to the lack of data points collected. Further reduction of different sizes would be required. Furthermore, the test accuracy was impacted by the image size, and hence number of patches available, indicating for the same amount of training epochs, the model with greater number of patches provided a greater validation accuracy.

7.1 Evaluation

There were four main objectives of this project, detailed in the introduction. The first objective, which was to use GPU accelerator to run the MLP -mixer model was successfully accomplished. The model chosen was the B/16 model, and it was finetuned to the CIFAR-10 dataset. The accuracy achieved was comparable with the results from the original paper [12]. Additionally, the memory requirements from the measured and compared to the availability of the Jetson Nano.

The second objective required to undergo sufficient model compression such that the model would run on the Jetson Nano successfully. Although two main methods were considered, as researched in 2.2, only one was tested, with various methods tried to implement it. However, the implementation was unsuccessful, largely due to fact that the model architecture was designed for dense model connections. Although, conversion from dense to sparse was looked into and intended, it was still not possible. Furthermore, even running untrained sparse model with randomly initialised weights resulted in runtime error.

The fourth objective was considered next. As the B/16 model was too large to run on the Jetson Nano, the target then became to manually reduce the model. As described in the introduction, this involved either dynamic or static methods. Static method was considered, where the size and architecture of the model was scaled down to S/16 model, as well as an in- between model, and so number of floating-point operations decreased. This was somewhat successful as although the model was not trained to competitive accuracy level, it ran validation on the Jetson Nano for a reasonable time period.

Finally, the third objective involved testing and comparison of the compressed/ reduced CNN, Vit and MLP models, against the evaluation matrix. The CNN model was not implemented on the Nano, although there was an attempt to do so. However, the model was still too big, and using swapfile did not provide solution as desired. Finally, the Vit model was not implemented as it required further research and time to compress the model. The outcomes were compared and discussed in chapter 5.

7.2 Future Works

The main task remaining would be to implement CNN and Vit architectures into the Jetson Nano, measure and compare with the other two. This could be done via researching more into the Vit pruning [28], or similar to the Mixer model, done via static reduction.

The maximum accuracy achieved of the reduced MLP model was 70%, which made the model not industry applicable. Hence, improvement of the accuracy is required. Although it can be trained for more training epochs, one possible method that would be interesting to test would be knowledge distillation. As the b/16 model has been trained for 95% accuracy, it can be used as the parent model, from which information can be extracted and applied to both the smaller models. Additionally, the efficacy of knowledge distillation can be investigated for the MLP-mixer model.

Another issue faced when doing compression was that although GPU usage was reduced, the CPU usage was still very high and not reduced. Hence, reducing the CPU usage would be ideal, which may in fact allow the B/16 model to be run and tested on the Nano.

7.3 Project Management

7.3.1 Time management

In appendix G, both Gannt charts are visible, tracking my progress. As seen, initially, the progress was on track, but when it came to designing and implementing pruned model, it slowed down the whole process. Factors such as technical difficulties, coronavirus had slightly affected this, but the issue found was figuring out what approach to take, as there was a huge gap between theoretical mathematical background and understanding computational calculations of sparse matrices and methods of reducing hardware costs of calculating such matrices. Additionally, once I was behind schedule, methods such as critical path analysis should have been considered. This meant that towards the end, there was more pressure towards the end of the time period.

7.3.2 Risk Evaluation:

Risk	Impact	Mitigation
Burning due to overheating	High	Remove power after each run and if over 75 degrees, unplug
No models being implemented	High -medium	Focusing on main goals to achieve- i.e., MLP mixer
Corona-virus	Medium-low	Plan ahead if you work is possible to do

7.4 Self Reflection

As an Electronic Engineering student with little background in artificial intelligence, I learnt a lot about basic AI principles, as well as more specific concepts such as CNN and Vit models. I strengthened my python skills, whilst also applying pytorch packages. Vital concepts in applied AI such as hyperparameter tuning on comparison factors of a model such as accuracy, were experimented with. With regards to more hardware related knowledge, I got to explore methods as to how to deal with hardware limitations, such as using swap file, as well as exploring methods to deal with hardware issues related to effective use of sparse matrices. However,

References

[1] X. Zhang and S. Xu, "Research on Image Processing Technology of Computer Vision Algorithm," 2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL), 2020, pp. 122- 124, doi: 10.1109/CVIDL51233.2020.00030.

[2] M. Kunaver and J. F. Tasic, "Image feature extraction - an overview," EUROCON 2005 - The International Conference on "Computer as a Tool", 2005, pp. 183-186, doi: 10.1109/EURCON.2005.1629889.

[3] X. Chen, S. Wei, S. Xing and K. Jia, "An effective image shape feature detection and description method," 2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC), 2011, pp. 1957-1960, doi: 10.1109/MEC.2011.6025871.

[4] M. k. Alsmadi, K. B. Omar, S. A. Noah and I. Almarashdah, "Performance Comparison of Multi-layer Perceptron (Back Propagation, Delta Rule and Perceptron) algorithms in Neural Networks," 2009 IEEE International Advance Computing Conference, 2009, pp. 296-299, doi: 10.1109/IADCC.2009.4809024.

[5] J. Amrutha and A. S. Remya Ajai, "Performance analysis of Backpropagation Algorithm of Artificial Neural Networks in Verilog," 2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), 2018, pp. 1547-1550, doi: 10.1109/RTEICT42901.2018.9012614.

[6] K. O'Shea en R. Nash, "An Introduction to Convolutional Neural Networks", ArXiv e-prints, 11 2015.

[7] "Depth wise Separable Convolutional Neural Networks - GeeksforGeeks", GeeksforGeeks, 2019. [Online]. Available: https://www.geeksforgeeks.org/depth-wise-separable-convolutional-neural-networks/. [Accessed: 14- Dec- 2021].

[8] S. Wan, C. -Y. Hsu, J. Li and M. Zhao, "Depth-Wise Convolution with Attention Neural Network (DWA) for Pneumonia Detection," 2020 International Conference on Intelligent Computing, Automation and Systems (ICICAS), 2020, pp. 136-140, doi: 10.1109/ICICAS51530.2020.00035.

[9] M. A. Islam*, S. Jia*, en N. D. B. Bruce, "How much Position Information Do Convolutional Neural Networks Encode?", in International Conference on Learning Representations, 2020.

[10] A. Vaswani et al., "Attention is All you Need", in Advances in Neural Information Processing Systems, 2017, vol 30

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929v2

[12] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and

Alexey Dosovitskiy. Mlpmixer: An all-mlp architecture for vision. arXiv preprint arXiv:2105.01601, 2021.

[13] S. Han, H. Mao, en W. J. Dally, “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”, in 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.

[14] G. Hinton, O. Vinyals, en J. Dean, “Distilling the Knowledge in a Neural Network”, in NIPS Deep Learning and Representation Learning Workshop, 2015.

[15] R. Arshad, A. Saleem and D. Khan, "Performance comparison of Huffman Coding and Double Huffman Coding," 2016 Sixth International Conference on Innovative Computing Technology (INTECH), 2016, pp. 361-364, doi: 10.1109/INTECH.2016.7845058. 17

[16] “Jetson Nano,” *NVIDIA Developer*, 29-Mar-2021. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano. [Accessed: 03-May-2022].

[17] Andrew G. Howard, Menglong Zhu,Bo,Chen Dmitry,Kalenichenko,Weijun Wang,Tobias Weyand, Marco Andreetto,Hartwig Adam: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861v1

[18] H. Shu και H. Zhu, ‘Sensitivity Analysis of Deep Neural Networks’, *ArXiv*, τ. abs/1901.07152, 2019.

[19] E. Tartaglione, A. Bragagnolo, F. Odierna, A. Fiandrotti, και M. Grangetto, ‘SeReNe: Sensitivity based Regularization of Neurons for Structured Sparsity in Neural Networks’, IEEE transactions on neural networks and learning systems, τ. PP, 2021.

[20] A. Farahani, B. Pourshojae, K. Rasheed, και H. R. Arabnia, ‘A Concise Review of Transfer Learning’, CoRR, τ. abs/2104.02144, 2021.

[21] J. Kukacka, V. Golkov, και D. Cremers, ‘Regularization for Deep Learning: A Taxonomy’, CoRR, τ. abs/1710.10686, 2017.

[22] D. Hempston, “The Iridis Compute Cluster,” *The Iridis Compute Cluster | iSolutions | University of Southampton*. [Online]. Available: https://www.southampton.ac.uk/isolutions/staff/iridis.page. [Accessed: 02-May-2022].

[23] “Torch.nn.utils.prune.global_unstructured¶,” *torch.nn.utils.prune.global_unstructured - PyTorch 1.11.0 documentation*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.utils.prune.global_unstructured.html. [Accessed: 02-May-2022].

[24] A. Dong, A. Taghvaei and T. T. Georgiou, "Lasso formulation of the shortest path problem," 2020 59th IEEE Conference on Decision and Control (CDC), 2020, pp. 402-407, doi: 10.1109/CDC42340.2020.9303909.

[25] “Torch.sparse.addmm¶,” *torch.sparse.addmm - PyTorch 1.11.0 documentation*. [Online]. Available:

https://pytorch.org/docs/stable/generated/torch.sparse.addmm.html. [Accessed: 03-May-2022].

[26] hyeon95y, “HYEON95Y/sparselinear: A custom pytorch layer that is capable of implementing extremely wide and sparse linear layers efficiently,” *GitHub*. [Online]. Available: https://github.com/hyeon95y/SparseLinear.git. [Accessed: 03-May-2022].

[27] “Ubuntu documentation,” SwapFaq - Community Help Wiki. [Online]. Available: https://help.ubuntu.com/community/SwapFaq. [Accessed: 03-May-2022].

[28] M. Zhu, K. Han, Y. Tang, και Y. Wang, ‘Visual Transformer Pruning’, CoRR, τ. abs/2104.08500, 2021.

[29] Google-Research, “Google-research/vision_transformer,” GitHub. [Online]. Available: https://github.com/google-research/vision_transformer.git. [Accessed: 03-May-2022].

[30] Jeonsworld, “Jeonsworld/MLP-mixer-pytorch: Pytorch reimplementation of the mixer (MLP-mixer: An all-MLP architecture for vision),” GitHub. [Online]. Available: https://github.com/jeonsworld/MLP-Mixer-Pytorch.git. [Accessed: 03-May-2022].

[31] P. Micikevicius κ.ά., ‘Mixed Precision Training’, CoRR, τ. abs/1710.03740, 2017.

[32] Wasi AhmadWasi Ahmad 31.2k3030 gold badges100100 silver badges154154 bronze badges, Szymon MaszkeSzymon Maszke 19k22 gold badges3030 silver badges7171 bronze badges, devil in the detaildevil in the detail 2, KashyapKashyap 5, iacobiacob 13.6k55 gold badges4949 silver badges9090 bronze badges, Sherif AliSherif Ali 33722 silver badges33 bronze badges, prostiprosti 34.7k99 gold badges161161 silver badges139139 bronze badges, and oukohououkohou 37133 silver badges99 bronze badges, “Adding L1/L2 regularization in pytorch?,” Stack Overflow, 01-Dec-1964. [Online]. Available: https://stackoverflow.com/questions/42704283/adding-l1-l2-regularization-in-pytorch. [Accessed: 02-May-2022].

[33] “Linear,” Linear - PyTorch 1.11.0 documentation. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html. [Accessed: 03-May-2022].

[34] “Quantization,” Quantization - PyTorch 1.11.0 documentation. [Online]. Available: https://pytorch.org/docs/stable/quantization.html. [Accessed: 03-May-2022].

[35] JetsonHacksNano, “Jetsonhacksnano/installswapfile: Install a swap file on the Nvidia Jetson Nano developer kit. this should help with memory pressure issues.,” GitHub. [Online]. Available: https://github.com/JetsonHacksNano/installSwapfile.git. [Accessed: 02-May-2022].

[36] Person, “Jetson Nano: Runtimeerror: Cuda error: The launch timed out and was terminated,” NVIDIA Developer Forums, 20-Dec-2021. [Online]. Available: https://forums.developer.nvidia.com/t/jetson-nano-runtimeerror-cuda-error-the-launch-timed-out-and-was-terminated/198530. [Accessed: 02-May-2022].

[37] Sovrasov, “Sovrasov/flops-counter.pytorch: Flops counter for convolutional networks in Pytorch Framework,” GitHub. [Online]. Available: https://github.com/sovrasov/flops-counter.pytorch.git. [Accessed: 02-May-2022].

[38] Eric-mingjie, “Eric-Mingjie/rethinking-network-pruning: Rethinking the value of network pruning (pytorch) (ICLR 2019),” GitHub. [Online]. Available: https://github.com/Eric-mingjie/rethinking-network-pruning. [Accessed: 02-May-2022].

Appendix A: Project Brief

Problem

In modern deep learning models for computer vision tasks, there are two main methods, including Convolutional Neural Networks (CNNs) and Vision Transformer (ViT) [2]. To learn features in different spatial locations and channels of input images, CNNs use multiple building blocks (e.g., convolutional layers, pooling layers, and fully connected layers). ViT has become a competitive alternative to CNN that utilizes multiple attention mechanisms to weigh the importance of each part of the input data differently. To achieve high accuracy, however, there is an issue involving CNN and ViT models. Both models require a high computational cost and large memory requirements due to their complex architecture, which takes a huge toll, especially on embedded computers with a lot more limited performance capability. Therefore, other models are required to be researched, tested, and compared with for vision on embedded. Currently, a simple alternative to both CNNs and ViT has been proposed by [1], coined 'MLP-Mixer', which does not use convolutions or self-attention. Instead, the architecture of MLP-Mixer is completely based on the multi-layer perceptron (MLP), which is repeatedly applied in the spatial location and feature channels [1]. MLP-Mixer with a simple structure can achieve competitive performance and higher throughput than CNN and ViT, which the architecture is hardware-friendly, especially for embedded devices. As a result, it is worth considering deploying MLP-Mixer on embedded platforms for the model inference process.

Goal

The main goal of this project is to successfully run the MLP-mixer network on an embedded computer and measure the performance capabilities of the computer whilst running the MLP-Mixer. An initial model that will be used will be trained on the desktop before being implemented into the embedded computer (e.g., Jetson Nano). The two datasets required will contain both training and testing data. The hardware specifications of the Jetson Nano include 4GB memory, 16GB eMMC storage, quad-core ARM Cortex CPU and Maxwell architecture with 128 cores GPU [3]. The datasets used needs to contain a large amount of training data to improve upon the accuracy. Test data needs to be carefully selected such that it is unique from the training. Furthermore, due to the large training dataset and the model both being downloaded into the device, storage limitation issues arise. Therefore, the model will then be used to undergo further modifications required to satisfy the hardware properties, which involves model compression through either pruning (removing low-effect weights), which affects the number of parameters element or quantization (which affects inference accuracy). Once satisfaction is received, the model will be implemented and compared with the CNN model MobileNet [4] and attention-based model ViT-L/16 [2], based on the evaluation matrix. The evaluation matrix involves four parameters, including test accuracy that how accurate the model is at classifying, test latency that the time is taken to achieve a classification of the test data and FLOPs and the number of model parameters. Additionally, observation of the relationship between FLOPs and latency of MLPs will be done (e.g., whether smaller FLOPs can reduce the latency of MLP on Jetson). Finally, the last target will be to find a solution for controlling the FLOPs or parameters if the MLP-Mixer model is over resource limitations on Jetson by static solution and/or dynamic solution?

Appendix B: Project Archive

File Name	Purpose
Edited MLP-Mixer folder	The edited mlp-mixer model, with implementations for sparsity
Sparselinear.py	Testing sparselinear package
L1-norm-pruning	Pruned resnet-56 and for running validation
Pruning_test.py	Design and implementing norm 1 pruning

Appendix C:

```
5/03/2022 02:21:21 - INFO - __main__ - Validation Results
5/03/2022 02:21:21 - INFO - __main__ - Global Steps: 100
5/03/2022 02:21:21 - INFO - __main__ - Valid Loss: 0.28817
5/03/2022 02:21:21 - INFO - __main__ - Valid Accuracy: 0.90890
|===========================================================================|
|                  PyTorch CUDA memory summary, device ID 0                 |
|---------------------------------------------------------------------------|
|           CUDA OOMs: 0            |       cudaMalloc retries: 0           |
|===========================================================================|
|        Metric         | Cur Usage  | Peak Usage | Tot Alloc  | Tot Freed  |
|---------------------------------------------------------------------------|
| Allocated memory      |  240348 KB |    1109 MB |    1366 GB |    1366 GB |
|       from large pool |  232896 KB |    1101 MB |    1366 GB |    1366 GB |
|       from small pool |    7452 KB |       7 MB |       0 GB |       0 GB |
|---------------------------------------------------------------------------|
| Active memory         |  240348 KB |    1109 MB |    1366 GB |    1366 GB |
|       from large pool |  232896 KB |    1101 MB |    1366 GB |    1366 GB |
|       from small pool |    7452 KB |       7 MB |       0 GB |       0 GB |
|---------------------------------------------------------------------------|
| GPU reserved memory   |    1442 MB |    1442 MB |    1442 MB |       0 B  |
|       from large pool |    1432 MB |    1432 MB |    1432 MB |       0 B  |
|       from small pool |      10 MB |      10 MB |      10 MB |       0 B  |
|---------------------------------------------------------------------------|
| Non-releasable memory |   34084 KB |  269285 KB |  139404 MB |  139371 MB |
|       from large pool |   33344 KB |  268544 KB |  138882 MB |  138849 MB |
|       from small pool |     740 KB |    6017 KB |     522 MB |     522 MB |
|---------------------------------------------------------------------------|
| Allocations           |     155    |     161    |   16574    |   16419    |
|       from large pool |      26    |      31    |   11742    |   11716    |
|       from small pool |     129    |     132    |    4832    |    4703    |
|---------------------------------------------------------------------------|
| Active allocs         |     155    |     161    |   16574    |   16419    |
|       from large pool |      26    |      31    |   11742    |   11716    |
|       from small pool |     129    |     132    |    4832    |    4703    |
|---------------------------------------------------------------------------|
| GPU reserved segments |      26    |      26    |      26    |       0    |
|       from large pool |      21    |      21    |      21    |       0    |
|       from small pool |       5    |       5    |       5    |       0    |
|---------------------------------------------------------------------------|
| Non-releasable allocs |      18    |      21    |    1792    |    1774    |
|       from large pool |      13    |      16    |    1527    |    1514    |
|       from small pool |       5    |       7    |     265    |     260    |
|---------------------------------------------------------------------------|
| Oversize allocations  |       0    |       0    |       0    |       0    |
|---------------------------------------------------------------------------|
```

```
Submitted batch job 1382433
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$ seff 1382433
Job ID: 1382433
Cluster: i5
User/Group: sm3g19/fp
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 2
CPU Utilized: 00:01:09
CPU Efficiency: 62.73% of 00:01:50 core-walltime
Job Wall-clock time: 00:00:55
Memory Utilized: 3.27 GB
Memory Efficiency: 30.97% of 10.55 GB
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$
```

Appendix D:

```
05/03/2022 02:53:05 - INFO - __main__ - Valid Accuracy: 0.90650
|===========================================================================|
|                  PyTorch CUDA memory summary, device ID 0                 |
|---------------------------------------------------------------------------|
|            CUDA OOMs: 0            |        cudaMalloc retries: 0         |
|===========================================================================|
|        Metric         | Cur Usage  | Peak Usage | Tot Alloc  | Tot Freed  |
|---------------------------------------------------------------------------|
| Allocated memory      |  130482 KB |  645781 KB |  716714 MB |  716586 MB |
|       from large pool |  126752 KB |  642048 KB |  716500 MB |  716377 MB |
|       from small pool |    3730 KB |    7449 KB |    213 MB |    209 MB |
|---------------------------------------------------------------------------|
| Active memory         |  130482 KB |  645781 KB |  716714 MB |  716586 MB |
|       from large pool |  126752 KB |  642048 KB |  716500 MB |  716377 MB |
|       from small pool |    3730 KB |    7449 KB |    213 MB |    209 MB |
|---------------------------------------------------------------------------|
| GPU reserved memory   |     996 MB |     996 MB |     996 MB |       0 B  |
|       from large pool |     988 MB |     988 MB |     988 MB |       0 B  |
|       from small pool |       8 MB |       8 MB |       8 MB |       0 B  |
|---------------------------------------------------------------------------|
| Non-releasable memory |   16974 KB |  189039 KB |  333486 MB |  333469 MB |
|       from large pool |   16608 KB |  188672 KB |  333266 MB |  333250 MB |
|       from small pool |     366 KB |    6017 KB |    219 MB |    219 MB |
|---------------------------------------------------------------------------|
| Allocations           |      156   |      162   |    16653   |    16497   |
|       from large pool |       27   |       32   |    11900   |    11873   |
|       from small pool |      129   |      132   |     4753   |     4624   |
|---------------------------------------------------------------------------|
| Active allocs         |      156   |      162   |    16653   |    16497   |
|       from large pool |       27   |       32   |    11900   |    11873   |
|       from small pool |      129   |      132   |     4753   |     4624   |
|---------------------------------------------------------------------------|
| GPU reserved segments |       25   |       25   |       25   |        0   |
|       from large pool |       21   |       21   |       21   |        0   |
|       from small pool |        4   |        4   |        4   |        0   |
|---------------------------------------------------------------------------|
| Non-releasable allocs |        9   |       18   |    10527   |    10518   |
|       from large pool |        6   |       14   |     8378   |     8372   |
|       from small pool |        3   |        6   |     2149   |     2146   |
|---------------------------------------------------------------------------|
| Oversize allocations  |        0   |        0   |        0   |        0   |
|---------------------------------------------------------------------------|
| Oversize GPU segments |        0   |        0   |        0   |        0   |
|===========================================================================|
```

```
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$ seff 1382532
Job ID: 1382532
Cluster: i5
User/Group: sm3g19/fp
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 2
CPU Utilized: 00:00:51
CPU Efficiency: 70.83% of 00:01:12 core-walltime
Job Wall-clock time: 00:00:36
Memory Utilized: 3.29 GB
Memory Efficiency: 31.23% of 10.55 GB
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$
```

Appendix E:

PyTorch CUDA memory summary, device ID 0

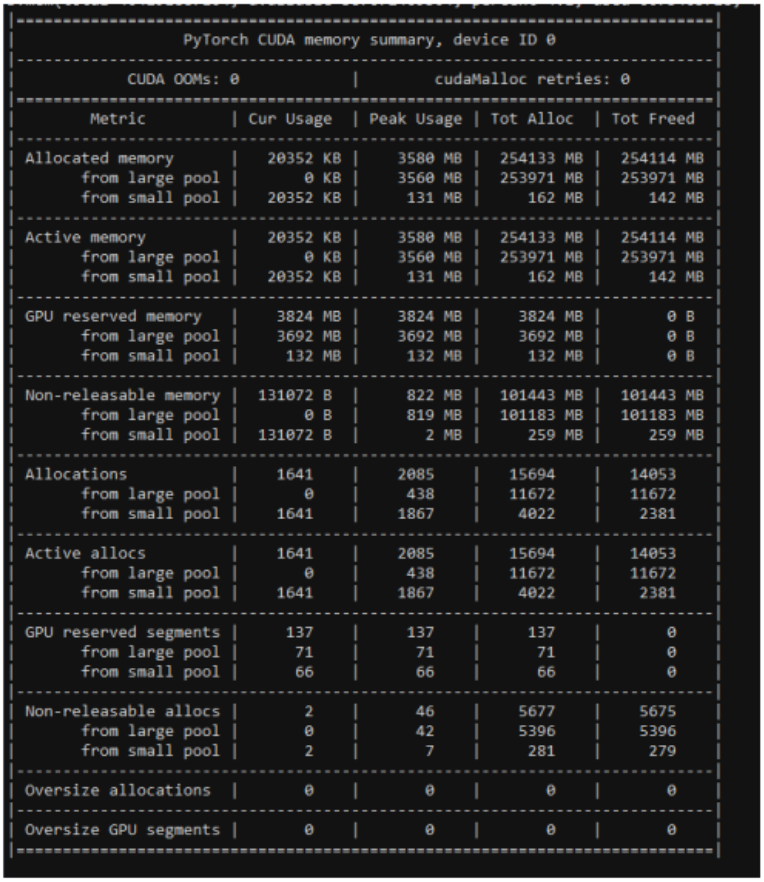
CUDA OOMs: 0 | cudaMalloc retries: 0

Metric	Cur Usage	Peak Usage	Tot Alloc	Tot Freed
Allocated memory	79805 KB	700162 KB	592268 MB	592190 MB
from large pool	76480 KB	696832 KB	591987 MB	591912 MB
from small pool	3325 KB	3591 KB	281 MB	278 MB
Active memory	79805 KB	700162 KB	592268 MB	592190 MB
from large pool	76480 KB	696832 KB	591987 MB	591912 MB
from small pool	3325 KB	3591 KB	281 MB	278 MB
GPU reserved memory	1024 MB	1024 MB	1024 MB	0 B
from large pool	1020 MB	1020 MB	1020 MB	0 B
from small pool	4 MB	4 MB	4 MB	0 B
Non-releasable memory	6211 KB	166654 KB	37741 MB	37735 MB
from large pool	5440 KB	165888 KB	37456 MB	37450 MB
from small pool	771 KB	3881 KB	285 MB	284 MB
Allocations	107	113	11422	11315
from large pool	18	23	7934	7916
from small pool	89	92	3488	3399
Active allocs	107	113	11422	11315
from large pool	18	23	7934	7916
from small pool	89	92	3488	3399
GPU reserved segments	15	15	15	0
from large pool	13	13	13	0
from small pool	2	2	2	0
Non-releasable allocs	6	10	2539	2533
from large pool	2	5	1040	1038
from small pool	4	6	1499	1495
Oversize allocations	0	0	0	0
Oversize GPU segments	0	0	0	0

```
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$ seff 1382375
Job ID: 1382375
Cluster: i5
User/Group: sm3g19/fp
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 2
CPU Utilized: 00:00:49
CPU Efficiency: 68.06% of 00:01:12 core-walltime
Job Wall-clock time: 00:00:36
Memory Utilized: 3.15 GB
Memory Efficiency: 29.87% of 10.55 GB
```

Figure 1: The GPU (right) and CPU usage when running just validation of the s/16 model

Appendix F:



|===|
PyTorch CUDA memory summary, device ID 0
CUDA OOMs: 0	cudaMalloc retries: 0
===
Metric	Cur Usage	Peak Usage	Tot Alloc	Tot Freed
Allocated memory	20352 KB	3580 MB	254133 MB	254114 MB
from large pool	0 KB	3560 MB	253971 MB	253971 MB
from small pool	20352 KB	131 MB	162 MB	142 MB
Active memory	20352 KB	3580 MB	254133 MB	254114 MB
from large pool	0 KB	3560 MB	253971 MB	253971 MB
from small pool	20352 KB	131 MB	162 MB	142 MB
GPU reserved memory	3824 MB	3824 MB	3824 MB	0 B
from large pool	3692 MB	3692 MB	3692 MB	0 B
from small pool	132 MB	132 MB	132 MB	0 B
Non-releasable memory	131072 B	822 MB	101443 MB	101443 MB
from large pool	0 B	819 MB	101183 MB	101183 MB
from small pool	131072 B	2 MB	259 MB	259 MB
Allocations	1641	2085	15694	14053
from large pool	0	438	11672	11672
from small pool	1641	1867	4022	2381
Active allocs	1641	2085	15694	14053
from large pool	0	438	11672	11672
from small pool	1641	1867	4022	2381
GPU reserved segments	137	137	137	0
from large pool	71	71	71	0
from small pool	66	66	66	0
Non-releasable allocs	2	46	5677	5675
from large pool	0	42	5396	5396
from small pool	2	7	281	279
Oversize allocations	0	0	0	0
Oversize GPU segments	0	0	0	0
===

```
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$ seff 1377472
Job ID: 1377472
Cluster: i5
User/Group: sm3g19/fp
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 2
CPU Utilized: 00:00:14
CPU Efficiency: 46.67% of 00:00:30 core-walltime
Job Wall-clock time: 00:00:15
Memory Utilized: 1.38 MB
Memory Efficiency: 0.01% of 10.55 GB
(base) [sm3g19@cyan51 MLP-Mixer-Pytorch-main]$
```

Appendix G: Gannt Charts

Tasks	Start Date	End Date	Description	Status
Literature review	01/10/2021	13/12/2021	On CNN/Vit/MLP	completed
Project brief	01/10/2021	15/10/2021	Submit project brief	completed
Gathering model and setting up	16/10.2021	20/10/2021	Familiarising with the code and setting up pytorch environment and libraries	Completed
Training model on CPU/TPU	22/10/2021	29/10/2021	Running the code on CPU	completed
Training model on GPU	01/11/2021	11/11/2021	Running the code on GPU	completed
Literature review model compression	15/11/2021	19/11/2021	model compression techniques	completed
Implementing model compression into code	22/11/2021	29/11/2021	adding code, debugging, testing and improving	overdue
Progress report	29/11/2021	13/12/2021	writing report	completed
Downloading model onto Jetson Nano	14/12/2021	17/12/2021	Implementing model onto Jetson Nano and running it	not started
Phase 2: Literature review: testing evaluations	31/01/2022	14/03/2022	Read about what each evaulation element signify and affect the effectiveness of model	not started
Evaluating the MLP-Mixer	07/02/2022	21/02/2022	Testing model	not started
Evaluating CNN and Vit	21/02/2022	07/03/2022	testing and comparing agaisnt other models	not started
Phase 3: Literature Review on improving model	07/03/2022	26/03/2022	Looking into how to control FLOPs/ dynamic & static solutions	not started
Improvement of the mlp model	07/03/2022	25/03/2022	Implemnting methods	not started
Report writing	26/03/2022	24/04/2022	writing report	not started

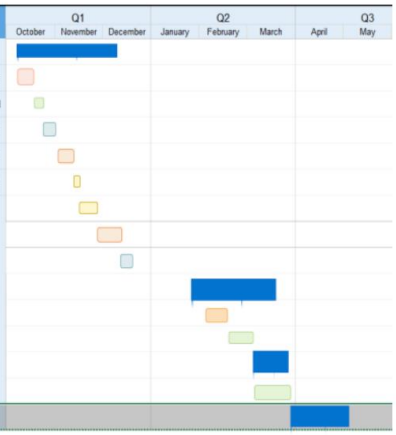


Figure 20: Initial Gannt chart

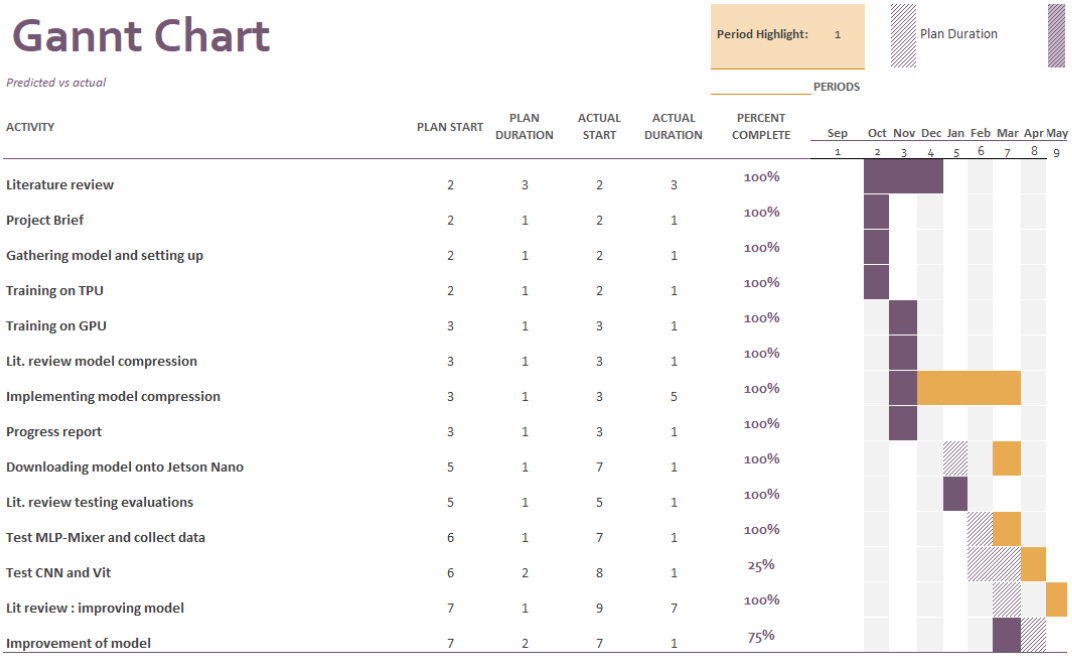


Figure 21: Final Gannt chart