

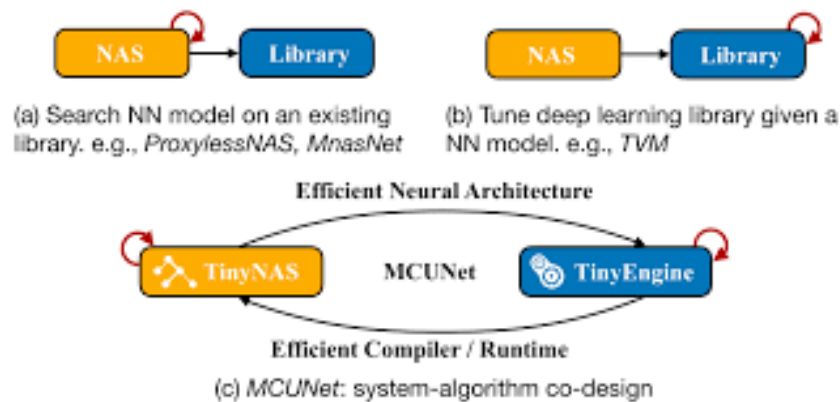
PROJECT REPORT [\(link to Github\)](#)

MCUNet: Analysis and Optimization

By:

Vedasamhitha Challapalli (B20CS078)

Tanisha Jain (B20CS093)



[\(link to the video explanation of the project work\)](#)

Introduction

Following is the report of the Embedded Systems project titled “MCUNET: Analysis and Optimization”. We have read the following papers for understanding Tiny ML and the architecture of MCUNet: <https://arxiv.org/abs/2007.10319> and <https://arxiv.org/abs/2110.15352>. We explained our understandings about MCUNet in the following sections regarding the emergence of Tiny ML, it's benefits over cloud ML and mobile ML, the architecture of MCUNet and its optimization into MCUNet V2 version, and also implemented the **mcunet-in0** model on **CIFAR-10**, **CIFAR-100** and **MNIST**. We have also implemented the model from scratch and compared the results with inbuilt models.

The report covers the following topics:

-
- Tiny ML
 - How CNNs are built on microcontrollers
 - Challenges faced my Tiny ML
 - How MCUNet overcame the challenges
 - What are Tiny NAS and Tiny engine
 - Optimization of MCUNet
 - MCUNet V2 architecture
 - Various resources utilized
 - Applications of MCUNet
 - Implementations (pre-trained and from scratch implementations)
 - Tweaking the architecture to perform on Gray Scale images (for MNIST dataset)
 - Comparisons
 - Analysis
 - Conclusion

TINY AI

Deep Learning is going tiny day by day, from cloud AI, to mobile AI, to Tiny AI. Tiny AI is something which can be accessed by IoT devices or microcontrollers also. There are many advantages when AI is getting accessed through microcontrollers. Many devices in the world are based on microcontrollers, also they are available at low cost, so many people can afford them. Also microcontrollers are less power and storage consuming. When AI is accessed through tinier systems, it can have varied applications, especially matching with those of embedded systems. Because typically, squeezing AI into microcontrollers is another example of **Embedded Systems** in itself. So the applications can be in every field such as healthcare, agriculture, household..etc. But there are also disadvantages with the microcontrollers. They cannot hold deep neural networks due to smaller memory and storage capacity. Microcontrollers are two to three orders smaller in magnitude compared to cloud or mobile AI. Also, like in mobile AI where there is a latency constraint, in tiny AI, there is memory constraint

CNNs on microcontrollers

CNN architecture has various activation layers applied and to these, when the pool kernel is applied to the input activation, we get the output activation. Now, the input and the output activation layers are stored in the SRAM (memory) and the kernel is stored in the DRAM or flash (storage). Now if we see the STM32F429ZI board, the SRAM is 256kB and the flash is 2MB. The flash usage is therefore static because it will hold the entire model whereas the memory usage is dynamic since the activation layers change for each convolutional layer. Therefore the CNNs we use today are way bigger than the memory available. For example: ResNet-50 needs nearly 8000 kB and MobileNet V2 optimized model needs nearly 2000kB. These are way bigger than 256kB. We should not only reduce the model size, but also the activation size. Therefore, we use **MCUNet**

Main Challenges faced

- Less memory
- Less storage
- Both model size and activation layer size must be reduced to fit in the microcontrollers

To overcome these challenges, **MCUNet** is used

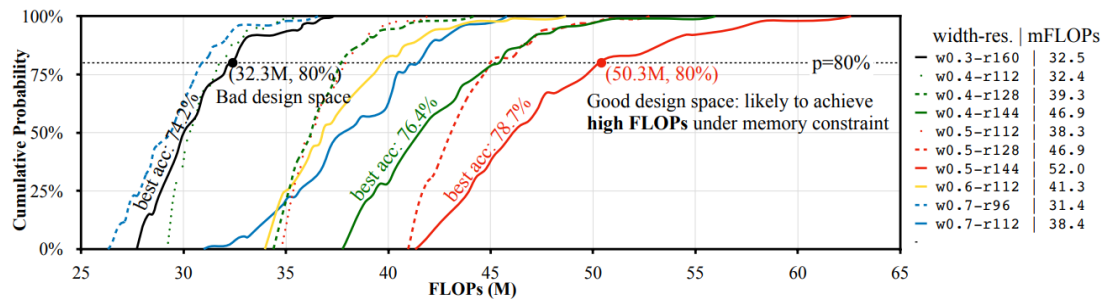
MCUNet

MCUNET is a system-algorithm co-design specifically designed to overcome the issue of memory and storage in microcontrollers. There is an approach where we search the neural network models on an existing library such as ProxylessNAS, MnasNet..etc. Another approach is we tune the deep learning library given a neural network model such as TVM..etc. But given the memory constraint in microcontrollers, we cannot do either of these approaches, we need to design both the neural network and the inference engine ourselves, this would provide efficient compilation and runtime. So TinyNAS is about optimizing the neural network and Tiny Engine is about optimizing the inference

TinyNAS

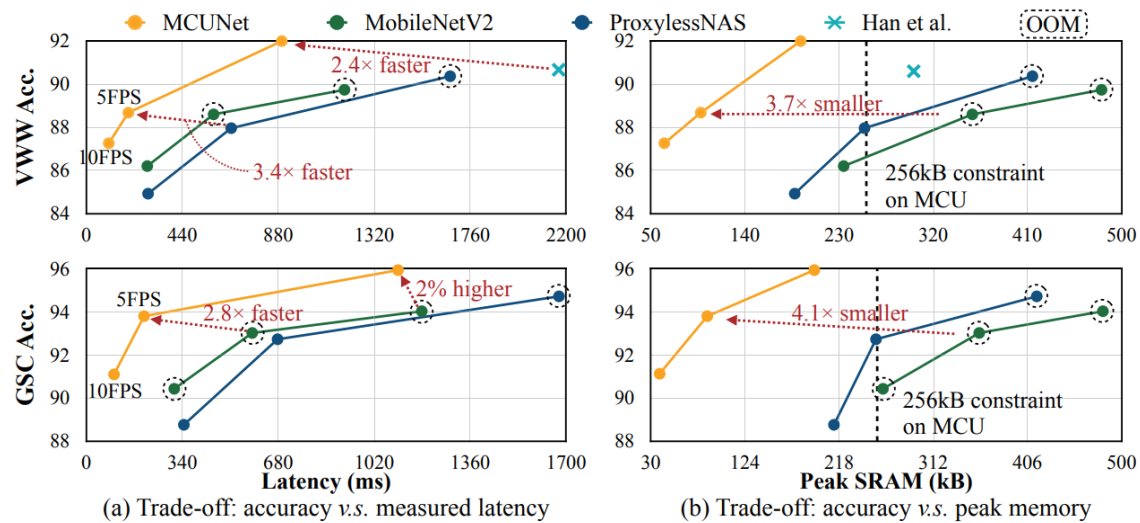
This is all about finding the right search space, one approach can be reusing the given search space carefully, or another approach can be to customize the search space for IoT devices. So, finally, the design space is designed first and then the sub network is searched.

- **Stage1: Automated Search Space Optimization:** The key principle here is that, computation is cheap, memory is expensive. So we need to perform more computations given a memory constraint. So, a plotting has been done to check how many FLOPs fit for a particular computational probability. And it has been observed that 20% of the models are larger than 32M FLOPs and 20% of the models are larger than 50M FLOPs. Based on this, different configurations for SRAM and Flash have been optimized.



- **Stage2: Resource-constrained model specialization:** Here, a super network is created by jointly fine tuning multiple sub networks and the smaller child networks are nested in the larger ones

By combining the two stages, MCUNET has been designed and following the stats, it has shown higher accuracy at lower memory.

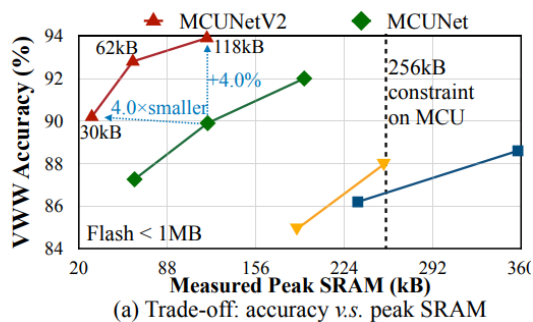


MCUNet Optimization (MCUNet V2)

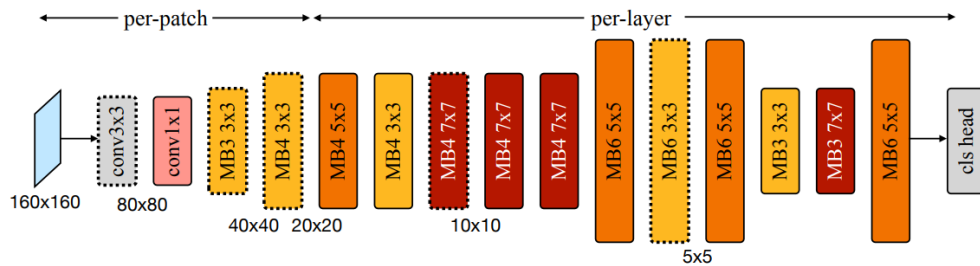
Changes made to original MCUNet to make it perform better:

- Saving memory with patch based interference
 - Tiny engine has been used for this and the calculations have been observed on the STM32F746 board
 - But, **repeated computation** problem arised
- Network redistribution to reduce overhead
 - Showed reduced overhead on image classification and object detection
- Joint Automated search for optimization

On the whole this showed elevated performances when compared to MCUNet alone



MCUNetV2 ARCHITECTURE



The architecture has different kernels right from the darker red (7X7) kernel to the lighter yellow (3X3) kernel. Kernel size in the per patch region is small to reduce spatial overlapping, also, expansion ratio in the middle stage is small to reduce peak memory, but larger in the later stage to improve performance. Input resolution is enhanced to fit in resolution sensitive datasets like the VWW, etc..

Resources Utilized

A big question that occurs is how to calculate the resources that MCUNet utilizes

SRAM and Flash: SRAM and flash can easily be calculated using the compiler's peak memory concept. Using it, we can calculate the SRAM and flash utilized by the CNN when run on the microcontroller.

Latency: MOP (model OP count) is used to calculate the latency of the model. This is very useful especially when sampling occurs from the same super set

Energy: Energy consumption is also a function of OP count, using OP count, energy can be calculated for a particular MCUNet

Applications of MCUNet

- **ImageNet- level image classification:** Using MCUNet, imageNet level image classification accuracies have been achieved

-
- **Face recognition:** Attributes like enabling face recognition only when a person is in front of the camera can be possible using MCUNet. This saves a lot of power consumption.
 - **Larger input resolution:** Object detection becomes much easier when larger input resolutions are allowed since object detection is input resolution sensitive. Patch based interference can be used here.
 - **Keyword spotting:** Translating the time domain into the frequency domain can enable this application. Using MCUNet, CNNs can perform better than DNNs in keyword spotting
 - **Video surveillance and other security related benefits:** Anomaly detection of MCUNet can improve the video surveillance performance better. Autoencoders are useful for this
 - **Speech recognition**
 - **Noise cancellation**

Implementation

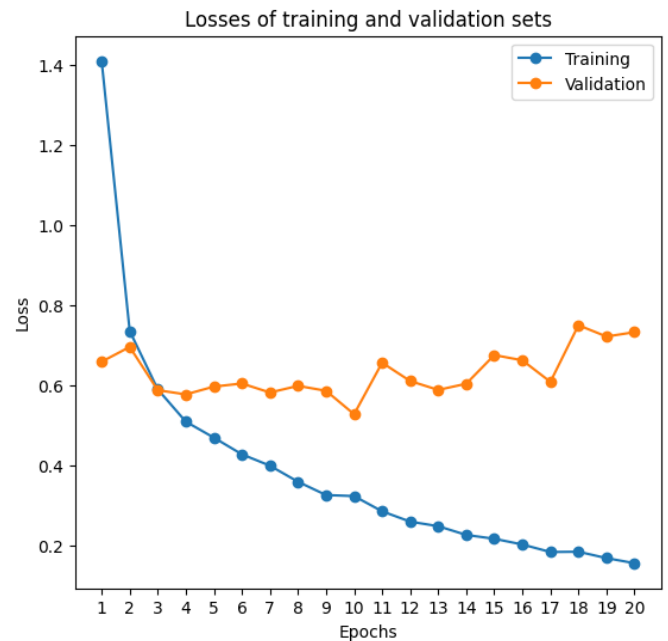
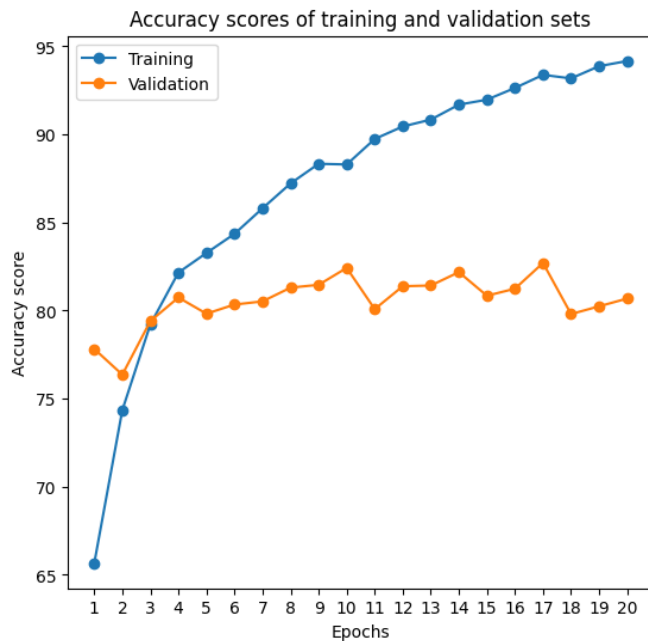
Pre trained model “**mcunet-in0**” has been chosen to be implemented on the datasets of CIFAR-10, CIFAR-100 and MNIST. The same model (**mcunet-in0**) has been studied and was tried to be implemented from scratch by optimizing it a little by modifying a few layers and removing and adding a few layers. The results have been compared in the next section

Pretrained:

1) CIFAR-10:

- CIFAR-10 dataset has been loaded
- It has been divided into training, testing and validation sets as 70-20-10 ratio
- Data Loading has been done with a batch size of 32
- Mcunet-in0 has been imported from the models_zoo
- It was implemented on the training set
- With a learning rate of 0.01 and momentum of 0.9, using an SGD optimizer, the training has been done for 20 epochs

- The corresponding training and validation accuracies and losses have been plotted
- The evaluation has been done on testing set



Test accuracy: 80.98%

Test loss: 0.73

2) CIFAR 100:

- CIFAR-100 dataset has been loaded
- It has been divided into training, testing and validation sets as 70-20-10 ratio
- Data Loading has been done with a batch size of 32
- Mcunet-in0 has been imported from the models_zoo
- It was implemented on the training set
- With a learning rate of 0.01 and momentum of 0.9, using an SGD optimizer, the training has been done for 20 epochs
- The corresponding training and validation accuracies and losses have been plotted
- The evaluation has been done on testing set

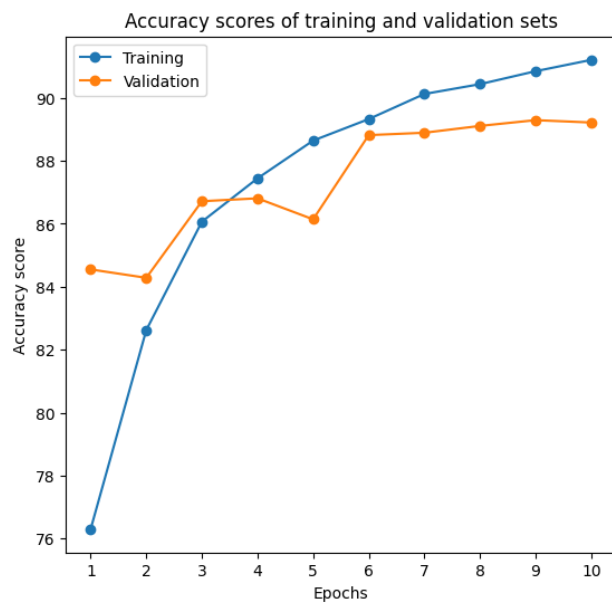


Test accuracy: 51.24%

Test loss: 2.25

3) MNIST:

- MNIST dataset has been loaded
- It has been divided into training, testing and validation sets as 70-20-10 ratio
- Data Loading has been done with a batch size of 32
- Mcunet-in0 has been imported from the models_zoo
- **Since the input tensor size of MNIST is $[(1,28,28)]$, the first convolution layer and kernel size has been tweaked to take the input as 1 rather than 3**
- It was then implemented on the training set
- With a learning rate of 0.001 and momentum of 0.8, using an SGD optimizer, the training has been done for 20 epochs
- The corresponding training and validation accuracies and losses have been plotted
- The evaluation has been done on testing set



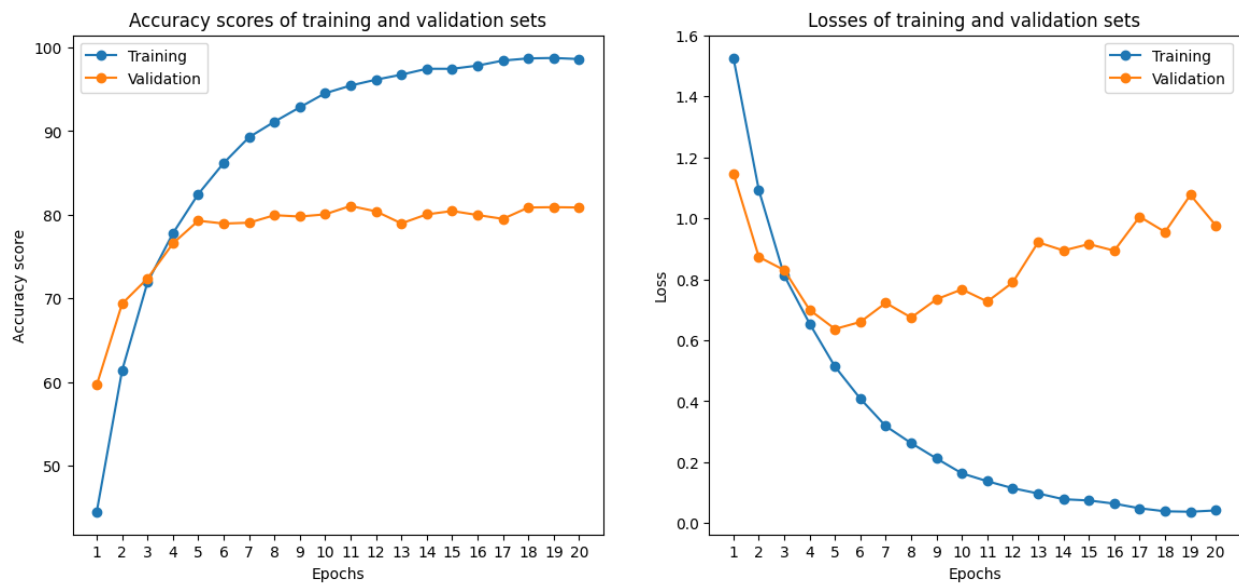
Test accuracy: 98.5

Test loss: 0.05

From scratch (Optimized versions):

1) CIFAR 10:

- CIFAR-10 dataset has been loaded
- It has been divided into training, testing and validation sets as 70-20-10 ratio
- Data Loading has been done with a batch size of 32
- Mcunet-in0 has been imported from the models_zoo
- **The inbuilt model has been analyzed and a simpler model from scratch has been designed to match the performance in a similar way as the inbuilt one. Yet, it is a much optimized one due to lesser number of layers**
- It was implemented on the training set
- With a learning rate of 0.01 and momentum of 0.9, using an SGD optimizer, the training has been done for 20 epochs
- The corresponding training and validation accuracies and losses have been plotted
- The evaluation has been done on testing set

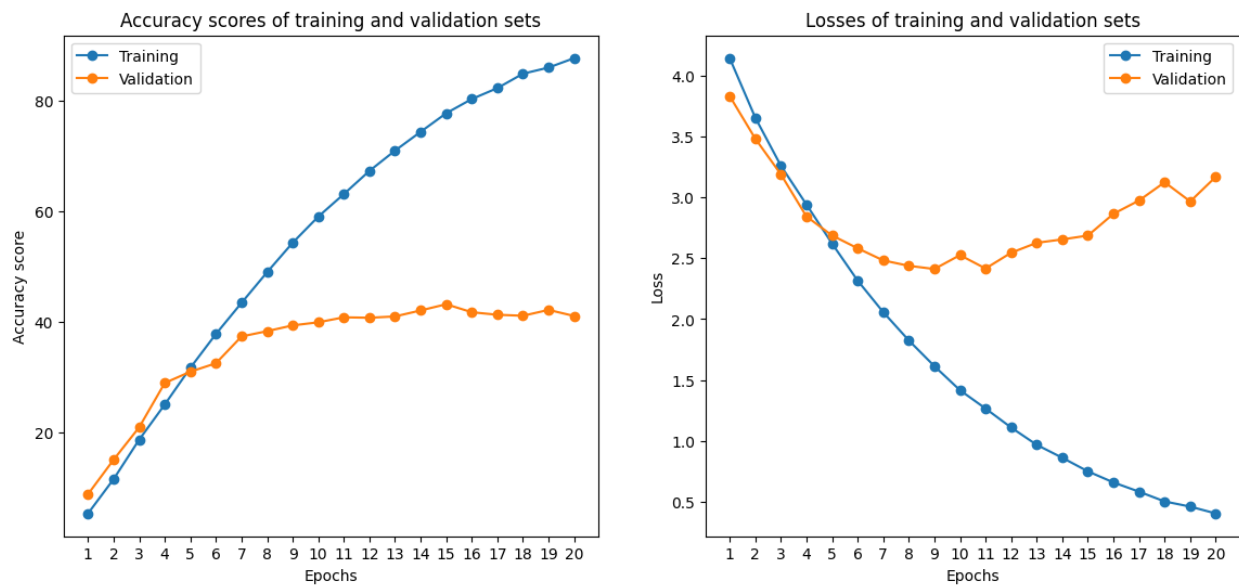


Test accuracy: 80.38

Test loss: 1.05

2) CIFAR 100:

- CIFAR-10 dataset has been loaded
- It has been divided into training, testing and validation sets as 70-20-10 ratio
- Data Loading has been done with a batch size of 32
- Mcunet-in0 has been imported from the models_zoo
- **The inbuilt model has been analyzed and a simpler model from scratch has been designed to match the performance in a similar way as the inbuilt one. Yet, it is a much optimized one due to lesser number of layers**
- It was implemented on the training set
- With a learning rate of 0.01 and momentum of 0.9, using an SGD optimizer, the training has been done for 20 epochs
- The corresponding training and validation accuracies and losses have been plotted
- The evaluation has been done on testing set



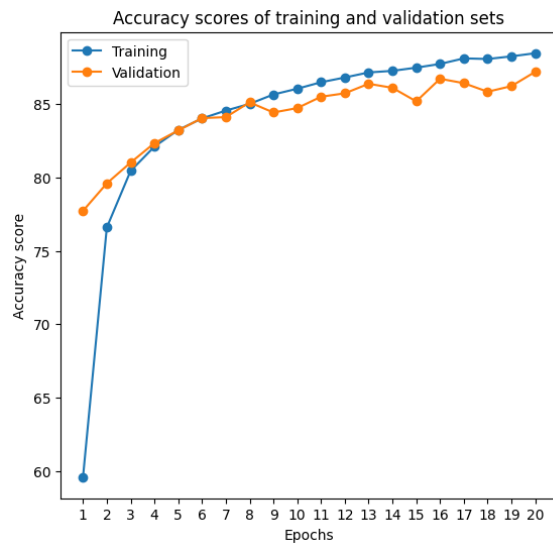
Test accuracy: 41.27

Test loss: 3.21

3) MNIST:

- CIFAR-10 dataset has been loaded
- It has been divided into training, testing and validation sets as 70-20-10 ratio
- Data Loading has been done with a batch size of 32
- Mcunet-in0 has been imported from the models_zoo
- **The inbuilt model has been analyzed and a simpler model from scratch has been designed to match the performance in a similar way as the inbuilt one. Yet, it is a much optimized one due to lesser number of layers**
- **Since the input tensor size of MNIST is $[(1,28,28)]$, the first convolution layer and kernel size has been tweaked to take the input as 1 rather than 3**
- It was implemented on the training set
- With a learning rate of 0.0001 and momentum of 0.5, using an SGD optimizer, the training has been done for 10 epochs
- The corresponding training and validation accuracies and losses have been plotted

- The evaluation has been done on testing set

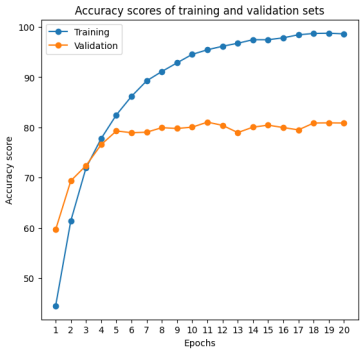
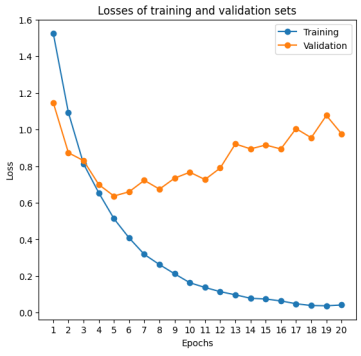
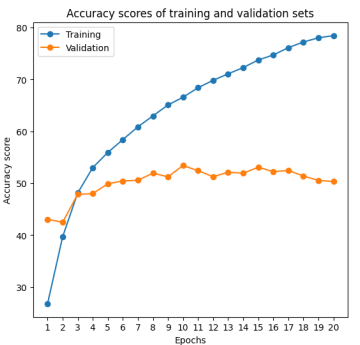
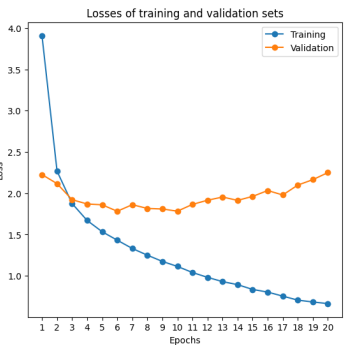
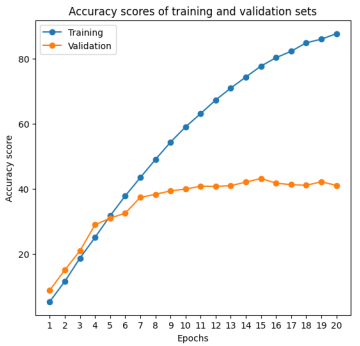
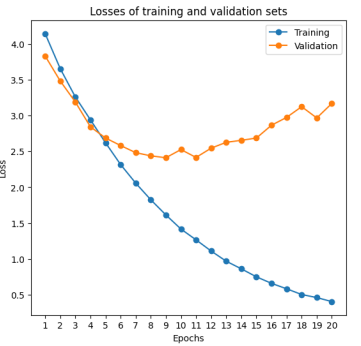
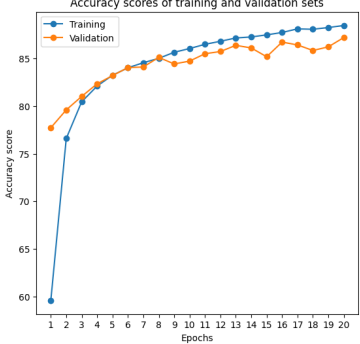
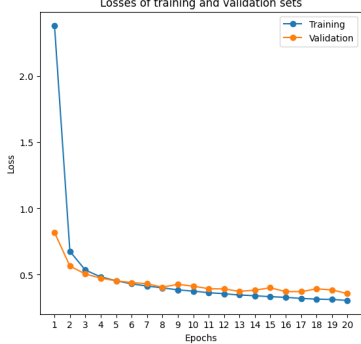


Test accuracy: 86.78

Test loss: 0.35

Comparison:

	Training and validation Accuracies and losses	Testing accuracy	Testing loss
CIFAR-10 (pretrained)		80.98%	0.73

CIFAR-10 (scratch)	<div>   </div>	80.38%	1.05
CIFAR-100 (pretrained)	<div>   </div>	51.24%	2.25
CIFAR-100 (scratch)	<div>   </div>	41.27%	3.21
MNIST (pretrained)	<div>   </div>	86.78%	0.35

MNIST (scratch)	<div style="display: flex; justify-content: space-around;"> <div data-bbox="435 237 781 575"> <p>Accuracy scores of training and validation sets</p> </div> <div data-bbox="802 237 1154 575"> <p>Losses of training and validation sets</p> </div> </div>	98.5%	0.05
----------------------------	--	-------	------

The performances of from scratch and inbuilt are nearly the same. However, the model performed the best for MNIST and the least for CIFAR-100

CONCLUSIONS

- There are many advantages when AI is getting accessed through microcontrollers.
- But there are also disadvantages with the microcontrollers. They cannot hold deep neural networks due to smaller memory and storage capacity.
- Both model size and activation layer size must be reduced to fit in the microcontrollers
- To overcome these challenges, **MCUNet** is used
- Optimized version of MCUNet is MCUNetV2
- Resources such as SRAM, Flash, latency and energy utilized can be calculated
- MCUNet has applications in almost every field
- One such application (Image classification) has been explored in depth in this project
- Both pre-trained and from scratch implementations gave similar results for all the datasets (CIFAR-10, CIFAR-100 and MNIST)
- Adding Max Pool layer, which was not there previously accelerated the performance

-
- However, the model performed the best for MNIST and the least for CIFAR-100, so the model works really well for grayscale images.

CONTRIBUTIONS

Both the team members Veda and Tanisha have done the project with equal contributions. We started with reading the papers and understanding the model and the architecture of MCUNet and MCUNetV2, we then analyzed the various layers of the architecture and also how the model can be optimized. We also understood how the resources utilized can be calculated. After sufficient reading of the papers has been done, we implemented the mcunet-in0 model on the three datasets. After understanding the architecture of the model, we tried to optimize it a little by removing a few layers like the max pool layer and modifying a few. Finally, we were able to achieve an optimum performance almost near to the inbuilt models. Then, we compared the results by plotting the graphs which were shown above.

REFERENCES

<https://arxiv.org/abs/2007.10319>

<https://arxiv.org/abs/2110.15352>

https://www.youtube.com/watch?v=YBER-SNlkqs&ab_channel=MITHANLab

https://www.youtube.com/watch?v=ZW_c1Oozhsw&t=2652s&ab_channel=xLABforSafeAutonomousSystems

<https://github.com/mit-han-lab/mcunet>