University of Southampton

Faculty of Engineering and Physical Sciences

Electronics and Computer Science

Dynamic DNN Inference

By

Haoyu Wang

September 2021

Supervisor: Geoff Merrett
Second Examiner: Ruomeng Huang

A dissertation submitted in partial fulfilment of the
degree of MSc: Internet of Things

Abstract

We have many efficient neural network designs now. For example, we can obtain some convolutional neural networks with excellent operation efficiency through MCUNet , which can be run on our micro platform after further optimization and pruning, such as STM32F746 development board. But in this way, we use static network which is not flexible. So, there is no way to allocate computing resources the way we want to, or based on the resources the system can provide, or the priority of the task. Therefore, we want to optimize the inference process. Thus, in the process of inference operation, we can call different networks in real time according to the priority of task or available resources of the system to ensure that the most efficient network can be used under different situation. Even further, we hope to change the operation mode of the inference engine to achieve a smaller network and lower computing resources by disabling some nodes in the inference process. With these solutions, we successfully achieve dynamic inference and efficiently do inference with the most fitted network. And change network during runtime without stop the inference progress. We can say that the process of neural network reasoning becomes intelligent and humanized.

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.

https://github.com/mit-han-lab/tinyml
https://github.com/tensorflow/tflite-micro

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.

Acknowledgements

Geoff Merrett: gvm@ecs.soton.ac.uk

Lei Xun: lx2u16@ecs.soton.ac.uk

Ji Lin: jilin@mit.edu

List of Acronyms

IoT: Internet of Things

AI: Artificial Intelligence

NAS: Neural Architecture Search

AutoML: Automated Machine Learning

HO: Hyperparameter Optimization

CNN: Convolutional Neural Networks

DNN: Deep Neural Network

MCU: microcontroller units

OFANet: Once-For-All Network

TF: TensorFlow

TF-lite: TensorFlow lite

TinyML: Tiny Machine Learning

RL: Reinforcement Learning

FLOPS: Floating-point Operations Per second

RNN: Recurrent Neural Network

ML: Machine Learning

Table of contents

Chapter 1: Introduction

1.1 Background

In recent years, with the rise and continuous development of the IoT technology, people have been endowed with the ability to communicate with the Internet, perceive the environment, and even react to objects. However, we have always expected that the interaction with objects can be simpler and more humanized, and the functions of the objects themselves can be more complex and intelligent.

But limited by many characteristics of the IoT devices themselves, we don't have a good way to make the objects in the IoT system more intelligent. Cost requirements, for example, make it impossible to use high-end chips, so computing power is limited. Power consumption requirements make it impossible to use a high-power consumption ratio hardware to reduce cost. For another example, volume requirements cause our device sensors, such as camera modules, must be controlled in an exceedingly small volume, so the perception of the outside world is limited, which means we can’t get a high-quality sensor data to process. Many of these constraints are interlocking, making it difficult for IoT devices to become smart.

In this paper, we focus on the use of Convolutional Neural Networks (CNN) for image recognition. This enables IoT devices to gain basic image discrimination. Therefore, the algorithms and optimization schemes generated in this field will be described in detail.

As mentioned earlier, convolutional neural networks for image recognition are developing rapidly. From AlexNet, to ResNet, GoogleNet, DenseNet and, most recently, Transformer. Although there are so many high-quality neural networks, the design of these network structures is limited to the manual design by experienced engineers or scholars, and many of the designs are accidental. As a result, people began to delegate the task of designing efficient neural networks to computers. almost all possible situations, let it automatically find the most suitable network for a particular condition, so as to achieve higher requirements. This method is called Neural Architecture Search, or NAS. It takes a prominent role in recent ML developments. In essence, it's a technique for automating the design of neural networks. Models found by NAS are often on par with or outperform han-designed architectures for many types of problems. The goal of NAS is to find the optimal architecture.

When NAS is used, the quality of the searched neural network is highly dependent on the search space. So there has been a tendency to use a larger and wider search space, trying to cover every possibility. But in fact, this thought brings a lot of trouble to the

use of NAS. Because we have to do our best to train the network individually for each possible hyper-parameter and verify the network's accuracy. This makes our computing power awfully expensive. So, people control the search space and assist with a variety of different search strategies. Thus, the energy consumption can be reduced while maintaining the high-quality network.

Therefore, we have a lot of efficient neural network design, which can even reduce the number of parameters while obtaining higher accuracy, to reduce the time required for inference operation.

In addition, for the network obtained by NAS, people greatly optimize the use of memory space through the quantization method, which greatly reduces the resources needed for inference operations. Also, it largely reduces the memory size needed, not only FLASH but also RAM (Random Access Memory). This kind of quantization is often done at only a small cost of accuracy, and sometimes it even performs better. For this reason, quantification operations are seen as key to being able to reason on small devices on the Internet of Things. At the same time, we can further optimize the network use method such as pruning. We know that not every link in the network is of great significance. In many cases, when the weight of the link is low, we can prune the link, so as to obtain a network with fewer parameters and less computation.

With these optimizations, we can reduce the computational power required to do neural network reasoning by another order of magnitude.

1.2 Project Management

To efficiently finish our project, we manage the time into several period, and each time period will have a certain job to do. Here is the Gantt chart for my project.

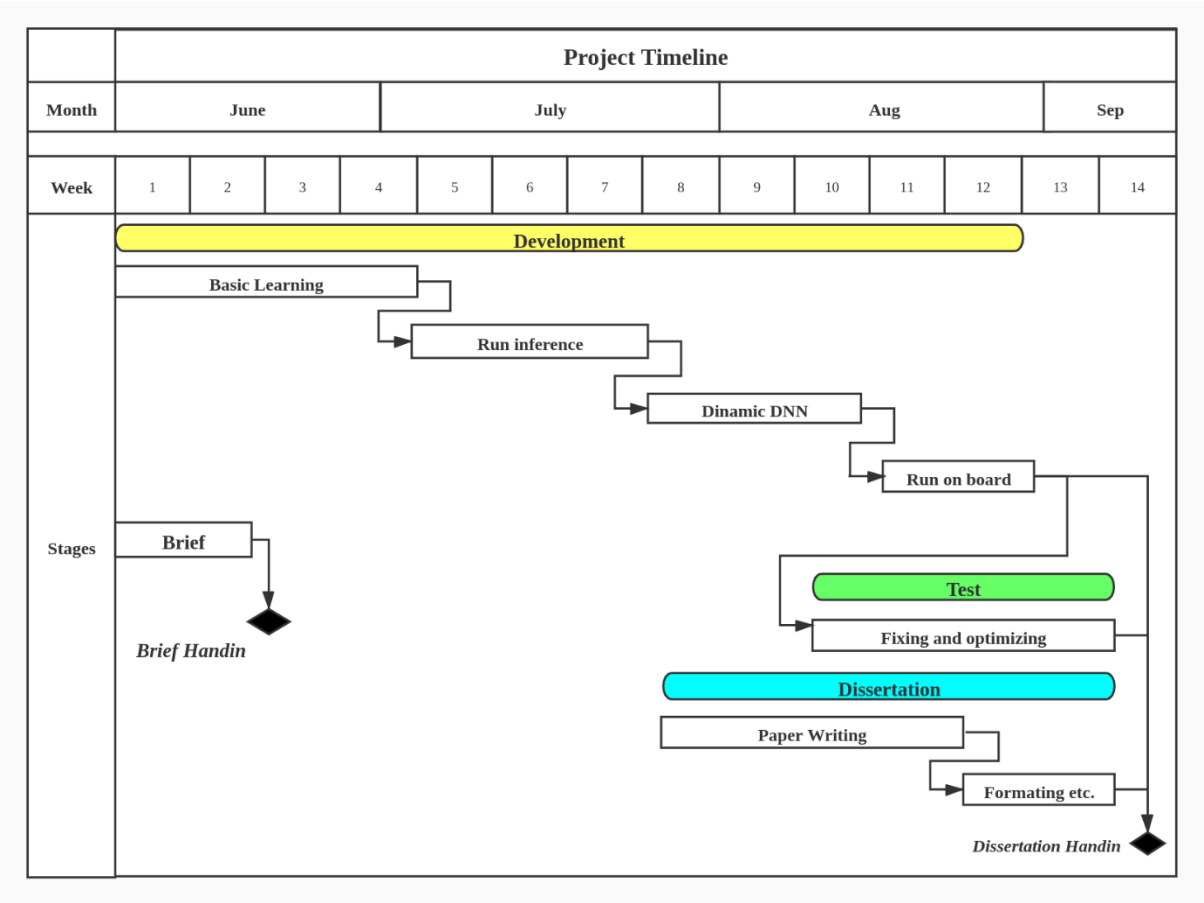


Figure 1.2: Project Gantt Chart

Because the IoT background I'm holding, we first need to get some basic understanding about deep learning. Then we basically just follow the step-by-step implementation method which will be talked in detail in 3.1 part to implement our algorithm.

Then, from week 10, we need to test our algorithm and the application running on board. And use the result we get to further optimise our application and algorithm.

The dissertation and video recording will start at week 8 and continue to the end of this project.

1.3 The problems and objectives

All the networks we have implemented so far, even though they have been designed to be very good and efficient, cannot hide the fact that they are static networks. Whether it's MobileNet, once-for-all Network, Proxyless, MCUNet or any other Network. At present, neural networks are static in the process of inference. A static network means that once a network is created, it has almost no flexibility left. This means that our network will participate in subsequent tasks in a fixed form as a complete and immutable whole. These tasks include inference, validation and so on.

Therefore, people began to hope that the network has greater flexibility in operation, so the concept of dynamic neural network came into being. Dynamic neural network has many important advantages compared with traditional static neural network.

First of all, the use of dynamic neural networks can undoubtedly achieve higher efficiency, because all resources in the network can be provided according to needs. For each network, for each constraint, what resources we can provide can be allocated dynamically, thus achieving higher efficiency. One of the main purposes of using dynamic neural networks is to minimize the amount of computation required to meet latency and other requirements with little loss of accuracy. So, less computation naturally means more efficiency.

Second, it has become more adaptable. Because the dynamic model can achieve a dynamic balance between accuracy and efficiency, the model can be deployed on different hardware platforms with different available resources of the current system, independently and flexibly to do these trade-offs. Then adjust the model to achieve people's requirements. Therefore, it can be adjusted dynamically on different platforms and automatically achieve similar or not bad results.

Moreover, dynamic inference using dynamic neural network improves the compatibility of the network. In the past, people might need to do a separate output for each network in each case, to get the most efficient specific network in that particular case. But with this situation, over time the number of networks would become uncontrollable. However, with dynamic network, we only need to generate a few networks, let the neural network automatically adjust which part of the network need to be taken in operation according to the requirements, so as to reduce the number of independently generated networks.

At the same time, dynamic neural network can have strong generality. Therefore, the methods and algorithms obtained in dynamic neural networks can be quickly applied to other fields of machine learning.

Last but not least, through dynamic neural networks, we can control the switch of a certain layer, or even whether a certain neuron is in used or not. So, we can further study the uninterpretable problem of neural networks and explore the relationship between neural network and human brain, make the operation logic behind neural network becomes clearer.

Therefore, we hope to adjust the inference process of MCUNet to a dynamic operation, to make better use of system resources, achieve higher efficiency of inference process, and make the system more flexible, and using different networks more convenient computing. By doing so, we can achieve a truly intelligent and convenient algorithm that can run flexibly on IoT devices.

Chapter 2: Literature Review

In this chapter we will further discuss some techniques that we are going to cover. These include Hyperparameter Tuning, NAS and AutoML, TinyML, Dimensionality Reduction, Model Compression, Dynamic Neural Network.

Hyperparameter Tuning, which is a necessary step of all the efficient neural network design, we need to start with this to talk about how we can improve our neural network by adjusting the hyperparameters.

Then it’s the important part of NAS and AutoML, which contains some basic ideas of how and why NAS and AutoML work, why we need them especially when one of the MCUNet’s main idea is to design an efficient tiny NAS algorithm. This part also contains some concepts and terms like search space, search strategy and efficiency measurement.

We will also cover TinyML because we will deploy our DNN on MCU or other tiny source-limited devices, this area is called TinyML. Usually in this area we focus on two main topics, efficient neural network design in order to make the accuracy and efficiency as higher as possible, and efficient inference engine in order to make inference step use less computing power.

Also, we need to talk a little about dimensionality reduction in order to decrease our model dimensionality especially when we deploy them on microcontroller.

Model compression is one of the must include steps in TinyML. The importance is that it can shrink the model size and computing power needed so that it can fit into MCU. We will talk two ways to do model compression, quantization and pruning.

Finally, we will talk about dynamic neural network, and what method we might use in this area. Because we want to make our MCUNet running dynamically.

2.1 Hyperparameter Tuning

In machine learning models, there are two types of parameters. There are model parameters. Those are parameters that the model must learn using the training set. They're fitted or trained parameters of our models. Usually, that means weights and biases. Then we have hyperparameters. These are adjustable parameters that must be tuned to create a model with optimal performance, but unlike model parameters,

hyperparameters are not automatically optimized during the training process. They need to be set before model training begins, and they affect how the model trains.

Hyperparameter tuning can have a high impact on model performance, and unfortunately, the number of hyperparameters can be large even for small models. For instance, in a ShallowNN[1], we can make hyperparameter choices for architecture options, for activation functions, for weight initialization strategy, and optimization hyperparameters, and others. Performing manual hyperparameter tuning can be a real brain teaser, which is a subtle way to put it, since we need to keep track of what we have tried and launch different experiments. So, a manual process like that is tedious. Nonetheless, when done properly, hyperparameter tuning helps boost model performance significantly. Several open-source libraries have been created using various approaches to hyperparameter tuning. The Keras team has released one of the best, Keras Tuner, and Auto Keras[2] which are libraries to easily perform hyperparameter tuning with TensorFlow 2.0. It offers a variety of different tuning methods [4][4], including random search, Hyperband[6], and Bayesian optimization [7].

2.2 NAS and AutoML

Usually, AutoML holds a set of very versatile tools to automate the machine learning process end to end. Finding the right model is an important piece of the whole progress. Neural Architecture Search is a process to help find relevant models.

Search spaces and strategies are key for both hyperparameter tuning using NAS and AutoML. We'll also discuss tools to quantify performance estimation on the explored models. Finally, we will talk about what exactly is AutoML and different AutoML offerings available in the Cloud by major service providers.

2.2.1 NAS

Neural Architecture Search or NAS [7][8]is at the heart of AutoML. There are three main parts to Neural Architecture Search, a search space, a search strategy and a performance estimation strategy. The search space defines the range of architectures that can be represented. To reduce the size of the search problem, we need to limit the search space to the architectures which are best suited to the problem that we're trying to model. This helps reduce the search space, but it also means that a human bias will be introduced, which might prevent Neural Architecture Search from finding architectural blocks that go beyond current human knowledge. The search strategy defines how we explore the search space. We want to explore the search space quickly, but this might lead to premature convergence to a sub-optimal region in the

search space. The aim of Neural Architecture Search is to find architecture is that perform well on our data.

Neural Architecture Search is a subfield of AutoML. It specifically focuses on the model selection part of the AutoML workflow and the design of neural networks. Neural Architecture Search has been used to design architectures that are on par with or outperform hand-designed models.

2.2.2 Search Space

As we said, the search space stands for the scope of architectures that can be explored.

There are two main types of search spaces, macro and micro. First, let's define what we mean by the node. A node is a layer in a neural network like a convolution or pooling layer. [9]

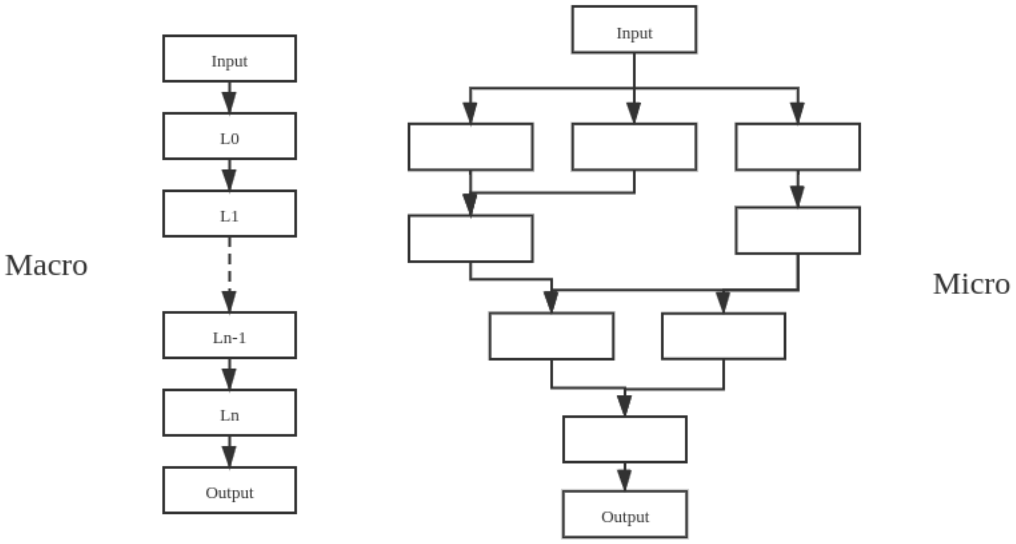


Figure 2.2.2: Micro search space VS Macro search space

A macro search space contains the individual layers and connection types of a neural network. And neural architecture search searches within that space for the best model, building the model layer by layer. A network can be built very simply by stacking individual layers referred to as a chain structured space or with multiple branches and skip connections in a complex space.

By contrast, in a micro search space, neural architecture search builds a neural network from cells where each cell is a smaller network. In the micro search space, there are two different types of cells, a normal cell on the top and a reduction cell on the bottom. Cells are stacked to produce the final network. This approach has been shown to have significant performance advantages compared to a macro approach.

However, it’s still possible that cells can also be combined in a more complex manner, like in multi-branch spaces by simply replacing layers with cells.

2.2.3 Search Strategy

The neural architecture search decides which options in the search space to try next by using a search strategy. Neural architecture search searches through the search space for the architecture that produces the best performance. A variety of different approaches can be used for that search. These include grid search, random search, bayesian optimization, evolutionary algorithms, and reinforcement learning.

In grid search, the algorithm covers every option we have in the search space. In a random search, it selects the next option randomly within the search space. Two algorithms work reasonably well in smaller search space, but both also fail when the search space grows beyond a certain size, which is all too common.

Bayesian optimization [[6], [10] is a little more sophisticated. In this method, we assume that a specific probability distribution like a Gaussian distribution is underlying the performance of model architectures. So, we can use results from tested architectures to constrain the probability distribution and guide the selection of the next option. This allows us to build up an architecture stochastically based on the test results and the constrained distribution.

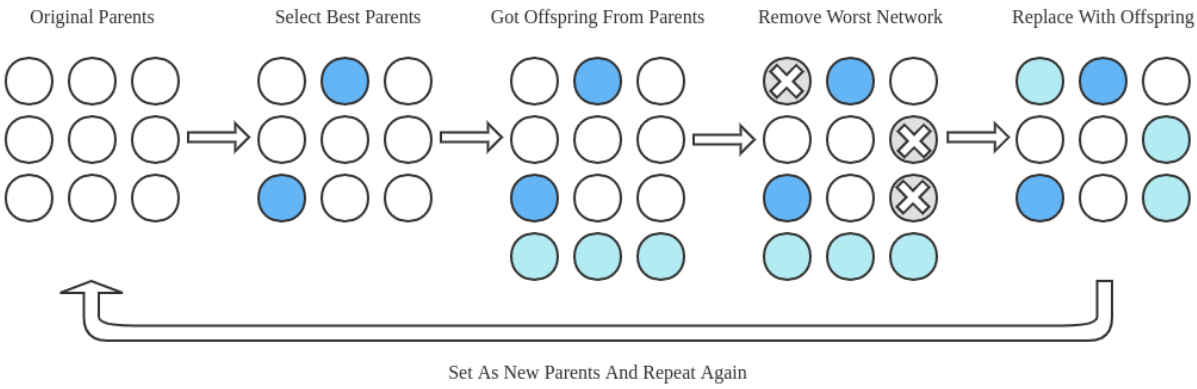


Figure 2.2.3: Evolution Search Step

Neural architecture search can also use an evolutionary method to search.[11], [12] First, we randomly generate n different model architecture and do evaluation on each of them, which the evaluation way is defined by the performance estimation strategy. After we got the result, the top K network will be selected as parents to generate a new generation, which we called as offspring. These new architectures induced random alterations or mutations, or they are just the combinations of the parents. There are a lot of possible mutations, for example, include operations like adding or removing a layer, adding or removing a connection, changing the size of a layer or changing another hyperparameter. The offspring's performance is assessed using the

performance estimation strategy. Then the worst n architectures are selected and removed from the population. We can set the algorithm to choose the worst performers, or oldest individuals in the population. These networks will be replaced by new offspring. Then all population we have now will be the new parents and start this progress all over again.

In reinforcement learning, agents take actions in an environment [13] to maximize a reward. After each action, the state of the agent and the environment is updated, and a reward is issued based on a performance metric. Then the range of possible next actions is evaluated, the environment in this case is the search space. And the reward function is the performance estimation strategy. A neural network can also be specified by a variable-length string where the elements of the string specify individual network layers. That enables us to use a recurrent neural network (RNN) to generate that string, as we might do for an NLP (Natural Language Processing) model. The RNN that generates the string is referred to as the controller. After training on real data, the network is referred to as the child network. We can measure the accuracy on the validation set, the accuracy determines the reinforcement learning reward in this case. Based on the accuracy, we can compute the policy gradient to update the controller RNN. Therefore, in the upcoming iteration, the controller will have learned to give higher probabilities to architectures that result in higher accuracies during training. This method works surprisingly well and can get a super good architecture even it starting from scratch.

2.2.4 Efficiency Measurement

Neural architecture search relies on being able to measure the accuracy or effectiveness of the different architectures that it tries, so it requires a performance estimation strategy. Therefore, they can generate architectures that perform better.

The performance estimation strategy helps in measuring and comparing the performance of various architectures. After the search strategy selects an architecture, the selected architecture will be passed to a performance estimation strategy, which returns its estimated performance to the search strategy.

The simplest approach is to measure the validation accuracy of each architecture that is generated, just like the method used in the reinforcement learning approach. This becomes computational heavy, especially for large search spaces and complex networks. And as a result, it can take several GPU days to find the best architectures using this approach, which makes it expensive and slow. It also makes neural architecture search impractical for many use cases.

To reduce performance cost estimation, several strategies have been proposed, including lower fidelity estimates, learning curve extrapolation, and weight inheritance or network morphism.

Lower fidelity or lower precision estimates try to reduce the training time by reframing the problem to make it easier to solve by training on a subset of data, or just using lower resolution images, or maybe reducing filters per layer and reducing cells. It greatly reduces the computational cost but ends up underestimating performance.

Learning Curve Extrapolation [[14], [15] assumes that we have mechanisms to predict the learning curve reliably, and so extrapolation is a sensitive and valid choice. Based on a few iterations and available knowledge, the method extrapolates the initial learning curves and terminates all architectures that perform poorly. The progressive neural architecture search algorithm [[16], which is one of the approaches for neural architecture search, uses a similar method by training a surrogate or proxy model and using it to predict the performance using architectural properties.

Another method for speeding up architecture search is to initialize the weights of novel architectures based on the weights of other architectures that have been trained before, similar to the way that transfer learning works. One way of achieving this is referred to as network morphism [[18]. Network morphism modifies the architecture without changing the underlying function. This is advantageous because the network inherits knowledge from the parent network so it only requires a few GPU days to design and evaluate. That means we can increase the capacity of the network but still maintain a relatively high performance without training from scratch. Another advantage of this approach is that it allows for search space that don't have a boundary on the architecture's size.

2.2.5 AutoML

Automated Machine Learning or AutoML is aimed at enabling developers with very little experience in machine learning, to make use of machine learning models and techniques. It tries to automate the process of machine learning end to end, in order to produce simple solutions, faster creation of those solutions and models that sometimes outperform even hand-tuned models. AutoML applies machine learning and search techniques to the process of creating machine learning models and pipelines. It covers the complete pipeline from the raw data set to the deployable machine learning model. In traditional machine learning, we write code for all the phases of the process. We start off with ingesting and cleansing the raw data and then perform feature selection and feature engineering. We select a model architecture for a task, train or model and perform hyperparameter tuning, manually or using a tuner like Keras tuner and then we validate our models' performance. ML requires a lot of manual programming and a highly specialized skill set. Auto ML aims to automate the entire ML workflow. If we can provide the AutoML system with raw data and our

model validation requirements. It goes through all the phases in the ML workflow and performs the iterative process of ML development in a systematic way until the model is trained. Also, AutoML facilitates building models in an automated fashion. Furthermore, itis not restricted to a specific family or subset of models.

One of the easiest ways to use AutoML is by using a Cloud service. The top-3 most popular cloud AutoML service are Amazon SageMaker Autopilot[18], Microsoft Azure Automated Machine Learning [19], and Google Cloud AutoML [20]. Each of them has their own advantages and disadvantages, you can find their character in their official description.

2.3 TinyML

Machine learning is increasingly becoming part of more and more devices and products. This includes the rapid growth of mobile and IoT applications, including devices which are situated everywhere from farmers' fields to train tracks. Businesses are using the data which these devices generate to train machine learning models to improve their business processes, products, and services. Even digital advertisers spend more on mobile than desktop. There are already billions of mobile and edge computing devices, and that number will continue to grow rapidly in the next decade. McKinsey predicts that by 2025, the overall economic impact of IoT and mobile could reach trillions of dollars, surpassing many sectors like automation of knowledge work or Cloud technology. As these devices become more and more ubiquitous and powerful, many of the machine learning tasks, which you think of as requiring months of high-powered compute time, will become part of more and more fairly common devices.

Traditionally, you can think of deploying machine learning models in the Cloud. This requires a server to run inference and return the results. But with the advance of machine learning research for applications on lower power devices, this processing can be offloaded to a device and run locally. This enables more opportunities for including machine learning as part of a device's core functionality. Moreover, the hardware costs for these devices continues to fall, which enables lower price points and higher volumes. A key aspect of on-device machine learning is that, in most cases, it ensures greater compliance with privacy regulations by keeping user data on the device.

If we host the model on a server, the mobile or IoT device needs to be connected so that it can make a network request. Another option is to embed the model on a mobile device directly, which is exactly the situation we will talk about. So, it's important that our model small enough and fast enough to perform inference on the device. Additionally, mobile devices offer limited processing capabilities which might affect

which types of models we can embed in them. Furthermore, we should decide whether the device have all the access it needs to the data that it needs, or does it need things like historical data that are only available on a server.

Using a server for inference has the advantage that it keeps the mobile app simple. The server encapsulates all the model complexity. This means that we can update the model or add new features anytime you want. To deploy the improved model, just update the model on the server. That means that you probably don't have to update the app itself, unless you need to change the request that the app sends. One big drawback is the timely inference is a strong requirement in this setting.

In case of on-device inference, we load the trained model into the app. Since the model runs in the app, it doesn't need to send a request over the Internet and wait for a reply. Instead, the prediction happens fast, and it doesn't need a network connection.

For TinyML[21]–[23] involved in our project, there are two main research directions: one is Efficient Neural Network, which is used to optimize the structure of Neural Network. The other is Efficient Inference Engine, which optimizes the Inference process of Inference Engine.

2.2.1 Efficient Neural Network

In recent years, we have seen many excellent and efficient neural networks developed. Some networks, such as MobileNet[24]–[26], developed by Google's team, are excellent examples.

MobileNets are a family of architectures that achieve a state-of-the-art trade-off between on-device latency and ImageNet classification accuracy.

One important method[27] used in MobileNets or other Efficient Neural Networks is that they use integer-only quantization. For any fixed accuracy requirement, latency time is lower for the 8-bit version of the model, that’s because arithmetic with lower bit depth is faster that higher bit depth. Although CPU might not be a huge problem for running float points computation now, but moving from 32 bits to eight bits, we usually get speedups of 4x reduction in memory. That means a lighter deployment model with less storage space. Using 8-bit integer also means that we can load more data into the same caches. This makes it possible to build applications with better caching capabilities that reduce power usage and run faster.

In general, the high computational requirements of traditional NAS algorithms make it difficult to search architectures directly on large-scale tasks. Differentiable NAS[28] can reduce GPU hour cost by continuous representation of network architecture, but have the problem of high GPU memory consumption. Therefore, we need to use proxy tasks as a guide. The proxy task maybe is training on smaller data

sets, or training for only a few periods, or learning with only a few blocks. But these architectures optimized for agent tasks can’t guarantee that it’s suitable for target tasks.

So ProxylessNAS[29] improves on this. It allows algorithms to directly learn the architecture of large-scale target tasks and target hardware platforms. We address the high memory consumption of differentiable NAS and reduce computational costs to the same level as regular training, while allowing it to work on large tasks and data sets. On ImageNet, ProxylessNAS 'model achieved 3.1% higher top-1 accuracy than MobileNetV2, while being 1.2 times faster at measured GPU latency. Moreover, ProxylessNAS has been used to provide an idea for neural architecture design of hardware with direct hardware specifications.

And then we can talk about Once-for-All Network[30], [31]. In this work, they suggest training a one-time (OFA) network to support different architectural setups by separating training and search to reduce costs. Using this method can quickly obtain a specialized sub-network by selecting from among OFA networks without additional training. To effectively train OFA networks, they also propose a new progressive shrinkage algorithm, which is a generalized pruning method that can reduce model size in more dimensions than pruning. And it's done on multiple level such as depth, width, kernel size, and resolution. It can obtain a surprising number of subnetworks (> 1019) and can adapt to different hardware platforms and latency limits while maintaining the same level of accuracy as standalone training. OFA consistently outperforms the most advanced (SOTA) NAS approach on a variety of edge devices (ImageNet Top1 improves accuracy by 4.0% over MobileNetV3, or the same accuracy but 1.5 times faster than MobileNetV3, 2.6 times faster than EfficientNet, with measurement latency) while reducing GPU hours and CO2 emissions by many orders of magnitude.

Finally, the neural network to be used in our project: MCUNet. Because MCUNet is a combination of Efficient Neural Network and Efficient Inference Engine design. So in this part, we will focus on TinyNAS.

TinyNAS uses a two-stage neural architecture search method, first optimizing the search space to adapt to resource constraints. We have a lot of options from a certain device to the flops constraint. Then it designs the network architecture specifically in the optimized search space. TinyNAS automatically handles various constraints at low search costs. For example, devices, latency, energy, memory etc.

2.2.2 Efficient Inference Engine

In terms of local inference engines, the most used is undoubtedly TensorFlow Lite from the Google team. We can deploy the TensorFlow Lite models on our embedded devices. It targets mobile platforms and uses TensorFlow Lite, the Google Cloud

Vision API and Android Neural Networks API to provide on-device machine learning, such as facial recognition, barcode scanning, and object detection among others.

TensorFlow Lite[21], [32] is developed by Google. It has APIs for many programming languages which means it’s easy and convenient to use. It's designed and optimized for on-device applications. So, the interpreter in it is specially tuned for on-device machine learning. To get network into real cases quickly, models are able to be converted to TensorFlow Lite format with many optimization method come with it. TensorFlow Lite also supports optimizing models for IoT and embedded applications, because this is an inference engine optimized for small devices, we can run the same network with much less memory requirements and get a decent degree of accuracy, even for devices with as little as 20k of memory. TensorFlow Lite models can be trained and evaluated in TFX pipelines, which is important when the model will become part of a production, product, or service.

However, Google's inference engine is not the best, and we still have room for further optimization and improvement, so many other inference engines have been derived and developed independently.

In order to accelerate CNN reasoning, existing deep learning frameworks focus on intra-operator parallelization. However, the reality is that due to the rapid development of high-performance hardware, a single operator can no longer take full advantage of available parallelism, resulting in a large gap between peak performance and actual performance. This performance gap is even more severe in smaller batches. Therefore, in IOS, the author extensively studied the parallelism between operators, and proposed an Inter-Operator Scheduler (IOS)[33] to automatically schedule the parallel execution of multiple operators through a novel dynamic programming algorithm. In modern CNN benchmarks, IOS consistently outperforms the most advanced libraries (e.g. TensorRT) by 1.1 to 1.5 times.

As we said TinyNAS is co-designed with TinyEngine, a memory-efficient inference engine that expands search space and fits larger models. TinyEngine adjusts memory scheduling based on the overall network topology rather than layer by layer optimization, resulting in a 3.4 times reduction in memory usage and a 1.7-3.3 times improvement in reasoning speed compared to TF-Lite Micro and CMSIS-NN.

2.4 Dimensionality Reduction

In the not-so-distant past data generation and to some extent data storage was a lot more costly than it is today. Back then a lot of domain experts would carefully consider which features or variables to measure before designing their experiments and feature transforms. As a result, data sets were expected to be well designed and

potentially contain only a small number of relevant features. Today, data science tends to be more about integrating everything end to end, generating and storing data is becoming faster, easier, and less expensive. So, there's a tendency for people to measure everything they can and include ever more complex feature transformations. As a result, datasets are often high dimensional, containing a large number of features, although the relevancy of each feature for analysing this data is not always clear.

There is a common misconception about neural networks. Many developers correctly assume that when they train their neural network models, the model itself, as part of the training process, will learn to ignore features that don't provide predictive information by reducing their weights to zero or close to zero. Although this is true, this will lead us to an un-efficient model. Much of the model can end up being shut off when running inference to generate predictions but those unused parts of the model are still there. They take up space and consume compute resources as the model server traverses the computation graph. Those unwanted features could also introduce unwanted noise into the data, which can degrade model performance. And outside of the model itself each extra feature still requires systems and infrastructure to collect that data, store it, manage updates, etc, which adds cost and complexity to the overall system. That includes monitoring for problems with the data and the effort to fix those problems if they happen. Those costs continue for the lifetime of the product or service that you're deploying, which could easily be years. In general, we shouldn't just throw everything at our model and rely on the training process to determine which features are actually useful.

When pre-processing a set of features to create a new feature set, it's important to retain as much predictive information as possible, without predictive information all the data in the world won't help your model learn. Features must be representative of the predictive information in the data set. This information also needs to be in a form that will help your model learn. While some inherent features can be obtained directly from raw data, you often need derived features, normalized, engineered or embedded features. A poor model fed with important features will perform better than a fantastic model fed with low quality or bad features. Many domains involve vast numbers of features and dimensions.

Often the first pick of features is an expression of domain knowledge, that can often result in more features than we really need or want. This means that you need to reduce dimensionality, or more precisely, the number of features you're including in your dataset while retaining or improving the amount of predictive information contained in the data.

Dimensionality reductions [[34]–[36] most essential function is to reduce the data set to the level that they only consist of useful info. That means we need to discard noisy features, by doing this we can avoid the noisy features cause a horrible result for supervised learning tasks.

Dimensionality reductions also reduce multicollinearity by removing redundant features, it helps when we're trying to visualize the data. In addition, features selection examines a set of potential features, select some of them and discards the rest. That's means usually, the best results from dimensionality reduction is come down to the user itself, the practitioner crafting the features.

In addition to manually reducing the dimensionality of datasets, we can also apply several algorithmic [[34]–[36] approaches to do dimensionality reduction.

First, in this approach, we linearly (but we can also do it non-linearly [[24] n-dimensional data onto a smaller k- dimensional subspace. Here, k is usually much smaller than n. There are infinitely many dimensional subspaces that we can project data onto. There are several ways to choose these k-dimensional subspaces. For example, in a classification test, you typically want to have the maximum separation among classes. Linear Discriminant Analysis, or LDA, generally works well for that. For regression, you want to maximize the correlation between the projected data and the output, where Partial Least Squares, or PLS, works well. Finally, and unsupervised tasks, we typically want to keep as much of the variance as possible. Principal Component Analysis, or PCA (Principal Component Analysis) [[25]–[27], is the most widely used technique for doing that. We can even use genetic algorithms [[28] to decrease the dimension of input data.

2.5 Model Compression

Model optimization is an area that can further optimize performance and resource requirements. The goal is to create models that are as efficient and accurate as possible and to achieve the highest performance at the least cost.

2.5.1 Quantization

Quantization involves transforming a model into an equivalent representation that uses parameters and computations at a lower precision. This improves the model's execution performance and efficiency, but it can often result in lower model accuracy. Quantization, in essence, lessens or reduces the number of bits needed to represent information. That’s because Neural network models can take up a lot of space. For example, AlexNet needs around 200 megabytes of disk space. And almost all these disk spaces are used to store the weights of every connection between millions of neurons. So, the most straightforward motivation for quantization is to shrink file sizes in order to use it on mobile and IoT devices.

Quantization can also reduce the computational resources that we need to do inference calculations. Because we will use a lower precision inputs and outputs, so

this will help us run models faster. Also, less computing power will be needed to run the task, which means a lot of energy will be saved. This is very important on mobile and tiny devices. There is no need to say that there are still a lot of embedded systems that can't run floating-point efficiently or even totally.

In terms of interpretability, there are some effects which may be imposed on the model after quantization. This means that it's hard to evaluate whether transforming a layer was going in the right or wrong direction. Mobile and embedded devices have limited computational resources, which means we need carefully adjust the balance between accuracy and efficiency.

Doing quantization during training or after the model has been trained are both available methods to use.

Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency with little degradation in model accuracy. TensorFlow Lite can do the quantization step for us if we use it to converting a tflite file. It's easy to use since it's integrated into the TFLite converter directly. What post-training quantization basically does is to convert, or more precisely, quantize the weights from floating point numbers to integers in an efficient way. By doing this, we can gain up to three times lower latency without taking a major hit on accuracy. With the default optimization strategy, the converter will do its best to apply a post-training quantization, trying to optimize the model for both size and latency.

Another method is quantization-aware training. By contrast of post-training quantization, quantization-aware training applies quantization to the model while it is being trained. The core idea is that quantization aware training simulates low precision inference time computation in the forward pass of the training process. By inserting fake quantization nodes, the rounding effects of quantization are assimilated in the forward pass, as it would normally occur in actual inference. The goal is to fine-tune the weights to adjust for the precision loss. If fake quantization nodes are included in the model graph at the points where quantization is expected to occur, for example, convolutions. Then in the forward pass, the flood values will be rounded to the specified number of levels to simulate the effects of quantization. This introduces the quantization error as noise during training and is part of the overall loss which the optimization algorithm tries to minimize. Therefore, the model learns parameters that are more robust to quantization.

2.5.2 Pruning

Another method to increase the efficiency of models is to remove parts of the model that did not contribute substantially to producing accurate results. This is referred to as pruning. It’s critical for mobile and IoT devices to reduce storage and computing

cost that means we have to reduce the number of weights as much as possible. Pruning as a biologically inspired concept, can effectively reduce the number of parameters and operations involved in generating a prediction by removing network connections. Therefore, the overall parameter count in the network will be reduced along with the storage and computing cost.

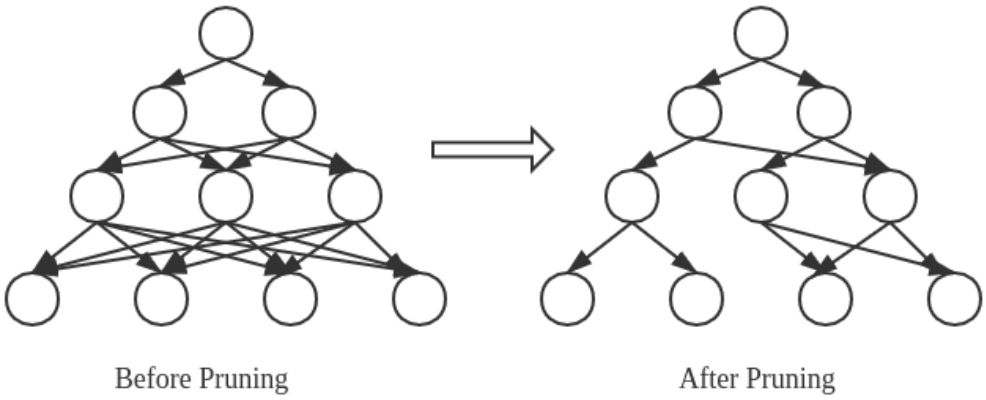


Figure 2.5.2: Pruning process

Networks usually look like the left one, where every neuron has a connection to every neuron in its next layer. That means multiplying a lot of floats numbers which cost a lot of computing resources. So, if we can let neurons connect fewer others and do fewer complex floats multiply, but also without losing too much of accuracy. We can gain a huge performance leap. And that's the basic idea of pruning, which we called connection sparsity. As we said, it’s a biologically inspired concept, because in our brain, the activity of neurons isn’t dense. The neuron didn’t connect with all other neurons like the common neural network architectures do. It only connects with a very few others.

Reducing the number of parameters would have several benefits. A pruned network is not only smaller, but it is also faster to train and use. Where hardware is limited, such as in embedded devices are smart phones, speed and size can make or break a model. Also, more complex models are more prone to overfitting. In some sense, restricting the search space can also act as a regularizer. However, even when all that said, it's not a simple task since reducing the model's capacity can also lead to a loss of accuracy. As in many other areas, there is a delicate balance between complexity and performance.

There are some methods to do pruning, for example Optimal Brain Damage. The pruning process was based on the idea that using magnitude as an approximation for saliency to determine less useful connections. So, in every iteration, it will remove the least useful connection between neurons, and set it to zero. But one particular challenge is that when the prune network is retrained. It turned out that due to its decreased capacity, retraining was much more difficult. The solution to this problem

arrived later, along with an insight called the lottery ticket hypothesis. Therefore, people are still developing new method to solve these problems.

2.6 Dynamic Neural Network

In 1.3 we have discussed the advantages of dynamic neural networks. At present, the methods to realize a dynamic neural network mainly focus on three levels: Sample-wise, spatial-wise and grain-wise.

The changes to spatial-wise are mainly focused on Pixel Level, Region Level and Resolution Level. In terms of Pixel Level, we can make a dynamic architecture such as sparse convolution and dynamic parameter such as additional refinement. But for Region level, we can use dynamic transformation to dynamically transform the data. Also, we can use hard attention such as recurrent attention. In resolution level we can call adaptive scaling to dynamically scale the input or using Muli-scale. In Temporal-wise, it's usually deployed in the scene of video and text, so we won't talk about it because we are doing image classification in this project.

Sample-wise has the most discussions and methods. The first adjustment we can do is to create a Dynamic Architecture. This adjustment includes Dynamic Depth, which dynamically adjusts the network Depth. For a network, the more layers there are, the deeper the network is. Therefore, the more computation is required. But as we know, whether due to the strain of system resources, the rigor of inference delay requirements, or the weight difference between different layers, it is not necessary to perform a complete calculation on all layers. The first implementation is an Early Exiting intention to withdraw from subsequent layers of computation by prematurely suspending them. Another method is Layer Skipping, which, as its name implies, means Skipping over some layers with little weight, or with not bad effect, or layers with too much computation and low performance-cost ratio. In Dynamic Architecture, we can also dynamically adjust width to reduce the width of the network architecture by skipping neurons, branches and channels. Another alternative is to use Dynamic Routing, which uses multi-branch, Tree structures or other structures to Routing inside the network model.

As we said, these methods are all based on the architecture level. But we can also modify the parameter to get a Dynamic Parameter. First way is to do the Parameter Adjustment by Soft Attention on Weights or Kernel Shape Adaptation. This will change the parameter to another value. Or we can do Weight Prediction on a feature-based level or based on specific task. Also, we can use Dynamic Features on channel-wise or spatial-wise to change the activation or features dynamically.

Chapter 3: Methodology

3.1 Separate Tasks

Let’s start talking about our method with the aim of our project: “Do inference on data with different sub-network under different computing resource using MCUnet.” We can get the job we need to do by separate this sentence.

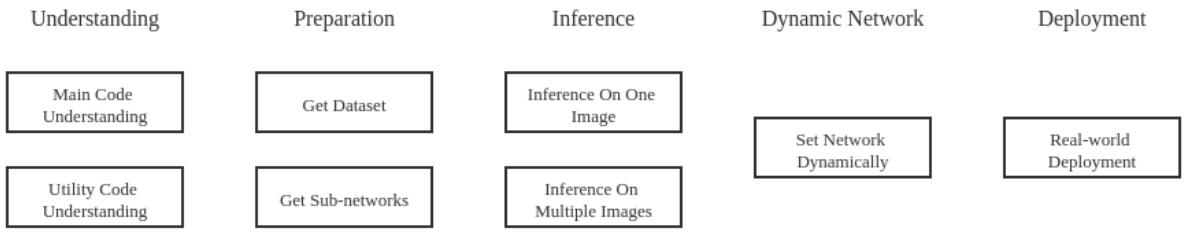


Figure 3.1: Task to finish

The first step is to obtain MCUnet's code and interpret it, to analyse and understand MCUnet's operating logic and implementation methods. Since this is a relatively unfriendly network structure and project for novices, we need to have a basic understanding of the implementation of each code block and find out the corresponding contents of the functions implemented by each code in the article, so as to further optimize and change it.

When we get a basic understanding of what the code is doing, we can find out there is no dataset included. So, we need to decide what data we should use for our inference process and obtain those data. As we know, there are a lot of datasets to choose for Machine Learning. But for the situation that we focused on; we only need the image dataset. In the available image classification dataset, we would like to choose some more common dataset instead of those not that famous. So, in this project, instead we locked our aimed dataset into CIFAR-10 and ImageNet. Usually, ImageNet is more famous and commonly accepted, but using CIFAR-10 provides quicker inference and validation. That’s because the ImageNet has too many high-quality images which is a heavy load for tiny device like IoT device. But we can use only a subset of ImageNet to validate our result, instead of using the whole dataset. And obviously, since we only focus on making the inference dynamically, so there is no training process needed in this project. Therefore, using a subset of ImageNet like ImageNet-1000 would be a good choice. Then we can head to the website and download them.

Then we need to get the sub-networks from MCUnet and use them for inference step. From the methods we mentioned in 2.2.2 Search Strategy, we choose to use

Evolution Search to search the sub-network from the MCUNet. That requires us to have a base understanding of how the MCUnet code works and what the restriction that this network can accept. We find that the Once For All has a good tutorial to start with, and then we can inference it to MCUnet. But there are some problems, the MCUNet didn't provide any kind of predictor or look up table, which makes the evolution search unable to continue. So, we need to find a method to get the sub-networks from MCUnet.

After we got the sub-networks, and we also got the data. We can start to approach inference. Like we start to do everything, we start with the easiest one. Try to predict the result from one image on one sub-network. That means we need to figure out how to use inference engine to inference the result. And what information we need to provide, what kind of information we can get after the calculation. Is there any argument can play with to implement our further prospective? When we figure it out, we can get the output prediction, and start to check whether the result is right or wrong.

Now, we can expand our data from one image data to a batch of image data. That means we need to get a more efficient way of inference, especially how to process batch data at same time. How to run batch image classification on the inference engine we have.

Because the aim is to run the inference dynamically, we want the algorithm to know what the available computing power looks like now. So, we need to let the algorithm able to get this information.

Finally, we can start to make the inference step dynamically. We do it by calling different sub-network to use under different computing resource available now. That means we need to decide under a certain percent of computing resource available, which network we would like to use. Furthermore, we can continue to even change the inference engine itself. To let it close some of the nodes and make the inference engine calculation step changeable. From the MCUnet that we have discussed in chapter 2, we know that the TinyEngine is included in MCUnet and should provide a lot higher efficiency that the original TensorFlow Lite inference engine. That's a huge gap between using these two inference engines on embedded devices. That means to better run dynamic neural network on small devices like IoT devices, we should choose TinyEngine to use. But unfortunately, the TinyEngine promised to be open-sourced, but it hasn't fulfilled the promises yet. We will keep an eye on the TinyEngine and do further optimisation on inference engine level later.

Above all those jobs, we want to really deploy some real-world application, for example, detection the movement we are doing, recognise the word we're saying, and

obviously, to do image classification on microcontroller. We will further talk about it in the 3.3 part.

3.2 Algorithm Workflow

From the jobs that we finished above; we can form the algorithm in a whole piece. In this case, we can define our algorithms workflow like below.

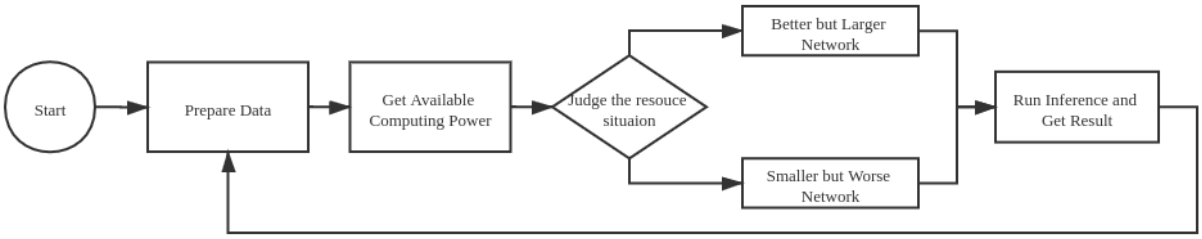


Figure 3.2: Algorithm Workflow

The basic idea of this dynamic image classification programme is based on the traditional inference.

We first prepare all the structure and needed operation and data we want to do inference, then get the free computing power in order to judge which network we will be using. And then use that network to do inference and output the result. Finally, loop this process again and again.

3.3 TinyML Application

The application we will cover include gesture detection, voice or wakeup word recognition and dynamic image inference.

In the gesture detection, we will generate a tiny network used to determine what kind of movement we’re doing now by input the accelerator's data. This will be deployed on smart phone and Arduino Nano 33 BLE sense, which only have 1Mb of Flash and 256Kb of SRAM.

Then it’s the voice or wakeup word detection/recognition. We will set up an algorithm that can distinguish Yes, No and other sounds. By inputing this kind of voice info as a sound spectrum, we can get an idea of how to run convolution operation on microcontrollers. And this will be deployed on the STM32F746-DISCO

board, which use STM32F746NG as an main processor, which has 1Mb of Flash and 320Kb of SRAM.

Finally, is the dynamic image inference, which will use our algorithm as a base to implement. It will automatically determine network to use to inference our image and output the corresponding inferencing time and result.

Chapter 4: Implementation

4.1 Code Understanding

We can describe our implementation based on the step in Chapter 3. First, we can download all the code for MCUNet from the address: https://github.com/mit-han-lab/tinyml/tree/master/mcunet. Based on this to analysis and understand MCUNet's operation logic and implementation methods. By reading and understanding the code, we can describe the MCUNet's code architecture with this graph:

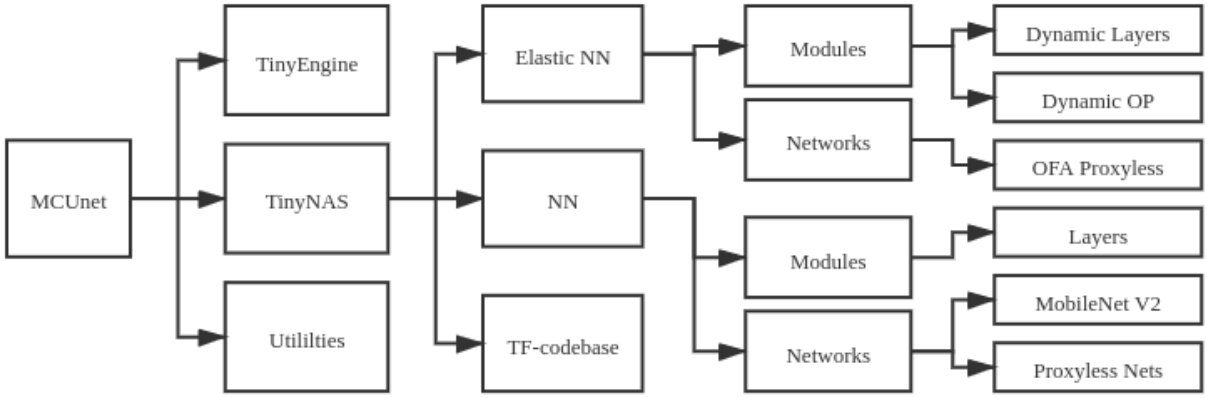


Figure 4.1: MCUNet code architecture

There are NN and Elastic NN in TinyNAS, each with its own modules and networks, associated with TensorFlow's codebase.

In NN, we define common static network structures and basic classes. The first is a module called Layer. In this Module, we set the set_layer_from_config() function, which sets the type and parameters of a layer in the neural network based on the given config parameters. At the same time, a number of classes of different Layer types are defined in this file. Due to the operation characteristics of MobileNet, we do not operate the complete convolution operation like the traditional convolution operation. Since we want to reduce the operation computation cost, we divide the convolution operation in each layer into two parts. The first part is called Depthwise Convolution. In this step, we perform hierarchical Convolution for the data of each Channel. Each Channel is calculated according to each Filter, after that, stacked together. The second part is the Pointwise Convolution. In this part, we perform 1*1*Nc Convolution operation on the results obtained after Depthwise Convolution operation and set the number of filters to the depth number we want to have. Finally, through these two steps, the convolution operation completed in the traditional way is disassembled, and the operation consumption and operation time are greatly reduced. 'My2DLayer', the distinctive 2DConv layer in MobileNet, runs operations within the

plane dimension, and all subsequent layers are defined according to this layer. Based on 'My2DLayer', we define 'ConvLayer' as the Convolution layer of Pointwise Convolution to carry out Convolution operation. The 'DepthConvLayer' is also a special deep layer in the MobileNet structure that performs depthwise convolution. There is also a 'PoolingLayer' operation for Pooling to reduce the height and width of the model. , 'IdentityLayer'[42], [43], passes the data intact. This is a pure Utility layer that needed to define and import to support other layers, it's also called a replicator. It's might be seemed as a useless layer, but kind get the same idea of 0 and Null and can also be performed as a duplicator. 'LinearLayer'[44], 'ZeroLayer', 'MBInvertedConvLayer' is further defined based on MyModule defined in Utilities. 'LinearLayer', defines the LinearLayer that performs linear operations, given an input, gets an output, which can be used as an Activation Layer. 'ZeroLayer' is, as the name suggests, an all-zero layer. 'MBInvertedConvLayer' is a concept unique to MobileNet V2, Because a inverted residual block connects narrow layers with a skip connection while layers in between are wide. With inverted residual blocks we do the opposite and squeeze the layers where the skip connections are linked.

At the same time, MCUNet constructs two normal (inelastic) networks in the networks in NN after completing the definition of all layers. One is MobileNetV2 from the Google team, and the other is Proxyless Nets from HAN LAB. They used these two networks as a basis for comparison and further dynamic modification.

In Elastic Neural Network, we define the Elastic structure of the Network. Modules defines the elastic module, and networks defines the elastic network. In Modules, we define the Dynamic Layers and Dynamic OP Modules. These are based on the methods in Once-for-All Networks.

In Dynamic Layers, we define three classes DynamicMBConvLayer, DynamicConvLayer, DynamicLinearLayer. In DynamicMBConvLayer, we can see that different layers are not treated in the same way. For DynamicMBConvLayer, only depth_conv in the middle is doing the job of changing kernel size. We use inverted_bottleneck to dynamically set the expand_ratio, here kernel_size is fixed at (1x1). Kernel_size is set with depth_conv. After the setting is complete, run the forward command. The elastic width is mainly modified to expand ratio. It is mentioned in the paper that L1 importance should be selected for weights. In the code, re_organize_middle_weights (self, expand_ratio_stage=0) function is used to sort the weights before each progressive shrinking training. After the sorting is completed, in the training process, just cut the data we want with a sequence range of self.conv.weight[:out_channel, :in_channel, :, :]. L1 importance was calculated in the function of DynamicMBConvLayer for weight reordering, and then the index of L1

order was obtained, and the weight order of each layer was adjusted according to the index. Adjust_bn_according_to_idx is used to adjust bn weight order.

In Modules, we also define dynamic OP. It is used to define the dynamic convolution kernel, wherein a learnable Transform matrix is set to compensate for the performance loss caused by picking out a small convolution kernel directly from the large convolution kernel. A linear transformation is performed on a small convolution kernel using the Transform matrix before training. And the conversion process is also gradual. So if we want to solve for a 3x3 convolution kernel, we solve for 5x5 from the 7x7 convolution, and do the transformation. Then take 3x3 out of 5x5, and do the transformation, and finallly get the final 3x3 convolution kernel. The Transform matrix size is (H*W)2, namely 9x9 (5->3) and 25x25(7->5). The initial method is scale_params ['%s_matrix' % param_name] = Parameter(torch. Eye (ks_small ** 2)), which generates a two-dimensional array with all 1s on the diagonal and all 0s on the rest.

In our MCUNet, TinyEngine, also called Inference Engine, is not open source yet. So we don't have a way to do code interpretation and further work right now.

Of course, in addition to the above code, there are some implementation of the task class code and utility class code, which will be mentioned in our algorithm implementation process.

4.2 Utility Understanding

In Utilities, we have written many tools to further reduce the amount of code in the body. We'll pick out some of the more important parts to explain.

In my_modules.py we define what Mymodule class is. This is a separate class that inherits Pytorch's own Module as the underlying class that defines various specialised networks. We also define what MyNetwork is, as the basement to define Proxyless and MobileNet. There are also many calculation methods defined for Batch Normalization, also known as BN. These include set_bn_param(), get_bn_param(), and replace_bn_with_gn(). At the same time, MyConv2d with Weight Standardization 2D layer is defined.

Common_tools include functions such as dictionary sorting, list splitting, list summation, and number to list. Sub_filter_start_end calculates the start and end of the kernel size. Min_divisible_value () to make sure V1 is divisible by N1, otherwise decrease V1.

There is also a set of tools for calculation for Batch Normalization. In the sampling training of super-network, every sub-network will influence the mean and variance var in BN layer during training. Therefore, two dictionaries were set up to record the mean and VAR of each BN under each batch. In order to achieve better performance, the subnet needs to recalculate mean and variance var of BN layer according to the structure of the subnet. calib_bn(subnet, path, net_config['r'][0], batch_size) is a function used to recasting the mean and var. It's structured the data_loader to be sent to set_running_statistics(model, data_loader, distributed=False) for count. forward_model is the model that been copied. Finally, we copy the runing_mean and runing_varresult from forward_model inference to model. Meanwhile, it has adjust_bn_according_to_idx(bn, idx) to adjust the batch normalization result based on the index from the sorting.

4.3 Get Dataset

As explained in Chapter3, we will use a subset of ImageNet for validation. We will use the Imagenet-1000 data set provided for us by the once-for-all Network. Can be downloaded from this address: https://www.kaggle.com/ifigotin/imagenetmini-1000

4.4 Get Sub-network

Usually, we will running the whole training process to get a super-network which contains all the possible network we will extract, and the finetune it. But as the computation power is limited for us, we can’t train the whole super-network again.

Also, by using evolution search to get the sub-network, we would need the accuracy info and efficiency info, especially the latency info which has been running on a real platform and use this information to consist a predictor to predict the sub-network's performance. But all these info were generated during the training process and running process. It’s would also be a problem to do evolution search without these info.

But fortunately, the HAN Lab has already done a lot of job for us, although they didn’t provide any super-network, or predictor, or even a look up table for latency. But they already directly provide a range of model fit for different kind of hardware situation. For example, we have a network config for MCU that have 256/320/512 kb ram and 1/2 mb flash or choose to use int4/int8 model. We also have the MobileNet V2 and ProxylessNet with width of 0.3/0.35 and resolution of 80/112/144/176.

These networks are all in json format. Therefore, we need to convert them into tflite file. In the tf_codebase we can find the generate_tflite.py file, we can use this to convert our json file into tflite file and let it able to be inference with TensorFlow Lite.

By far, we can have a variety of different sub-network that comes from the MCUnet and able to be used because they are in tflite file type.

4.5 Inference on One Image

As we got the tflite file, we can start to do the inference step. We first set an argument parser holds info like image position, tflite model file that will be executed, name of the file that containing labels, the number of threads to use etc.

Then we can start with define an interpreter use tflite.Interpreter(model_path=args.model_file, num_threads=args.num_threads). The network will be loaded into the interpreter and waiting to be executed. We can have a look about the input details and the output details of this specific sub-network by get_input/output_details().

As we got the input info, we need to transfer our data to that type, for here it's the resolution of the data. We have many methods to do so. But there will be a difference by using different method, therefore the final output image NumPy array maybe different, which means that the final accuracy actually can be infected by the method you use. For example, we can use image as a tool by Image.open(args.image).resize((width, height)) to resize the image, or using torchvision.transforms as a tool by using T.Resize(int(resolution * 256 / 224)), T.CenterCrop(resolution) to resize the image. The second way will actually reach a higher possibility distribution in the final result and get a higher accuracy.

After resize the resolution, we still need to modify the tensor we get to match the form of the input data requirement. We do it by add and change the dimension.

Then, we can set this tensor as input data, and set its computing mode. For example, interpreter.set_tensor(input_details[0]['index'], np.int8(input_data)), set the input_data as input with a type of int8 to match the input details.

Finally, we can start to invoke the interpreter, and get the output data. As we already get the output data's details, we decided to extract the top 5 result we got. We need to

use the output data to find the corresponding label of the input image and compare with the original image's label. Then we can compare the possibilities and labels we got for the top 5 results.

4.6 Inference on Batch Images

For the aim of increase the evaluation speed and efficiency, we would like to inference with batch of images. That's means the efficiency will be our top priority and we need to maximize the usage of the computing power we have.

So, instead of input only one image data per time, we load the whole dataset into our memory, and use a batch of images to inference at the same time.

To do so, we also need to set an arguments parser contains almost the same thing as the old one, but with a directory of the image folder's path, also along with the worker's number to set the data loader's size. Then cache the whole test set into memory for faster data loading, and get the images and targets as pairs, combined it together as a loader cache. Then use multi-processing for faster evaluation by using Pool from multiprocessing and set the thread number to 32.

For better visualisation of the validation process, we also include tqdm library to see the progress bar of the process. Then by using the `accuracy()` function in utility library, we can output the final accuracy of testing all test images.

4.7 Get Computing Resource

As long as we want to make the algorithm to change its network, we need to let it know the available computing resources we have.

We can use psutil library to do this. By using `psutil.virtual_memory().percent` we can get the current available memory usage. Also, we can get the CPU available percentage by output the `psutil.cpu_percent()`.

4.8 Set Sub-network Dynamically

After we have the computing resources available, we can freewheeling choose the network we want to use based on our own settings.

For example, we would like to use a better network to inference when the memory is sufficient, which is mcunet-512kb-2mb_imagenet.tflite at our project. Or when the CPU source is tense, change it to mcunet-256kb-1mb_imagenet.tflite to free CPU for other tasks running on the system, in order to make other tasks successfully being executed.

4.9 Application 1: Gesture Detection

In this application, we first collect the data from our phone or Arduino Nano 33 BLE Sense to get enough data for training. Then got them all well-labelled.

But the raw data is not suitable for the network to use. One of the most important things we can do in machine learning, is figure out which features to extract from our data to send to our model for training and, ultimately, inference. Many modern deep learning techniques do allow us to send raw data right to the model, and have the model figure out which features it cares about. But that often requires a lot more processing power than we have available in some of our embedded systems. We can do spectral analysis with our accelerators' data, which contains RMS (Root Mean Square), peak frequency and height, spectral power etc. By setting these down, we can extract a lot of nice features as the input data. In our case, it’s to extract the x y z accelerators’ data into 33 features.

And we can check the feature distribution, we choose x RMS, y RMS and z RMS here for better visualisation.

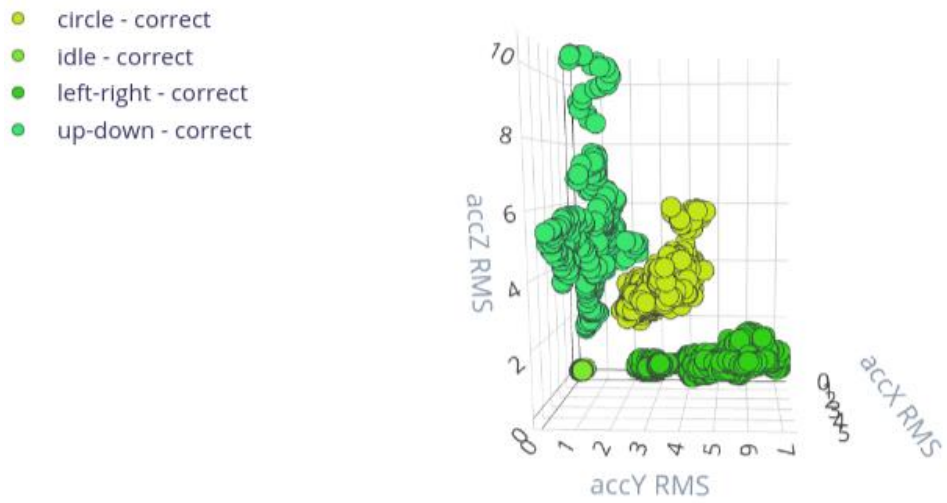


Figure 4.9: Feature Visualisation

Then we can start to train our model with these data. We use two dense layers here, first with 20 neurons second with 10 neurons, to form this network.

And finally, we transfer our network into optimized source code that we can run on our device, which is Arduino Nano 33 BLE sense here. And by using Arduino IDE, we can use the serial monitor to check the output prediction about the movement we're doing now.

4.10 Application 2: Wakeup Words Detection

We're going to build an embedded application that uses an 18 KB model, trained on a dataset of speech commands, to classify spoken audio. The model is trained to recognize the words "yes" and "no," and is also capable of distinguishing between unknown words and silence or background noise. And we run it with TensorFlow lite for microcontroller[45], [46] in order to prove the feasibility of convolution step on microcontrollers.

The labelled info for training is basically the same as the last one. The model takes in one second's worth of data at a time. It outputs four probability scores, one for each of these four classes, predicting how likely it is that the data represented one of them.
But in this time, we need to extract our feature by spectrogram, which are two-dimensional arrays that are made up of slices of frequency information, each taken from a different time window. By isolating the frequency information during pre-processing, we make the model to work easier. Because a spectrogram is a two-dimensional array, we feed it into the model as a 2D tensor. Then we feed them into the network, and get a .tflite file that we will use xxd to transfer to .cc and .h files.

In order to do so, we design an execution system worked in this sequence:

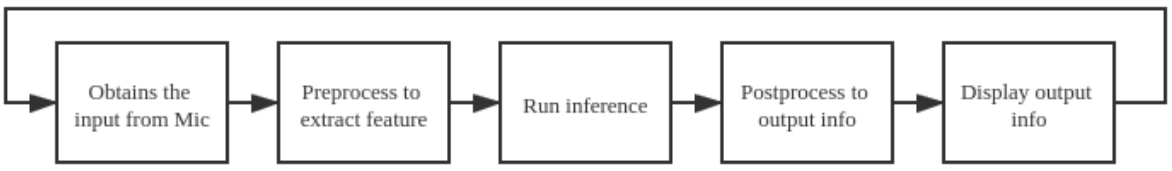


Figure 4.10: Implementation sequence

First, we set an audio provider to capture raw audio data from the microphone. Because we are using STM32F746DISCO here as hardware basement which already contained a microphone. So, we just include the source file that already provided for us to obtain the sounds info.

Then we pre-process the data for feature extraction. We process every data into spectrogram which is represented as a 2D array. In our spectrogram, each row represents a 30-millisecond sample of audio split into 43 frequency buckets. To create each row, we run a 30-ms slice of audio input through a fast Fourier transform (FFT) algorithm. By doing this step, we can analyse the frequency distribution of audio in the sample by getting an array of 256 frequency buckets. To build the entire 2D array, we combine the results of running the FFT on 49 consecutive 30-ms slices of audio, with each slice overlapping the last by 10 Ms. Then calculates how many of any existing slices it should keep.

The class RecognizeCommands is defined, along with a constructor that defines default values for a few things. The length of the averaging window (average_window_duration_ms). The minimum average score that counts as a detection (detection_threshold). The amount of time we'll wait after hearing a command before recognizing a second one (suppression_ms). The minimum number of inferences required in the window for a result to count.

Then we can start to inference with newly input data, note that we do continuously inference with the different chops of data in order to avoid the wrong detection for word that include the YES or NO sound. And we accumulated them into a confidence table for a certain time, and only when it's possibility is higher enough we will output the result as a word has been detected.

Finally, we can output the information to the screen use the build-in LCD.cc file to show info on STM32F746DISCO board's screen. We do this by showing a text message on the screen.

4.11 Application 3: Dynamic Image Inference

This should be the actual application to really deploy our algorithm on MCU to run inference. So, we will directly use the network that we already get from MCUNet.

For this project, the main thought is to get the system available resources and change the network, other things are pretty similar to the wakeup words detection application. But for our network, we can't simply fit two networks into the MCU because they are too big. And the biggest problem is that we actually don't have enough space for a single MCU to save images along with the label file. So, we just simply don't have enough space to do that.

Chapter 5: Evaluation & Analysis

We can evaluate our result in two ways.

As we said in Chapter 4, we implement the batch image evaluation to get the network's accuracy. So, we can first check the difference between different networks by comparing their output's inference time and accuracy.

Time and Accuracy	MCUnet-5fps	MCUnet-10fps	MCUnet-256kb-1mb	MCUnet-320kb-1mb	MCUnet-512kb-2mb
Time Cost	9s	4s	42s	54s	1min 32s
Top 1 Accu	50.3%	39.4%	59.9%	62.9%	69.8%
Top 5 Accu	75.3%	64.4%	83.6%	85.1%	90%

Table 5 (1): Using different MCUnet subnetwork

As we can see, using a lite network will cost much less time than the complex one, with only a little sacrifice of accuracy. It does a good trade-off between time cost and accuracy.

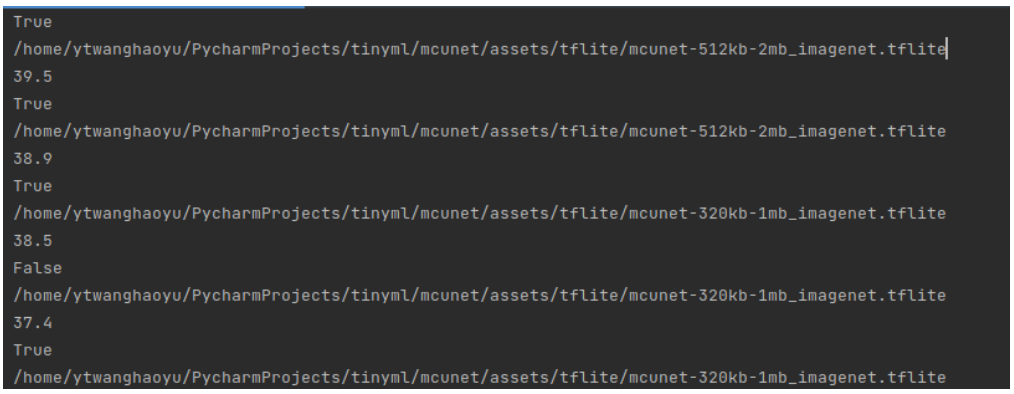
Also, we can compare MCUNet with other efficient network design, for example, ProxylessNet and MobileNet V2.

Time and Accuracy	Proxyless-w0.25-r112	Proxyless-w0.3-r176	Mbv2-w0.3-r80	Mbv2-w0.35-r144	MCUnet-256kb-1mb
Time Cost	7s	26s	5s	18s	42s
Top 1 Accu	45.1%	58%	39.6%	51%	59.9%
Top 5 Accu	69.6%	81.2%	66.1%	75.3%	83.6%

Table 5 (2): Compare MCUnet with other Networks

From the result we know that MCUnet is one of the best networks we can find to run, because at almost same situation and near time cost, it got a much higher accuracy.

Another way to evaluate is to change the usable memory status and see if the algorithm successfully changes the network it uses. We would do it by close some tasks on our system, trying to release the performance of our devices and left a little more computing power for the inference task.



Running log of dynamic inference

As we can see, at a certain time (when we're running tasks that require a lot of computational power), the interpreter output that the network has been changed from mcunet-320kb-1mb_imagenet.tflite to mcunet-256kb-1mb_imagenet.tflite. That means our algorithm is working successfully, and it will continue to run the inference step while the computing source is tense.

As to our application deployment, for our motion detection project. We get the final model testing results shows below. And it’s fairly to say it’s a good model to use.

	CIRCLE	IDLE	LEFT-RIGHT	UP-DOWN
CIRCLE	100%	0%	0%	0%
IDLE	0%	100%	0%	0%
LEFT-RIGHT	0%	0%	100%	0%
UP-DOWN	0%	0%	0%	100%
F1 SCORE	1.00	1.00	1.00	1.00

Table 5 (3): Confusion matrix

Things are also similar for the voice wakeup word detection. In this application by testing with examples of different sound of Yes and No and other sound. It shows an accuracy table below:

	YES	NO	UNKOWN	SILENCE
YES	93	0	0	0
NO	0	84	3	0
UNKOWN	7	16	195	5
SILENCE	0	0	2	5
# TEST	100	100	200	10

Table 5 (4): Test result matrix

From the table we can see that we got a descent accuracy in word detection, especially consider the network is only 14Kb, but the NO word is much worse than the Yes word. We think it’s might because the NO world is actually kind of the part of other words so it makes the algorithm confused with UNKOWN words.

Chapter 6: Future Work

As we can see from the evaluation result. We have a lot of jobs to do in the future to further optimise this algorithm.

First and the most important, once the TinyEngine is open-sourced, we can start to modify them to shut down some nodes or even layers in the networks. This can greatly reduce the computation power that running inference needs. By doing so, we should be able to hold the networks the same and cut the lowest requirement to run it and can run it dynamically without changing to a new network.

There are lot of actions that really need to be taken in this modification. For example, we need to let the inference engine know which node or layer is less important than others, which node is cost much more power than other, how much computing resource we can save from skipping this layer or node. And, in order to achieve better performance, when we need to shut down some neurons, when we need to change to a new network. All these things we will need to figure out.

We can also see from the results that although the network can be changed during this process, and the computing cost should drop a lot when it's under a tense computing power situation. But the PC's power is way more sufficient for the sub-network to run inference, and we can't use batch image to test the situation that the computing power is running low here because it won't change the network fast enough. That's basically not what we want to see.

Also, a lot of innovate idea just come out while doing this project, and they been inspired me a lot about the future step to do.

First one in obviously, run it on a tiny device like an IoT device. Because our algorithm's networks are all efficient enough and can run on IoT devices. For example, we would like to use some STM32 MCU to run our Dynamic MCUnet. This will show the result of our work much clearer.

Then is to generate a priority list or something like that along with the algorithm. With this list, the operation system will be able to know which task is more important than others. So, our algorithm will know when to give up some computing power to transfer it to some higher priority tasks. In order to do that, we must let the system know what task is under running and all tasks' priority, it's better to be in a list that the algorithm can refer to.

Also, we need to test how the algorithm works in real world. Would it be able to save energy for the IoT devices? Would it make the system running smoother for IoT devices? Or would it let the computation steps take shorter time but maintain a close accuracy. We need to do it by finishing the last application. We now have several ways in mind to solve these problems we mentioned earlier. For example, we can input data from computer one by one to save the space, and we can change the label file with only ID without corresponding name to save space. And most importantly, we can change our model to only detect human or other certain thing, instead of trying to classify everything in ImageNet.

So, our project still has a lot of job to do in the future and further optimize our algorithm.

Chapter 7: Conclusion

With this project finished, we got an basic understanding that dynamic neural network can be realised. We realise an inference step on image with a certain network and get the result. After that, we got the system's computing power information, and use it as a judge condition. And finally, successfully implement changing the network to use for inference while under different hardware resources.

Also, by our real-world application, we can deploy neural network on tiny devices like Arduino Nano 33 BLE sense and STM32F726NG. And successfully run some machine learning tasks on them. It proves the ability to run AI on MCU.

By using Dynamic MCUNet, we can easily let the machine to decide which network to use for a certain condition. It’s already showed a good result that it can run the network more flexible. After the modification with TinyEngine, it should have the power to lower the computation requirement. That means it will impressively lower the threshold of machine learning operations on low power-supply/computing power devices. That means we can change almost all devices in the world to AI devices.

References

[1] M. Bianchini and F. Scarselli, “On the complexity of neural network classifiers: A comparison between shallow and deep architectures,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 8, pp. 1553–1565, 2014, doi: 10.1109/TNNLS.2013.2293637.

[2] H. Jin, Q. Song, X. H.-P. of the 25th A. SIGKDD, and undefined 2019, “Auto-keras: An efficient neural architecture search system,” *dl.acm.org*, 2019, doi: 10.1145/3292500.

[3] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for Hyper-Parameter Optimization,” *Advances in Neural Information Processing Systems*, vol. 24, 2011.

[4] R. Bardenet, M. Brendel, … B. K.-… conference on machine, and undefined 2013, “Collaborative hyperparameter tuning,” *proceedings.mlr.press*, vol. 28, 2013, Accessed: Aug. 24, 2021. [Online]. Available: http://proceedings.mlr.press/v28/bardenet13.html

[5] L. Li, K. Jamieson, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization,” *Journal of Machine Learning Research*, vol. 18, pp. 1–52, 2018, Accessed: Aug. 24, 2021. [Online]. Available: http://jmlr.org/papers/v18/16-558.html.

[6] T. T. Joy, S. Rana, S. Gupta, and S. Venkatesh, “Hyperparameter tuning for big data using Bayesian optimisation,” *Proceedings - International Conference on Pattern Recognition*, vol. 0, pp. 2574–2579, Jan. 2016, doi: 10.1109/ICPR.2016.7900023.

[7] T. Elsken, J. H. Metzen, and F. Hutter, “Neural Architecture Search: A Survey,” *Journal of Machine Learning Research*, vol. 20, pp. 1–21, 2019, Accessed: Aug. 25, 2021. [Online]. Available: http://jmlr.org/papers/v20/18-598.html.

[8] M. Wistuba, A. Rawat, and T. Pedapati, “A Survey on Neural Architecture Search,” May 2019, Accessed: Aug. 25, 2021. [Online]. Available: https://arxiv.org/abs/1905.01392v2

[9] “Understanding Search Spaces | Coursera.” https://www.coursera.org/learn/machine-learning-modeling-pipelines-in-production/lecture/bYLkA/understanding-search-spaces (accessed Aug. 24, 2021).

[10] A. Agnihotri and N. Batra, “Exploring Bayesian Optimization,” *Distill*, vol. 5, no. 5, p. e26, May 2020, doi: 10.23915/DISTILL.00026.

[11] K. Stanley, *Efficient evolution of neural networks through complexification*. 2004. Accessed: Aug. 24, 2021. [Online]. Available: https://search.proquest.com/openview/9c5d2693f2073b76320dc1f3728e24ae/1?pq-origsite=gscholar&cbl=18750&diss=y&casa_token=VzCrnz_oxogAAAAA:_yVbOTk3rJ2FrVRjaz7a7VLtexldvKvYY4RMKSZPyGuI7dACIGVa0eI5mdpjDSXtK2TcL1TF

[12] V. Maniezzo, “Searching among Search Spaces: Hastening the genetic evolution of feedforward neural networks,” *Artificial Neural Nets and Genetic Algorithms*, pp. 635–642, 1993, doi: 10.1007/978-3-7091-7533-0_92.

[13] B. Zoph and Q. v le Google Brain, “NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING”.

[14] A. Klein, S. Falkner, J. Springenberg, and F. Hutter, “Learning curve prediction with Bayesian neural networks,” 2016, Accessed: Aug. 26, 2021. [Online]. Available: https://openreview.net/forum?id=S11KBYclx

[15] J. D.-I. transactions on Aerospace and undefined 1964, “Learning curve approach to reliability monitoring,” *ieeexplore.ieee.org*, Accessed: Aug. 26, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4319640/?casa_token=QX9VQCtSc8oAAAAA:7XaQ0ZgdlAY3l09QDtq0vvbAqfclbUnvedLQb8N4J38UtIOoFin4x4aq2eDAbYa2Gi0kDHn6

[16] C. Liu *et al.*, “Progressive Neural Architecture Search.” pp. 19–34, 2018. Accessed: Aug. 25, 2021. [Online]. Available: http://github.com/tensorflow/

[17] T. Wei, C. Wang, Y. Rui, and C. W. Chen, “Network Morphism,” *33rd International Conference on Machine Learning, ICML 2016*, vol. 2, pp. 842–850, Mar. 2016, Accessed: Aug. 25, 2021. [Online]. Available: https://arxiv.org/abs/1603.01670v2
[18] “Amazon SageMaker Autopilot | Amazon SageMaker.” https://aws.amazon.com/sagemaker/autopilot/ (accessed Aug. 26, 2021).
[19] “Automated Machine Learning | Microsoft Azure.” https://azure.microsoft.com/en-in/services/machine-learning/automatedml/ (accessed Aug. 26, 2021).
[20] “Cloud AutoML Custom Machine Learning Models | Google Cloud.” https://cloud.google.com/automl (accessed Aug. 26, 2021).
[21] P. Warden and D. Situnayake, “TinyML Tensorflow Lite”.
[22] P. Warden and D. Situnayake, “TinyML : machine learning with TensorFlow Lite on Arduino and ultra-low-power microcontrollers”.
[23] “Home | tinyML Foundation.” https://www.tinyml.org/ (accessed Sep. 15, 2021).
[24] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” Apr. 2017, Accessed: Aug. 26, 2021. [Online]. Available: https://arxiv.org/abs/1704.04861v1
[25] B. Koonce, “MobileNetV3,” *Convolutional Neural Networks with Swift for Tensorflow*, pp. 125–144, 2021, doi: 10.1007/978-1-4842-6168-2_11.
[26] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” pp. 4510–4520, 2018.
[27] A. Howard *et al.*, “Searching for MobileNetV3.” pp. 1314–1324, 2019.
[28] B. Wu, X. Dai, P. Zhang, Y. Wang, … F. S.-P. of the, and undefined 2019, “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search,” *openaccess.thecvf.com*, Accessed: Aug. 24, 2021. [Online]. Available: http://openaccess.thecvf.com/content_CVPR_2019/html/Wu_FBNet_Hardware-Aware_Efficient_ConvNet_Design_via_Differentiable_Neural_Architecture_Search_CVPR_2019_paper.html
[29] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” *7th International Conference on Learning Representations, ICLR 2019*, 2019.
[30] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, “Once-for-All: Train One Network and Specialize it for Efficient Deployment,” Aug. 2019, Accessed: Aug. 24, 2021. [Online]. Available: http://arxiv.org/abs/1908.09791
[31] W. Lou, L. Xun, A. Sabet, J. Bi, … J. H.-P. of the, and undefined 2021, “Dynamic-OFA: Runtime DNN Architecture Switching for Performance Scaling on Heterogeneous Embedded Platforms,” *openaccess.thecvf.com*, Accessed: Aug. 24, 2021. [Online]. Available: https://openaccess.thecvf.com/content/CVPR2021W/ECV/html/Lou_Dynamic-OFA_Runtime_DNN_Architecture_Switching_for_Performance_Scaling_on_Heterogeneous_CVPRW_2021_paper.html
[32] “Introducing the Model Optimization Toolkit for TensorFlow | by TensorFlow | TensorFlow | Medium.” https://medium.com/tensorflow/introducing-the-model-optimization-toolkit-for-tensorflow-254aca1ba0a3 (accessed Aug. 25, 2021).
[33] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, “IOS: Inter-Operator Scheduler for CNN Acceleration,” *Proceedings of Machine Learning and Systems*, vol. 3, pp. 167–180, Mar. 2021, Accessed: Aug. 27, 2021. [Online]. Available: https://github.com/mit-han-lab/inter-operator-scheduler.
[34] I. Fodor, “A survey of dimension reduction techniques,” 2002, Accessed: Aug. 26, 2021. [Online]. Available: https://cs.nju.edu.cn/_upload/tpl/01/0b/267/template267/zhouzh.files/course/dm/reading/reading03/fodor_techrep02.pdf
[35] Y. Ma, L. Z.-I. S. Review, and undefined 2013, “A review on dimension reduction,” *Wiley Online Library*, vol. 81, no. 1, pp. 134–150, Apr. 2013, doi: 10.1111/j.1751-5823.2012.00182.x.

[36] P. Cunningham, “Dimension reduction,” *Cognitive Technologies*, pp. 91–112, 2008, doi: 10.1007/978-3-540-75171-7_4.
[37] D. DeMers, G. C.-A. in neural information processing, and undefined 1993, “Non-linear dimensionality reduction,” *Citeseer*, Accessed: Aug. 25, 2021. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.893.8791&rep=rep1&type=pdf
[38] “Everything you did and didn’t know about PCA · Its Neuronal.” http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/ (accessed Aug. 25, 2021).
[39] J. Shlens, “A Tutorial on Independent Component Analysis”.
[40] J. Shlens, “A Tutorial on Principal Component Analysis”.
[41] M. Raymer, W. Punch, … E. G.-I. transactions on, and undefined 2000, “Dimensionality reduction using genetic algorithms,” *ieeexplore.ieee.org*, Accessed: Aug. 25, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/850656/?casa_token=Bnv-ia5I56UAAAAA:3drSpzWYnYReNnHdgXN5Jg-5r6xnyT-OGw9fw8CVaKXKsBRLbyTI_euCP8Gq98xthvF_X5Tn
[42] M. Hardt and T. Ma, “Identity Matters in Deep Learning,” *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Nov. 2016, Accessed: Aug. 28, 2021. [Online]. Available: https://arxiv.org/abs/1611.04231v3
[43] K. He, X. Zhang, S. Ren, and J. Sun, “Identity Mappings in Deep Residual Networks”, Accessed: Aug. 28, 2021. [Online]. Available: https://github.com/KaimingHe/
[44] “Linear Neural Networks - MATLAB & Simulink - MathWorks 中国.” https://ww2.mathworks.cn/help/deeplearning/ug/linear-neural-networks.html (accessed Aug. 28, 2021).
[45] P. Warden and D. Situnayake, “TinyML Tensorflow Lite”.
[46] “TensorFlow Lite for Microcontrollers.” https://www.tensorflow.org/lite/microcontrollers (accessed Sep. 15, 2021).

Appendices