DISS. ETH NO. 28024

TOWARDS ON-DEVICE INTELLIGENCE

Submitted to obtain the title of

DOCTOR OF SCIENCES ETH ZURICH
(Dr. sc. ETH Zurich)

Presented by

XIAOXI HE

M.Sc. ETH Zurich

born on 24.06.1994

Accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Mi Zhang, co-examiner

2022

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 194

Xiaoxi He

Towards On-Device Intelligence

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. 28024

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Mi Zhang, co-examiner

Examination date:

献给亲爱的父母

Acknowledgments

Table of Contents

Abstract

Enabled by artificial neural networks, deep learning has become the state-of-the-art machine learning method for artificial intelligence. In recent years, we have witnessed successful applications of deep learning systems in natural language processing, audio processing and visual data processing. These modern deep learning systems are fuelled by a large amount of data, often collected by mobile and embedded edge devices existing in our everyday life. In the past, such edge devices used to upload the collected data to the cloud, where deep learning systems were deployed for data processing. Due to the onboard resource constraints, it was considered infeasible or impractical to deploy deep learning models directly on edge devices. However, recent developments in hardware technologies have made edge devices increasingly powerful and opened up the chance for deploying deep learning systems on-device. We are at the beginning of a new era: *On-Device Intelligence*, as data are collected, processed and inferred locally on-device. Mobile and embedded edge devices, such as smartphones, wearables and automobiles, can be made more intelligent with the help of deep learning systems and enhance all aspects of our lives.

However, deploying a modern deep learning system on edge devices is still challenging, even with the increasingly powerful hardware. Many state-of-the-art deep learning models are too compute- and memory-intensive to execute on resource-constrained edge devices, even only for inference. To this end, many techniques have been developed to compress and optimise deep learning models, reducing the memory and computation requirements while preserving their inference accuracy. However, existing model compression and optimisation techniques do not cover multi-model deep learning systems, in which multiple deep neural networks for correlated tasks are performed continuously and concurrently. This can be seen in many artificial intelligence application scenarios. Hence we introduced the concept of *multi-model compression*, which aims at compressing and optimising such multi-model deep learning system for on-device deployment. We distinguish between three types of multi-model deep learning systems, which are covered by two multi-model compression methods: *weight sharing* and *neuron merging*.

Besides inference, it is often desired to train deep learning models on-device. However, modern deep learning models are even more compute- and memory-intensive during training. Moreover, the limited available data collected by a single edge device makes the training even harder. Existing techniques like meta-learning provide deep learning systems with the ability to train effectively with a limited amount of data. Yet, the trained models are often over-parameterised and have high memory and computation requirements. To this end, we provide

a solution to reduce the memory and computation cost of meta-learned deep models in this dissertation.

The main contributions of this dissertation are as follows.

- Multi-Task Zipping (MTZ), a weight sharing based network merging framework designed to automatically merge correlated, pre-trained deep neural networks. Central in MTZ is a layer-wise neuron sharing and incoming weight updating scheme that induces a minimal change in the error function. MTZ inherits information from each model and demands light retraining to re-boost the accuracy of individual tasks. MTZ supports typical network layers (fully-connected, convolutional and residual) and applies to inference tasks with different input domains.
- Multi-Task Stitching (MTS), a novel graph rewriter for efficient multitask inference with weight-shared deep neural networks, such as those merged via our MTZ. MTS adopts a model stitching algorithm which outputs a single computational graph for weight-shared DNNs without duplicating any shared weight. MTS also utilises a model grouping strategy to avoid overwhelming the GPU when co-running tens of DNNs.
- Pruning-Aware Merging (PAM), a neuron merging based network merging scheme to construct multitask networks that can be effectively pruned via existing pruning schemes, and the computation of all task combinations can be minimised, which is often demanded by modern mobile applications.
- Adaptation-aware Network Pruning (ANP), a novel network pruning scheme that works with existing meta-learning methods for compact networks capable of learning with limited data. ANP uses a weight importance metric based on the sensitivity of the meta-objective rather than the conventional loss function and adopts approximation of derivatives and layer-wise pruning techniques to reduce the overhead of computing the new importance metric.

Zusammenfassung

Dank künstlicher neuronaler Netze hat sich Deep Learning zur modernsten maschinellen Lernmethode für künstliche Intelligenz entwickelt. In den letzten Jahren haben wir erfolgreiche Anwendungen von Deep-Learning-Systemen in der Verarbeitung natürlicher Sprache, der Audioverarbeitung und der Verarbeitung visueller Daten gesehen. Diese modernen Deep-Learning-Systeme werden durch eine große Menge an Daten gespeist, die häufig von mobilen und eingebetteten Endgeräten in unserem Alltag gesammelt werden. In der Vergangenheit haben diese Geräte die gesammelten Daten in die Cloud hochgeladen, wo Deep-Learning-Systeme zur Datenverarbeitung eingesetzt wurden. Aufgrund der eingeschränkten Ressourcen auf dem Gerät wurde es als undurchführbar oder unpraktisch erachtet, Deep-Learning-Modelle direkt auf Edge-Geräten einzusetzen. Die jüngsten Entwicklungen bei den Hardwaretechnologien haben jedoch dazu geführt, dass Edge-Geräte immer leistungsfähiger werden und die Möglichkeit bieten, Deep-Learning-Systeme auf dem Gerät einzusetzen. Wir stehen am Anfang einer neuen Ära: *On-Device Intelligence*. Die Daten werden lokal auf dem Gerät gesammelt, verarbeitet und abgeleitet. Mobile und eingebettete Endgeräte wie Smartphones, Wearables und Autos können mit Hilfe von Deep-Learning-Systemen intelligenter gemacht werden und alle Aspekte unseres Lebens verbessern.

Die Implementierung eines modernen Deep-Learning-Systems auf Edge-Geräten ist jedoch trotz der leistungsfähigeren Hardware immer noch eine Herausforderung. Viele hochmoderne Deep-Learning-Modelle sind zu rechen- und speicherintensiv, um auf ressourcenbeschränkten Edge-Geräten ausgeführt zu werden, und sei es nur für Inferenzen. Aus diesem Grund wurden viele Techniken entwickelt, um Deep-Learning-Modelle zu komprimieren und zu optimieren und so die Speicher- und Rechenanforderungen zu reduzieren, ohne die Genauigkeit der Schlussfolgerungen zu beeinträchtigen. Die bestehenden Verfahren zur Komprimierung und Optimierung von Modellen decken jedoch keine Deep-Learning-Systeme mit mehreren Modellen ab, bei denen mehrere tiefe neuronale Netze für korrelierende Aufgaben kontinuierlich und gleichzeitig ausgeführt werden. Dies ist in vielen Anwendungsszenarien der künstlichen Intelligenz zu beobachten. Daher haben wir das Konzept der Multi-Model-Komprimierung eingeführt, das darauf abzielt, solche Multi-Model-Deep-Learning-Systeme für den Einsatz auf Geräten zu komprimieren und zu optimieren. Wir unterscheiden zwischen drei Arten von Multi-Modell-Deep-Learning-Systemen, die durch zwei Multi-Modell-Kompressionsmethoden abgedeckt werden: *Gewichtsteilung* und *Neuronenfusion*.

Neben der Inferenz ist es oft erwünscht, Deep-Learning-Modelle auf dem Gerät

zu trainieren. Allerdings sind moderne Deep-Learning-Modelle beim Training noch rechen- und speicherintensiver. Darüber hinaus erschweren die begrenzten Daten, die von einem einzelnen Edge-Gerät gesammelt werden, das Training zusätzlich. Bestehende Techniken wie das Meta-Lernen bieten Deep-Learning-Systemen die Möglichkeit, mit einer begrenzten Menge an Daten effektiv zu trainieren. Allerdings sind die trainierten Modelle oft überparametrisiert und haben einen hohen Speicher- und Rechenbedarf. Aus diesem Grund bieten wir in dieser Dissertation eine Lösung zur Reduzierung der Speicher- und Rechenkosten von meta-gelernten tiefen Modellen.

Die Hauptbeiträge dieser Dissertation sind wie folgt.

- Multi-Task Zipping (MTZ), ein auf Gewichtsteilung basierendes Framework zur automatischen Zusammenführung korrelierter, vortrainierter neuronaler Netze. Im Mittelpunkt von MTZ steht ein schichtweises Neuronen-Sharing und ein Aktualisierungsschema für eingehende Gewichte, das eine minimale Änderung der Fehlerfunktion bewirkt. MTZ erbt Informationen von jedem Modell und erfordert ein leichtes Umlernen, um die Genauigkeit einzelner Aufgaben zu verbessern. MTZ unterstützt typische Netzschichten (vollverknüpfte Schichten, Faltungsschichten und Residualschichten) und ist für Inferenzaufgaben mit unterschiedlichen Eingangsdomänen geeignet.

- Multi-Task Stitching (MTS), ein neuartiger Graph-Rewriter für effiziente Multitask-Inferenz mit gewichtsgeteilten neuronalen Netzen, wie sie durch unsere MTZ zusammengeführt werden. MTS verwendet einen Modell-Stitching-Algorithmus, der einen einzigen Berechnungsgraphen für gewichtsgeteilte DNNs ausgibt, ohne dass ein gemeinsames Gewicht dupliziert wird. MTS verwendet auch eine Modellgruppierungsstrategie, um die GPU nicht zu überlasten, wenn Dutzende von DNNs gemeinsam ausgeführt werden.

- Pruning-Aware Merging (PAM), ein auf Neuronenfusion basierendes Netzwerkfusionsschema zum Aufbau von Multitasking-Netzwerken, die mit Hilfe bestehender Pruning-Schemata effektiv beschnitten werden können, wobei die Berechnung aller Aufgabenkombinationen minimiert werden kann, was von modernen mobilen Anwendungen häufig gefordert wird.

- Adaptation-aware Network Pruning (ANP), ein neuartiges Netzwerkbeschneidungsschema, das mit bestehenden Meta-Lernmethoden für kompakte Netzwerke arbeitet, die mit begrenzten Daten lernen können. ANP verwendet eine Gewichtungsmetrik, die auf der Sensitivität des Meta-Ziels anstelle der konventionellen Verlustfunktion basiert, und verwendet eine Annäherung der Ableitungen und schichtweise Pruning-Techniken, um den Aufwand für die Berechnung der neuen Wichtigkeitsmetrik zu reduzieren.

1

Introduction

Artificial neural network enabled deep learning (DL) has become one of the most potent data inference tools, if not the most, in the last decade. It has been successfully applied in many areas, including natural language processing, audio processing and visual data processing [1]. DL systems are fuelled by the large amount of data collected on mobile and embedded devices, which used to be highly resource-constraint in computing power, memory and energy. For example, a Cortex™-M4 system-on-chip (SoC), often found in microcontrollers, runs at 64 MHz with 256KB SRAM and consumes less than 0.1 watts. On the other hand, the DL systems powering these applications mentioned above are often compute- and memory-intensive. For example, VGG-16 models [2], widely used in visual data processing, contain over 130M parameters and easily consume gigabytes of run-time memory during training and inference. Therefore, executing DL models on mobile and embedded devices were considered infeasible or impractical: either a well-trained model is too large, such that it is impossible to be deployed on-device, or the largest model supported by the device does not have enough capacity, such that it yields unsatisfying inference accuracy.

However, recent developments in hardware technologies have made mobile and embedded devices increasingly powerful and efficient. As an example, smartphones and smartwatches now come with gigabytes of volatile memory and multi-core processors capable of hundreds of GFLOPS and still consume only a few watts [3]. On the other hand, new advances in DL techniques reduce DL model size and computation to a magnitude of ten or more. Therefore, it seems feasible and practical now to run DL models on-device.

We are at the beginning of a new era: *on-device intelligence*, as data are collected, processed and inferred locally on-device. Mobile and embedded devices, such as smartphones, wearables and automobiles, can be made more intelligent with the help of DL systems and enhance all aspects of our lives.

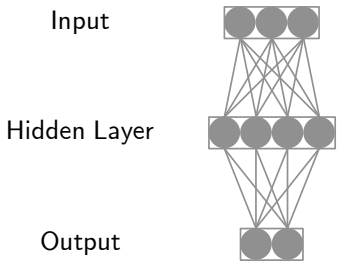


Figure 1.1 The computational graph of a neural network with one hidden layer, also known as a multilayer perceptron

1.1 Key Terminologies

As there are many keywords related to the subject of this dissertation, and the associated concepts are sometimes vague or confusing, we start this chapter with a few clarifications and definitions of important terminologies.

Artificial Intelligence (AI). In this dissertation, artificial intelligence refers to the intelligence shown by machines like computer systems instead of animals like humans. This is demonstrated by the capability to solve cognitive tasks associated with the "human mind", such as understanding speech and text, recognising objects and contexts from visual information, and playing strategic games. In this dissertation, AI is also referred to as *machine intelligence*.

Artificial Intelligence Tasks. In this dissertation, an AI task (henceforth task, with "artificial intelligence" implied) referred to a formulated problem to be solved by machines, thus demonstrating machine intelligence. A task consists of inputs and ground truths. A machine "performs/solves the task" by processing the inputs and giving outputs, which are supposed to be equal to or close to the ground truths. A formal definition of a task is given in Sec. 4.3.1.

Machine Learning (ML). Machine learning refers to one type of method developed for solving tasks and thus achieving AI. As opposed to knowledge-based approaches like the expert system, an ML method takes more of a data-driven approach and builds models based on sample data of a task. These models are then used to perform the task.

Training. Training refers to the process of building models in ML. As ML models are built on data, the training process requires sampled data of a task, known as *training data*. Training data consist of input data and labels sampled from the task's inputs and ground truths.

Inference. Inference refers to the process of using an ML model to solve a task. After the training process, the trained ML model is fixed and ready to take inputs and compute outputs for the task.

Deep Learning (DL). Deep learning is a branch of machine learning methods which uses *artificial neural networks* as its underlying model to solve tasks.

Artificial Neural Network (NN). Artificial Neural Networks (henceforth NNs, with "artificial" implied) are computational models initially inspired by biological neural networks inside animal brains. An NN can be described using a computational graph as illustrated in Fig. 1.1. The vertices are called *neurons*, which are connected by weighted edges. A neuron is a computing unit that takes the signal from its incoming edges, processes the signals based on the weights and pre-defined activation functions, and then passes the output to the next neuron. During the training process, the connection weights and graph topology are sometimes adjusted. Note that neural networks with more than one hidden layer are often referred to as deep neural networks (DNNs). The mathematical formulation of the computational graph is further discussed in Sec. 4.3.1.

1.2 Edge vs. Cloud Intelligence

As discussed at the beginning of this chapter, running DL models on mobile and embedded devices was considered infeasible or impractical due to the high demand for DL models and the resource constraints on-device. For example, a VGG-16 model [2] requires around 1 GB of run-time memory during inference and about 14 GB during training. Back in 2007, the first generation of the iPhone had only 128 MB of eDARM. An immediate and viable solution to this problem is moving the intensive part of the computation to a centralised location, where resource constraints are lifted and computation is done rapidly.

Cloud computing, an advanced form of the centralised computing paradigm, has been widely used for AI applications. In an edge-cloud paradigm, edge refers to the devices located where data are generated and collected, which are often mobile and embedded devices with limited resources. The cloud refers to the centralised computing infrastructure where computing power can be considered unlimited compared to edge devices.

In this dissertation, *cloud intelligence* refers to the computing paradigm which requires uploading all collected data from the edge to the cloud, and data processing, including both training and inference, are conducted mainly on the cloud. If needed, for example, during inference, the model outputs are sent back to the edge after the computation is done.

However, edge devices that became increasingly powerful and efficient, along with DL models that are more compact and efficient, opened up the opportunity to move part of, or even the whole data processing, to the edge. For example, the 13th generation of the iPhone debuted in 2021 had 4 to 6 GB RAM, which indicates that the deployment of even an uncompressed VGG-16 for inference, which requires about 1 GB of run-time memory, would be possible.

In this dissertation, *edge/on-device intelligence* refers to the computing paradigm in which at least the inference is made completely on the edge devices. Edge intelligence is desired for many reasons, including low latency,

high availability and strict privacy:

Latency. Deep learning models are widely used in cyber-physical systems at the stage where inputs, such as visual or audio data, are processed. There are often other modules like control units relying on the outputs of the deep models [4]–[6]. Therefore, the latency of the deep models during inference must be low and stable. Edge intelligence eliminates the need to transmit data to the cloud and results back to the edge during inference, thus removing all delays caused by the communication. Of course, the computing power of edge devices is no comparison to the cloud, and the computation latency is therefore much lower on the cloud. Moreover, advances in communication technologies such as the 5G/6G networks and the wide availability of various WLAN and Bluetooth protocols can reduce the transmission latency. However, as of now, the end-to-end latency of edge intelligence is still often lower than that of cloud intelligence [7].

Availability. Cloud intelligence conducts inference on the cloud and thus requires the communication to be always available when performing tasks. However, machine intelligence is often required when communication is unstable or even unavailable. For example, autonomous vehicles adopt AI techniques to process information and make decisions. They are desired to be operable outside cities, where mobile networks are less established than in urban areas, or even in uncharted areas such as Mars, where transmitting data for real-time inference is currently infeasible. Other AI systems, such as those used in medical or military facilities, are desired to be robust and still operable when communication is unavailable in particular situations. Again, the advances in communication technologies may provide more reliable channels and more area coverage in the future, but the current best solution for the aforementioned application scenarios is still the edge intelligence [7]

Privacy. In many edge-cloud systems, edge devices are owned and operated by individual customers, whereas the cloud is owned and operated by organisations like companies or governments. This has brought up privacy-related concerns in the last few years. During inference, edge intelligence requires uploading neither the user data nor the inference results. Thus, user information is kept away from the cloud, and privacy is preserved. While many techniques and methods, such as secure multi-party computation, have been developed to maintain privacy in the cloud intelligence paradigm, keeping the entire inference process on the device is still a simple and currently the more viable solution.

1.3 On-Device Deployment of Deep Learning Models

Edge intelligence is not enabled solely by more powerful edge devices. Although devices like smartphones are now so powerful that it is even possible to directly execute some modern DL models, some other edge devices are still not capable

of executing them. For instance, VGG-16 models [2] are also adequate for tasks like audio data processing, which is a crucial requirement for many wearable devices like smartwatches. However, a typical smartwatch like an apple watch has less than 1 GB RAM, and this prohibits it from executing a VGG-16 model [2] which requires more than 1 GB run-time memory. To this end, many techniques have been developed to compress and optimise NNs, reducing the memory and computation requirements while preserving their inference accuracy. Existing model compression and optimisation methods are summarised into four steps:

Model Compression. The first step is model compression, the process of simplifying and optimising the topology and parameters of NNs [8]. Modern NNs are known to be over-parameterised, which is a necessity for proper training [9]. However, this also means that a well-trained NN can be compressed by removing redundant parameters without sacrificing inference accuracy. Popular techniques include neural network pruning [10]–[13] and knowledge distillation [14]. Network pruning methods remove redundant connections between neurons and thus reduce the number of weights. These methods usually rely on mathematical methods to assess the importance of each weight, such as Hessian-based [11], [12] or mutual-information based metrics [10], and consequently remove the unimportant ones. Knowledge distillation methods [14] use a master-student paradigm to train a compact model on the training data with the help of a large but well-trained model. As the intermediate results of the master model are used for the training of the student model, the training of the student model is much more efficient than the initial training of the master model. Therefore less over-parameterisation is needed for the student model to be well-trained. Both pruning and knowledge distillation are designed to optimise and compress the DL *model*.

Quantisation. After model compression, the memory footprint and computation of the neural network can be further reduced by reducing the number of bits for each weight parameter, i.e., quantisation [15], [16]. Most DL models are mathematical computation models with real number parameters, yet they are deployed on digital computers, which brings up the problem of numerical representation [17]. The number of bits for storing the model parameters can be reduced via quantisation without incurring a significant computation accuracy that affects the model's inference accuracy. Quantisation is therefore used to optimise and compress the *numerical representation* of the DL model.

Computational Graph Optimisation. After quantisation, high-level graph rewriting schemes, which take full advantage of high- and operator-level optimisations, can be used to optimise the computational graph for the hardware [18]. A computational graph of a DL model can be transformed into functionally equivalent graphs for optimisations, including operator fusion (fusing multiple small operations), constant-folding (pre-computing static graph parts), static memory planning pass (pre-allocating memory for each intermediate tensor), and data layout transformations (transforming internal

data layouts into back-end-friendly forms) [18]. In general, these methods can be understood as an optimisation of the *scheduling* of the DL model.

Operator-Level Optimisation and Code Generation. In the last step, further optimisations are done on the translator, compiler and assembler level in order to generate the most efficient executable for deployment [18]. This includes techniques like tensor expression formulations for automatic code generation, a schedule primitive for optimal parallelism on modern GPUs, and explicit memory latency hiding [18]. These techniques are used to optimise and accelerate the low-level execution of the DL model on concrete hardware.

The aforementioned existing techniques cover different system levels, but they are all designed with a single NN in mind, especially the model compression methods. On the other hand, modern AI applications often require the concurrent performance of multiple tasks, which consequently requires the execution of multiple DL models simultaneously. Deploying multiple DL models on-device is challenging and requires optimisations specific for such a situation, which will be discussed in this dissertation in detail.

1.4 Multi-Model Compression

AI-powered mobile applications increasingly demand *multiple* deep neural networks for *correlated* tasks to be performed continuously and concurrently on resource-constrained mobile devices such as wearables, smartphones, and drones [5], [6], [19]–[23]. Examples include wearable cameras that recognise objects and identify people for the visually impaired and drones that detect vehicles and identify road signs for traffic surveillance. While many pre-trained models for different inference tasks are available [2], [24], [25], it is often infeasible to deploy them directly on mobile devices due to their large memory footprints. Again we take the VGG-16 model [2], [26] as an example: each VGG-16 model requires more than 1GB of run-time memory during inference, packing 4 of such models easily strains mobile storage and memory on even the latest high-end smartphones, such as the iPhone 13 which has only 4GB RAM. And the aforementioned applications [5], [23] often require five to ten tasks at the same time, which can be effectively solved via VGG-16 models, but is infeasible for many mobile and embedded devices due to memory limitations.

Model compression and quantisation techniques discussed in Sec. 1.3 are effective approaches to radically reduce the size of a deep neural network without sacrificing its accuracy. However, all these existing proposals focus on *single-model compression*. Consequently, they generate sub-optimally compressed neural networks for multiple correlated inference tasks because there can still be notable redundancy across models due to task relatedness. For example, deep neural networks trained for different visual tasks tend to learn similar low-level features that resemble either Gabor filters or colour blobs [27]. Sharing

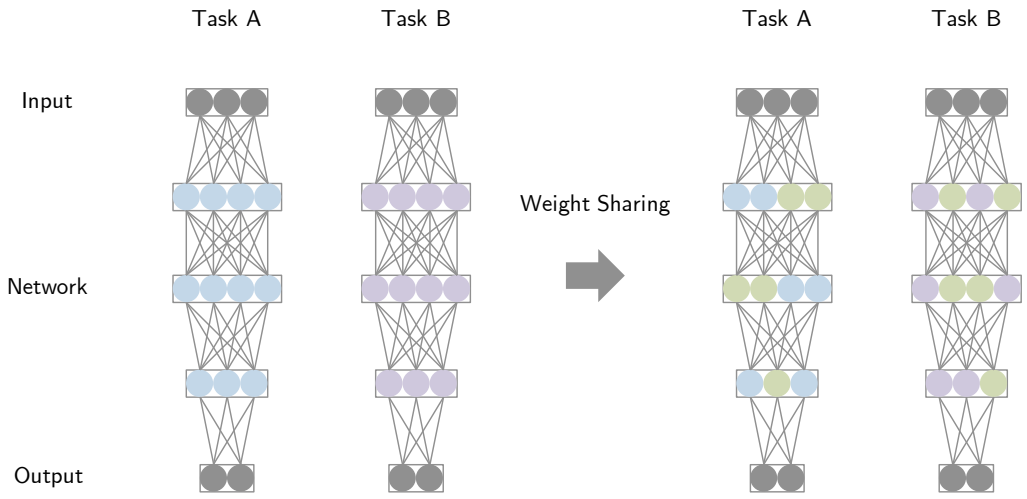


Figure 1.2 **Left:** an MMDL system with two independent models for two tasks; **Right:** after weight sharing, neurons coloured in green share the same incoming weights across the two NNs.

information *among tasks* holds the potential to further reduce the sizes of multiple correlated models without incurring a drop in task inference accuracy.

To this end, we introduced the concept of *multi-model compression*, which seeks effective and efficient information sharing mechanisms among pre-trained models for multiple tasks to reduce the size of the combined model without accuracy loss in each task. In the following part of this section, we are going to introduce three types of multi-model deep learning (MMDL) systems, and explain how multi-model compression is conducted on each of them.

1.4.1 MMDL: Independent Tasks

The first type of MMDL system consists of multiple independent NNs. An example is illustrated on the left of Fig. 1.2. This type of MMDL system is used for solving multiple *independent* yet *correlated* tasks. Each NN solves one single task and the output depends solely on the input of the same task. In other words, no two or more outputs have a dependency on the same input, and no single output has a dependency on more than one input. Hence the tasks are *independent*. However, the independent tasks can still be *correlated*, if they require similar operations during the processing of the task inputs. For instance, in a modern automobile, cameras are widely used to collect information for different tasks. A front camera facing the road may be used for object detection [2], while a cockpit camera facing the driver may be used for emotion recognition [28]. Here we have two independent tasks, but their inputs are both visual data and hence similar processing on the inputs, such as the computation of low-level features that resemble either Gabor filters or colour blobs, can be found in the NNs [27]. In this case, an MMDL system with two independent NNs can be deployed on the onboard processing unit for solving these two tasks.

As mentioned in Sec. 1.1, the computation of a nNN is largely characterised by its weights. For correlated tasks, as similar operations and computations are adopted, it has been shown that there are also numerical similarities between the weights of different tasks [23], [27]. Therefore, a viable solution to reduce the overall memory consumption for storing the weights is to share weights across models [29]. This procedure is referred to as *weight sharing*, which is illustrated in Fig. 1.2. The neurons coloured in green have the shared incoming weights across both models. Such weight sharing is normally done in two steps [29]: the first step is to identify groups of weights with similar functionality, which hold therefore the potential to be shared across models. The second step is to calculate the shared weights, which may not be identical to the original weights from any of the models, and then reconstruct the weight matrices for storing the weights. In this dissertation, an efficient and effective weight sharing method will be introduced in Chapter 2. The effect of sharing weights across multiple models is quite significant, especially when many models are required on the same device at the same time. For instance, it will be shown that on an MMDL system with 9 NNs trained for different visual tasks, the total number of weights can be compressed by $5\times$ with almost no loss in their inference accuracy, saving a significant amount of memory usage.

Weight sharing across models is extremely challenging, as NNs are notorious for their lack of explainability, which means that it is very hard to determine the functionality of individuals or groups of weights. One of the biggest challenges is that the weight matrices, which are used to store the computational graphs of NNs, can be very different even for isomorphic computational graphs. Isomorphic computational graphs have exactly the same functionality, hence finding groups of weights to be shared could be as hard as solving the subgraph isomorphism problem, if not harder because of the randomness during training. And the subgraph isomorphism problem is known to be NP-complete. Moreover, as discussed in Sec. 1.3, over-parameterisation is necessary for effectively training an NN [9], but over-parameterisation also increases the potential number of isomorphic subgraphs and make the problem even harder (in fact, some hypothesis even suggest that this is one of the reasons for over-parameterisation being a necessity for effective training). For example, if we train two individual NNs on the same task but with different initialisation of the weights, we could very possibly end up with two NNs that have completely different weight matrices. An effective weight sharing method should be able to fully share all weights across these two NNs because they have in effect the *same functionality*, which is solving the same task. It will be shown also in Chapter 2 that our weight sharing method passes this test, showing its effectiveness in finding similar functionality of the weights across models.

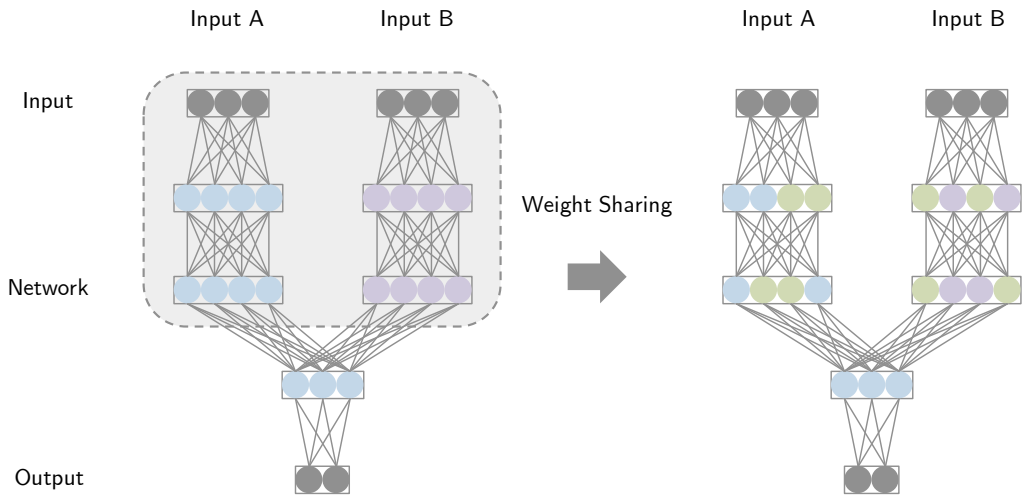


Figure 1.3 **Left:** a DL model designed for a task with two inputs from different domains. The part marked with a grey square with dotted lines is in effect the same MMDL system introduced in Sec. 1.4.1; **Right:** after weight sharing, neurons coloured in green share the same incoming weights across the two NNs.

1.4.2 MMDL: Task with Multiple Inputs

In some AI applications, tasks have inputs from multiple domains. For instance, [30] illustrated a task requiring emotion recognition based on both audio and visual data. As human beings, machines have shown the ability to combine information from both audio and visual domains for better recognition of human emotions. The recognition accuracy with data from both domains was higher than that from solely either audio or visual domain. This kind of task with inputs from different domains is also known as multimodal learning [28] and multi-view learning [31].

An example of a popular NN architecture designed for this kind of task is illustrated on the left of Fig. 1.3. In the first few layers, which are marked by a grey square with dotted lines, both inputs are processed independently of each other. At the later stage of the NN, the intermediate results are merged for the final output. The marked part of the NN can in effect be seen as the same MMDL system with independent models introduced in Sec. 1.4.1. Therefore, in this dissertation, this kind of DL model is also considered a type of MMDL model. Interestingly, as will be shown in Chapter 2, some of the weights in these separated layers, which process inputs from totally different domains, still possess similar functionality, and can be shared across the models with our weight sharing method.

One important difference between this type of MMDL against the one discussed in Sec. 1.4.1 is that the final output of the model has a dependency on both inputs, whereas in an MMDL system for independent tasks, every output depends only on one single input. This brings up a scheduling problem, as the end-to-end latency is affected by how quickly those separated layers are all

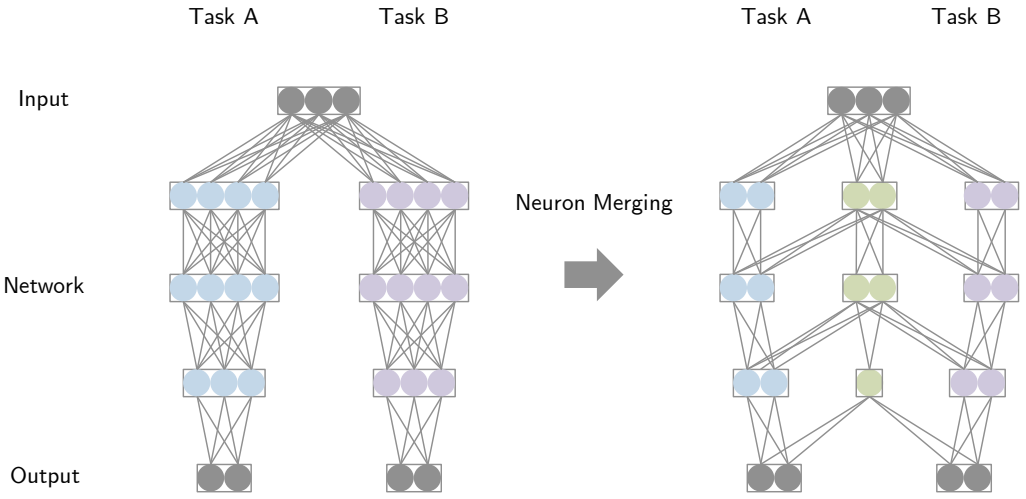


Figure 1.4 **Left:** an MMDL system with two tasks sharing the same input; **Right:** after neuron merging, neurons coloured in green are share between the two tasks.

executed. As mentioned before, the marked part in Fig. 1.3 can be seen as multiple independent models, with no cross-dependency between the layers for different inputs, hence they can be executed in parallel. But parallel execution of multiple NNs is not easy. The first problem is that, as will be shown in Chapter 3, most DNN graph rewriters (part of the computational graph optimisation discussed in Sec. 1.3) do not support the execution of MMDL models. Moreover, GPU vendors also provide inefficient APIs for parallel execution of multiple NNs at runtime. Only a few pilot works suggested graph rewriters [32] with cross-model graph fusion for parallel NN execution, but they do not support NNs with shared weights. Sharing weights across multiple models is crucial for saving memory usage and hence holds the potential of enabling many complicated AI applications, yet the efficient parallel execution of multiple NNs with shared weights is also desired for applications requiring low latency. To this end, in Chapter 3, we will discuss a novel graph rewriter for efficient parallel execution of NNs with shared weights in MMDL systems.

1.4.3 MMDL: Tasks Sharing the Same Input

The third type of MMDL system consists of multiple NNs sharing the same input. An example is illustrated on the left of Fig. 1.4 As opposed to the first type of MMDL discussed in Sec. 1.4.1, the tasks here share the same input and therefore the outputs of all the models depend on this input. A good example of this type of MMDL system can be found in [5]. Here the authors provided a wearable camera system designed for life-logging, on which five tasks are performed together on the visual data, including object detection, scene recognition, face detection, saliency computation and memorability inference.

Similar to what we discussed in Sec. 1.4.1, the tasks in this scenario are also correlated and weights in different models can be found to have similar

functionality and are hence shareable. However, we can do even more than weight sharing. As the models process the same input, not only the operations are shareable, but also the intermediate results. The sharing of the intermediate results is achieved by neuron sharing, as illustrated in Fig. 1.4. By sharing neurons across models, not only the number of weights is reduced, but also the number of arithmetic operations conducted during the execution of the DL models. Therefore, multi-model compression on an MMDL system with tasks sharing the same input reduces both memory usage and computation cost.

However, different to the scenario in Sec. 1.4.1, there is one more challenge for the third type of MMDL system: the asynchronous execution of the models. Take again the example from [5], in which the camera feeds the system with pictures at constant rates. But the results for each task may not be required at the same frequency. For example, object detection may be required every 3 frames, but scene detection may only be performed every 10 frames, as the scenes change slower than objects. Other tasks like face detection and memorability inference may even be performed on-demand. This brings a new challenge for a neuron shared MMDL system: part of the system should be able to shut down in order to save computation and energy. This requires network typologies like the one shown on the right of Fig. 1.4. When, for example, task A is not required when task B is performed, the neurons coloured in blue can be deactivated. Since there is no path from any of the blue neurons to the output of task B, the output of task B is completely independent of those blue neurons, and therefore the deactivation has no impact on the computation for task B. To this end, in Chapter 4, we will discuss a neuron merging framework satisfying the aforementioned requirements.

1.5 Efficient On-Device Adaptation

In Sec. 1.2, we define on-device intelligence as the computing paradigm in which *at least* the inference is done on the edge devices. This raises naturally the question: what about the training? In this dissertation, we will investigate a practical application scenario, where DL models are pre-trained before deployment, but require further training on-device. This setup is often referred to as on-device adaptation, which is to learn previously unseen tasks by updating a pre-trained initial model. This is desired in many on-device intelligence applications including personal drones, home robots and self-driving vehicles, since uploading newly collected data for model updating can be infeasible due to unstable wireless connections, limited bandwidth or privacy concerns, as discussed in Sec. 1.2.

The training of DL models is highly demanding in two perspectives: computation and data. Current DL models are mostly trained with gradient-based methods using backpropagation, which requires much more memory and computation compared to inference. Take VGG-16 [2] as an example again: the

training of a VGG-16 model can take up to 14 GB of runtime memory, while the inference tasks only about 1 GB. On the other hand, the amount of data that can be collected on individual edge devices is very limited. The DL models are often required to learn unseen tasks with even less than 5 samples, known as few-shot learning [33], [34]. Therefore, in order to enable on-device adaptation, the memory consumption during training needs to be reduced and the training needs to be effective with very limited data.

For reducing the memory usage, the existing method introduced in Sec. 1.3 may actually be helpful. In this dissertation, we focus on the pruning methods, which hold the potential to drastically reduce the number of parameters, and consequently the memory usage during training. For instance, [11] showed that pruning on a VGG-16 model can reduce its size to only 7.5% without hurting the inference ability, thus enabling the training computation for many edge devices.

For effective learning with a limited amount of data, one of the existing solutions is meta-learning, where the initial model for deployment is meta-trained on many different tasks, such that meta-knowledge is learned and the model is pre-prepared for fast adaptation with a limited amount of data [33], [34]. Of particular interest is Model-Agnostic Meta-Learning (MAML), a general gradient-based algorithm that meta-learns the weights of a given initial architecture, such that the meta-trained model excels at adaptation with only a few data [33].

However, existing pruning methods do not work in synergy with meta-learning methods like the MAML. As mentioned in Sec. 1.3, network pruning methods remove redundant connections between neurons and thus reduce the number of weights. These methods usually rely on mathematical methods to assess the importance of each weight and consequently remove the unimportant ones. The importance of the weights is assessed w.r.t. a known, single task, instead of some unseen tasks as in the meta-training procedure. Therefore, the network topology after pruning is not meta-learned, which means the topology is not prepared for training on limited data for an unseen task. To this end, in Chapter 5, we will disucss a novel pruning scheme that works with existing meta-learning methods, such that a compact DL model can be constructed for efficient on-device adaptation.

1.6 Outline and Contributions

In this dissertation, we will discuss some recent advances in improving the performance of and providing new possibilities for on-device intelligence. We will see novel weight sharing and neuron merging methods used for multi-model compression, which enables the deployment of MMDL systems on edge devices. We will also discuss a novel pruning method designed for on-device adaptation.

Chapter 2. In this chapter, a novel weight sharing method is introduced,

which is applied to the first and second types of MMDL system, discussed in Sec. 1.4.1 and Sec. 1.4.2. We propose Multi-Task Zipping (MTZ), a framework to automatically merge correlated, pre-trained deep neural networks via weight sharing. Central in MTZ is a layer-wise weight sharing and incoming weight updating scheme that induces a minimal change in the error function. MTZ inherits information from each model and demands light retraining to re-boost the accuracy of individual tasks. MTZ supports typical network layers (fully-connected, convolutional and residual) and applies to inference tasks with different input domains. Evaluations show that MTZ is able to drastically reduce the total number of parameters in targeted MMDL systems, hence saving memory usage and enabling the deployment on resource-constraint edge devices.

Chapter 3. As discussed in Sec. 1.4.2, it is a non-trivial problem to efficiently execute weight-shared neural networks, such as the networks merged by MTZ, on GPU enabled mobile and embedded edge devices. Following Chapter 2, we design Multi-Task Stitching (MTS), a novel graph rewriter for efficient multitask inference with weight-shared DNNs. MTS adopts a model stitching algorithm which outputs a single computational graph for weight-shared DNNs without duplicating any shared weight. MTS also utilises a model grouping strategy to avoid overwhelming the GPU when co-running tens of DNNs. Extensive experiments show that MTS is able to effectively exploit the parallel computing ability of modern GPUs and accelerates multitask inference by up to $6.0\times$.

Chapter 4. In this chapter, a novel neuron merging method is introduced, designed for the third type of MMDL system, discussed in Sec. 1.4.3. With the help of information theory, we will analyse the redundancy inside the third type of MMDL model and identify the optimal topology for merging. We also theoretically identify the conditions such that the merged network can be effectively pruned via existing pruning schemes and the computation of all task combinations can be minimised, which is often demanded by modern mobile applications. On this basis, we propose Pruning-Aware Merging (PAM), a heuristic network merging scheme to construct a multitask network that approximates these conditions. The merged network is then ready to be further pruned via existing network pruning methods. Evaluations with different pruning schemes, datasets, and network architectures show that PAM is able to effectively reduce both computation and memory costs.

Chapter 5. In this chapter, we investigate the on-device adaptation problem discussed in Sec. 1.5. We propose Adaptation-aware Network Pruning (ANP), a novel pruning scheme that works with existing meta-learning methods for a compact network capable of fast adaptation. ANP uses a weight importance metric that is based on the sensitivity of the meta-objective rather than the conventional loss function and adopts approximation of derivatives and layer-wise pruning techniques to reduce the overhead of computing the new importance metric. Evaluations show that ANP can work with meta-learning methods and provide up to 85% reduction in memory consumption.

2

Multi-Task Zipping: Multi-Model Compression via Weight Sharing

AI-powered mobile applications increasingly demand *multiple* deep neural networks for *correlated* tasks to be performed continuously and concurrently on resource-constrained mobile devices. This kind of deep learning system with multiple neural networks is referred to as multi-model deep learning (MMDL) system. As we discussed in Sec. 1.3, existing model compression and optimisation methods do not cover this MMDL scenario. To this end, we introduce multi-model compression, a series of methods designed for the compression, optimisation, and efficient execution of MMDL systems.

We have discussed in Sec. 1.4 three types of MMDL systems, which require two different multi-model compression methods: weight sharing and neuron merging. In this chapter, we first introduce a novel weight sharing method, which is applied to the first and second type of MMDL system, discussed in Sec. 1.4.1 and Sec. 1.4.2, respectively. We propose Multi-Task Zipping (MTZ), a framework to automatically merge correlated, pre-trained deep neural networks for cross-model compression. Central in MTZ is a layer-wise neuron sharing and incoming weight updating scheme that induces a minimal change in the error function. With this, MTZ is able to tackle the challenge of identifying similar functionalities among weights discussed in Sec. 1.4.1.

MTZ supports typical network layers (fully-connected, convolutional and residual) and applies to inference tasks with different input domains. Evaluations show that MTZ can fully merge the hidden layers of two VGG-16 networks with a 3.18% increase in the test error averaged on ImageNet for object classification and CelebA for facial attribute classification, or share 39.61% parameters between the two networks with $< 0.5\%$ increase in the test errors. The number of iterations to retrain the combined network is at least $17.8\times$ lower than that of training a single VGG-16 network. Moreover, MTZ can effectively merge nine residual networks for diverse inference tasks and models

for different input domains. And with the model merged by MTZ, the latency to switch between these tasks on memory-constrained devices is reduced by $8.71\times$.

2.1 Introduction

AI-powered mobile applications increasingly demand *multiple* deep neural networks for *correlated* tasks to be performed continuously and concurrently on resource-constrained mobile devices such as wearables, smartphones, and drones [5], [6], [19]–[22]. Examples include wearable cameras that recognise objects and identify people for the visually impaired and drones that detect vehicles and identify road signs for traffic surveillance. While many pre-trained models for different inference tasks are available [2], [24], [25], it is often infeasible to deploy them directly on mobile devices due to their large memory footprints. For instance, VGG-16 models for object classification [2] and facial attribute classification [26] both contain over 130M parameters. Packing multiple such models easily strains mobile storage and memory at inference time.

Model compression [8] is an effective approach to radically reduce the size of a deep neural network without sacrificing its accuracy by pruning unimportant operations (pruning) [10]–[12], [35] or reducing the precision of operations (quantization) [15]. However, all these proposals focus on *single-model compression*. Consequently, they generate sub-optimally compressed neural networks for multiple correlated inference tasks because there can still be notable redundancy across models due to task relatedness. For example, deep neural networks trained for different visual tasks tend to learn similar low-level features that resemble either Gabor filters or colour blobs [27]. Sharing information *among tasks* holds potential to further reduce the sizes of multiple correlated models without incurring drop in individual task inference accuracy.

We study information sharing in the context of *cross-model compression*, which seeks *effective* and *efficient* information sharing mechanisms among *pre-trained* models for multiple tasks to reduce the size of the combined model without accuracy loss in each task (see Fig. 2.1). A solution to cross-model compression is multi-task learning (MTL), a paradigm that jointly learns multiple tasks to improve the robustness and generalisation of tasks. However, most MTL studies use heuristically configured shared structures, which may lead to dramatic accuracy loss due to improper sharing of knowledge [36], [37]. Some recent proposals [38], [39] automatically decide "what to share" in deep neural networks. Yet deep MTL usually involves enormous training overhead [36]. Hence it is inefficient to ignore the already trained parameters in each model and apply MTL for cross-model compression.

In this chapter, we propose Multi-Task Zipping (MTZ), a framework which automatically and adaptively merges correlated and well-trained deep neural

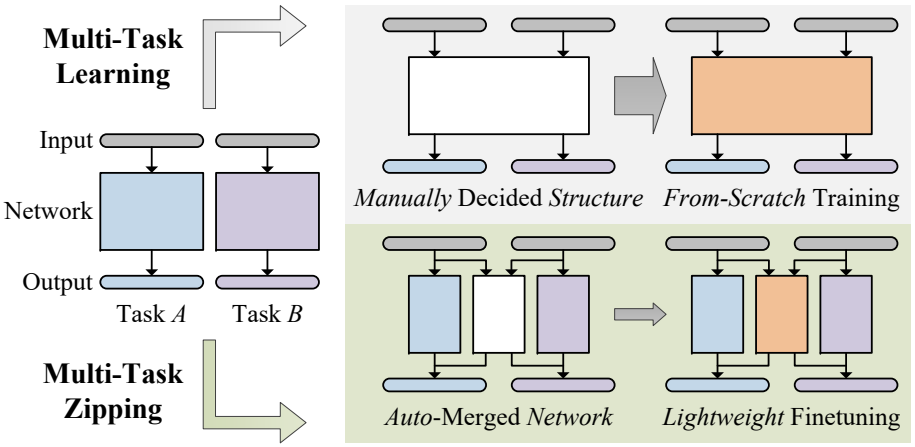


Figure 2.1 Differences between multi-task learning (MTL) and our multi-task zipping (MTZ) as solutions to cross-model compression (illustrated with two tasks). Given two pre-trained models, MTL manually decides the shared structures, and then retrain the multi-task model from scratch *i.e.*, the original weights of the individual models are discarded. In contrast, our MTZ automatically determines what to share, and only a lightweight finetuning is necessary to re-boost the accuracy on each task because the original weights of the individual models are either kept (for non-shared neurons) or analytically calculated (for shared neurons).

networks for cross-model compression, with the help of neuron sharing. It decides the optimal shareable pairs of neurons on a layer basis and adjusts their incoming weights such that minimal errors are introduced in each task. Unlike MTL, MTZ inherits the parameters of each model and optimises the information to be shared among models such that only light retraining is necessary to resume the accuracy of individual tasks. In effect, it squeezes the *inter-network redundancy* from multiple already trained deep neural networks. MTZ may be further integrated with existing proposals for *single-model compression*, which reduce the *intra-network redundancy* via network pruning [10]–[12], [35] or network quantization [15].

The contributions and results of this chapter are as follows.

- We propose MTZ, a framework that automatically merges multiple correlated, pre-trained deep neural networks. It squeezes the task relatedness across models via layer-wise neuron sharing, while requiring light retraining to re-boost the accuracy of the combined model. We also extend MTZ to support different layer types and tasks with different input domains. To the best of our knowledge, this is one of the first studies on cross-model compression for deep neural networks.

- MTZ managed to share 39.61% parameters between the two VGG-16 networks pre-trained for object classification (on ImageNet [40]) and facial attribute classification (on CelebA [41]), while incurring less than 0.5% increase in test errors. Even when all the hidden layers are fully merged, there is a moderate (averaged 3.18%) increase in test errors for both tasks. MTZ achieves

the above performance with at least $17.9\times$ fewer iterations than training a single VGG-16 network from scratch [2].

- MTZ can merge models of different input domains (*e.g.*, audio- and video-based models), and is able to share 90% of the parameters among nine ResNets on nine different visual recognition tasks while inducing negligible loss on accuracy. Furthermore, with the joint model merged by MTZ, the latency to switch between these inference tasks on memory-constrained devices can be reduced by $8.71\times$.

This chapter has made the following additional contributions:

- We enhance the theoretical analysis of MTZ by showing that the accumulated error at the output layer in our *layer-wise* neuron sharing is bounded (Sec. 2.3.5).
- We propose an optimised network zipping scheme for ResNets (Sec. 2.4.2.2 to support batch normalisation layers and Sec. 2.4.2.3 to support residual blocks).
- We empirically show that MTZ can support different input domains *e.g.*, audio- and image-based models (Sec. 2.5.3) and is scalable in merging more than two networks (Sec. 2.5.4). Experimental results show that MTZ can merge 9 ResNets pre-trained for diverse visual inference tasks, which reduce the model storage from $9\times$ to only $1.8\times$ of a single ResNet, with marginal loss in all the 9 inference tasks. In addition, MTZ can reduce the latency by $8.71\times$ when switching between the 9 inference tasks on memory-constrained embedded platforms.

In the rest of this chapter, we first review related work in Sec. 2.2, and then introduce our MTZ framework in Sec. 2.3 and its extensions in Sec. 2.4. We present the evaluations of MTZ in Sec. 2.5 and finally conclude in Sec. 2.6.

2.2 Related Work

MTZ compresses multiple well-trained deep neural networks of correlated inference tasks. It is relevant to research on multi-task learning and single-model compression. Our work belongs to the emerging field of cross-model compression and is complementary to resource scheduling of deep neural networks and edge-assisted inference.

2.2.1 Multi-Task Learning

Multi-task learning (MTL) jointly trains multiple correlated tasks to achieve higher accuracy than training each task individually. Determining "what to share" among tasks is a central issue in MTL, where can take place at different levels [36]. For MTL with neural networks, common techniques include hard or soft parameter sharing of the hidden layers [37]. Hard parameter sharing enforces sharing most or all of the parameters among all tasks while keeping a few task-specific output layers [42]. It causes notable accuracy drop when

many tasks are trained jointly [43]. In soft parameter sharing, individual tasks are connected via information sharing [44]. The two sharing schemes can also be combined for more flexible parameter sharing *i.e.*, adaptively sharing a subset of parameters in the hidden layers [39].

The shared topology in most MTL studies is heuristically configured, which may lead to improper knowledge transfer [27]. Only a few schemes [38], [39] optimise *what to share among tasks*, especially for deep neural networks. Our MTZ resembles these automatic shared structure optimisation studies for MTL in effect, but differs in objectives. MTL jointly trains multiple tasks to improve their generalisation and accuracy, while MTZ aims to compress multiple *already trained* tasks with mild training overhead. Specifically, MTZ inherits the parameters directly from each pre-trained network when optimising the neurons shared among tasks in each layer and demands light retraining.

2.2.2 Single-Model Compression

There have been various model compression proposals to reduce the size of a *single* neural network without incurring loss in accuracy [8]. Pruning-based methods compress a deep neural network by eliminating unimportant operations such as weights [11], [12] or neurons [10], [45]. Neuron-level pruning is more desirable since it leads to regular sparsity in the pruned networks, and thus avoids the need for customised hardware [8]. The memory footprint of a neural network can be further reduced by lowering the precision of parameters (network quantization) [15].

Unlike previous research that deals with the *intra-redundancy* of a single network, our work reduces the *inter-redundancy* among multiple networks. In principle, our method is a neuron-level *cross-model* pruning scheme. Our work may be integrated with single-model compression to further reduce the size of the combined neural network.

2.2.3 Cross-Model Compression

Cross-model compression aims to construct an accurate and compact multi-task neural network for efficient inference on resource-constrained platforms. Georgiev *et al.* [19] are the first to explore cross-model compression. They directly apply MTL techniques by heuristically configuring the shared structure and training the multi-task network from scratch. Our preliminary version [29] and NeuralMerger [20] are among the earliest studies to merge well-trained neural networks without training from scratch. NeuralMerger [20] utilises a joint encoding scheme for weight sharing, which can be understood as a cross-network *quantization*. Our technique is orthogonal to [20] since we focus on network *merging*. Neural weight virtualisation (NWV) [6] and ZipperNet [22] are two latest studies that explore merging for cross-model compression. NWV [6] shares all parameters among tasks and retrains to recover the accuracy *i.e.*,

hard parameter sharing. ZipperNet [22] relaxes the constraint by a layer-wise merging strategy, *i.e.*, all the parameters are shared till a given layer. In contract, MTZ allows partial merging in each hidden layer. In addition, ZipperNet [22] adopts a heuristic neuron similarity metric and only applies to convolutional layers. In contrast, our MTZ shares neurons and updates weights via sensitivity analysis, and our method supports not only convolutional layers, but also fully connected layers, batch normalisation layers and residual blocks. We compare the performance with NWV [6] and ZipperNet [22] in Sec. 2.5.

2.2.4 Resource Scheduling for Deep Neural Networks

Orthogonal to reducing the complexity of deep neural networks themselves, resource scheduling algorithms enables efficient on-device execution of deep neural networks. DeepX [46] is a software accelerator that splits deep neural networks into blocks to be executed across multiple co-processors. DeepEye [5] proposes to interleave the execution of convolutional layers and fully-connected layers from multiple deep neural networks to improve the runtime efficiency of multi-model execution. NestDNN [21] designs a dynamic model pruning and recovery scheme and a resource-aware runtime scheduler to adaptively select the best models and allocate them to the available resources to maximise the overall inference accuracy and minimise the overall latency of concurrently running deep neural networks. As with [5], [21], our work also focuses on optimising multiple deep neural networks. However, our approach is complementary, which aims to reduce the memory footprint of multiple models by enforcing neuron sharing rather than scheduling their executions.

2.2.5 Edge-Assisted Deep Inference

In addition to on-device execution, offloading is also a popular strategy to run deep neural networks in the era of edge computing [47]. Particularly, the memory- or computation-intensive portion of a deep model can be offloaded to the edge to meet the resource constraints on end devices. For example, DeepDecision [48] dynamically decides whether to execute the model on-edge or on-device according to the available resources. Neurosurgeon [49] explores the optimal layer to partition a deep neural network for collaborative execution between the edge and the device that minimises latency and energy consumption. EdgeDuet [50] runs a full model on-edge and a compressed version on-device and only uploads image tiles to the full model when necessary. EalgeEye [51] partitions the multiple-model pipeline for face identification both spatially and temporally and runs the partitions in parallel on both the edge and the device. Magnum [52] adopts a lightweight blockchain-based framework to enable transfer learning in industrial IoT applications.

Our work is complementary to model partition. On the one hand, cross-model compression can be combined with model partition schemes for higher efficiency

when running multiple tasks in edge-device collaborative inference. For instance, AMVP [53] proposes an adaptive scheduler that integrates single- and cross-model compression with model partition for multi-task video processing at the edge. On the other hand, model compression is preferable over model partition to applications where communication with the edge is prohibited due to data privacy or unreliable network connections [47], [54]–[56].

2.3 Layer-wise Network Zipping

This section explains the principles and details of our network zipping method with two feed-forward networks of dense fully connected (FC) layers. We discuss the extensions to other layers and settings in Sec. 2.4.

2.3.1 Problem Statement

Consider two inference tasks A and B with the corresponding *well-trained* deep neural networks M^A and M^B, *i.e.*, trained to a local minimum in error. We assume the same input domain and the same number of layers in M^A and M^B. Performing multiple correlated inference tasks on the same input domain is common in mobile applications (*e.g.*, face recognition, age and gender identification from a wearable camera [5], [6]; or speaker identification and ambient scene analysis from a smartphone microphone [19]). Note that our method also works for different input domains (see Sec. 2.5.3). The assumption on the same number of layers follows the practice in multi-task learning for ease of joint training [36]. Note that the models for different tasks can vary in the widths in their layers. Our goal is to construct a combined model M^C by sharing as many neurons between layers in M^A and M^B as possible such that *(i)* M^C has minimal loss in inference accuracy for the two tasks and *(ii)* the construction of M^C involves minimal retraining. As with other studies on cross-model compression [6], [19], [20], [22], the process to construct the combined model, *i.e.*, model merging and retraining, takes place offline on the cloud or the edge before model deployment. The combined model is then deployed to resource-constrained devices for accurate multi-task inference. Although extensive model training is affordable on the cloud/edge, it is still desirable to minimise the retraining overhead to allow fast model deployment and to serve more model merging requests at the same time.

2.3.2 Layer Zipping via Neuron Sharing

We take a layer-wise approach to the neuron sharing problem described in Sec. 2.3.1. This subsection presents the procedure of zipping the l-th layers $(1 \leq l \leq L-1)$ in models M^A and M^B given the previous $(l-1)$ layers of the two models have been merged (see Fig. 2.2).

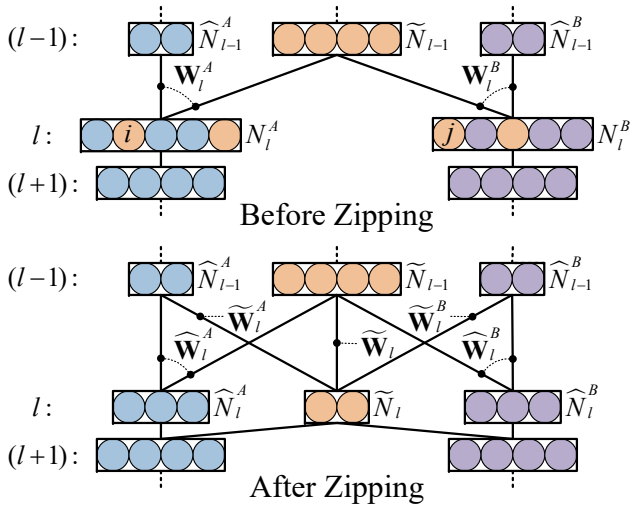


Figure 2.2 An illustration of neurons and the corresponding weight matrices before and after zipping the l-th layers of M^A and M^B.

Denote the input layers as the 0-th layers. The L-th layers are the output layers of M^A and M^B. Denote the weight matrices of the l-th layers in M^A and M^B as $\mathbf{W}_l^A \in \mathbb{R}^{N_{l-1}^A \times N_l^A}$ and $\mathbf{W}_l^B \in \mathbb{R}^{N_{l-1}^B \times N_l^B}$, where N_l^A and N_l^B are the numbers of neurons in the l-th layers in M^A and M^B. Assume $\tilde{N}_{l-1} \in [0, \min\{N_{l-1}^A, N_{l-1}^B\}]$ neurons are shared between the $(l-1)$-th layers in M^A and M^B. Hence there are $\hat{N}_{l-1}^A = N_{l-1}^A - \tilde{N}_{l-1}$ and $\hat{N}_{l-1}^B = N_{l-1}^B - \tilde{N}_{l-1}$ task-specific neurons left in the $(l-1)$-th layers in M^A and M^B. Zipping the l-th layers in in M^A and M^B consists of two steps: *neuron sharing* and *weight matrices updating*.

2.3.2.1 Neuron Sharing

To enforce neuron sharing between the l-th layers in M^A and M^B, we calculate the *functional difference* (details in Sec. 2.3.3) between the i-th neuron in layer l in M^A, and the j-th neuron in the same layer in M^B. The functional difference is measured by a metric $d[\tilde{\mathbf{w}}_{l,i}^A, \tilde{\mathbf{w}}_{l,j}^B]$, where $\tilde{\mathbf{w}}_{l,i}^A, \tilde{\mathbf{w}}_{l,j}^B \in \mathbb{R}^{\tilde{N}_{l-1}}$ are the incoming weights of the two neurons from the *shared* neurons in the $(l-1)$-th layer. We do not alter incoming weights from the non-shared neurons in the $(l-1)$-th layer because they are likely to contain task-specific information only.

To zip the l-th layers in M^A and M^B, we first calculate the functional difference for each pair of neurons (i, j) in layer l and select $\tilde{N}_l \in [0, \min\{N_l^A, N_l^B\}]$ pairs with the smallest functional difference. These pairs of neurons form a set $\{(i_k, j_k)\}$, where $k = 0, \cdots, \tilde{N}_l$ and each pair is merged into one neuron. Thus the neurons in the l-th layers in M^A and M^B fall into three groups: $\tilde{N}_l$ shared, $\hat{N}_l^A = N_l^A - \tilde{N}_l$ specific for A and $\hat{N}_l^B = N_l^B - \tilde{N}_l$ specific for B.

2.3.2.2 Weight Matrices Updating

After neuron sharing, the weight matrices $\mathbf{W}_l^A$ and $\mathbf{W}_l^B$ are re-organised as follows. The weights vectors $\tilde{\mathbf{w}}_{l,i_k}^A$ and $\tilde{\mathbf{w}}_{l,j_k}^B$, where $k = 0, \cdots, \tilde{N}_l$, are merged and replaced by a matrix $\tilde{\mathbf{W}}_l \in \mathbb{R}^{\tilde{N}_{l-1} \times \tilde{N}_l}$, whose columns are $\tilde{\mathbf{w}}_{l,k} = f(\tilde{\mathbf{w}}_{l,i_k}^A, \tilde{\mathbf{w}}_{l,j_k}^B)$, where $f(\cdot)$ is an *incoming weight update function*. $\tilde{\mathbf{W}}_l$ represents the task-relatedness between A and B from layer $(l-1)$ to layer l. The incoming weights from the $\hat{N}_{l-1}^A$ neurons in layer $(l-1)$ to the $\hat{N}_l^A$ neurons in layer l in M^A form a matrix $\hat{\mathbf{W}}_l^A \in \mathbb{R}^{N_{l-1}^A \times \hat{N}_l^A}$. The remaining columns in $\mathbf{W}_l^A$ are packed as $\tilde{\mathbf{W}}_l^A \in \mathbb{R}^{\hat{N}_{l-1}^A \times \tilde{N}_l}$. Matrices $\hat{\mathbf{W}}_l^A$ and $\tilde{\mathbf{W}}_l^A$ contain the task-specific information for A between layer $(l-1)$ and layer l. For task B, we organise matrices $\hat{\mathbf{W}}_l^B \in \mathbb{R}^{N_{l-1}^B \times \hat{N}_l^B}$ and $\tilde{\mathbf{W}}_l^B \in \mathbb{R}^{\hat{N}_{l-1}^B \times \tilde{N}_l}$ in a similar manner. We also adjust the order of rows in the weight matrices in the $(l+1)$-th layers, $\mathbf{W}_{l+1}^A$ and $\mathbf{W}_{l+1}^B$, to maintain the correct connections among neurons.

The above layer zipping process can reduce $\tilde{N}_{l-1} \times \tilde{N}_l$ weights from $\mathbf{W}_l^A$ and $\mathbf{W}_l^B$. Essential in MTZ are the neuron functional difference metric $d[\cdot]$ and the incoming weight update function $f(\cdot)$. They are designed to demand only light retraining to recover the original accuracy.

2.3.3 Deriving Neuron Functional Difference Metric $d[\cdot]$ and Incoming Weight Update Function $f(\cdot)$

This subsection introduces our neuron functional difference metric $d[\cdot]$ and weight update function $f(\cdot)$ leveraging previous research on parameter sensitivity analysis [11], [12].

2.3.3.1 Preliminaries

A naive approach to accessing the impact of a change in some parameter vector $\boldsymbol{\theta}$ on the objective function (training error) E is to apply the parameter change and re-evaluate the error on the entire training data. An alternative is to exploit second order derivatives [11], [12]. Specifically, the Taylor series of the change δE in training error due to certain parameter vector change $\delta\boldsymbol{\theta}$ is [12]:

$$\delta E = \left(\frac{\partial E}{\partial \boldsymbol{\theta}}\right)^\top \cdot \delta\boldsymbol{\theta} + \frac{1}{2}\delta\boldsymbol{\theta}^\top \cdot \mathbf{H} \cdot \delta\boldsymbol{\theta} + O(\|\delta\boldsymbol{\theta}\|^3) \tag{2.1}$$

where $\mathbf{H} = \partial^2 E / \partial \boldsymbol{\theta}^2$ is the Hessian matrix containing all the second order derivatives. For a network trained to a local minimum in E, the first term vanishes. The third and higher order terms can also be ignored [12]. Hence:

$$\delta E = \frac{1}{2}\delta\boldsymbol{\theta}^\top \cdot \mathbf{H} \cdot \delta\boldsymbol{\theta} \tag{2.2}$$

Eq.(2.2) approximates the deviation in error due to parameter changes. However, it is still a bottleneck to compute and store the Hessian matrix $\mathbf{H}$ of a

modern deep neural network. For instance, applying the weight pruning scheme proposed in [12] on a VGG-16 model [2] trained on the ImageNet ILSVRC-2012 dataset [40] requires the calculation of a Hessian matrix with approximately $(138 \times 10^6)^2 = 1.9044 \times 10^{16}$ elements.

As next, we harness the trick in [11] to break the calculations of Hessian matrices into layer-wise, and propose a Hessian-based neuron difference metric as well as the corresponding weight update function for neuron sharing.

2.3.3.2 Our Method

Inspired by [11] we define the error functions of M^A and M^B in layer l as

$$E_l^A = \frac{1}{n_A} \sum \|\tilde{\mathbf{y}}_l^A - \mathbf{y}_l^A\|^2 \tag{2.3}$$

$$E_l^B = \frac{1}{n_B} \sum \|\tilde{\mathbf{y}}_l^B - \mathbf{y}_l^B\|^2 \tag{2.4}$$

where $\mathbf{y}_l^A$ and $\tilde{\mathbf{y}}_l^A$ are the *pre-activation* outputs of the l-th layers in M^A before and after layer zipping, evaluated on one instance from the training set of A; $\mathbf{y}_l^B$ and $\tilde{\mathbf{y}}_l^B$ are defined in a similar way; $\|\cdot\|$ is l^2-norm; n_A and n_B are the number of training samples for M^A and M^B, respectively; Σ is the summation over all training instances. Since M^A and M^B are trained to a local minimum in training error, E_l^A and E_l^B will have the same minimum points as the corresponding training errors.

We further define an error function of the combined network in layer l as

$$E_l = \alpha E_l^A + (1-\alpha) E_l^B \tag{2.5}$$

where $\alpha \in (0, 1)$ is used to balance the errors of M^A and M^B. The change in E_l with respect to neuron sharing in the l-th layer can be expressed in a similar form as Eq.(2.2):

$$\delta E_l = \frac{1}{2}(\delta \tilde{\mathbf{w}}_{l,i}^A)^\top \cdot \tilde{\mathbf{H}}_{l,i}^A \cdot \delta \tilde{\mathbf{w}}_{l,i}^A + \frac{1}{2}(\delta \tilde{\mathbf{w}}_{l,j}^B)^\top \cdot \tilde{\mathbf{H}}_{l,j}^B \cdot \delta \tilde{\mathbf{w}}_{l,j}^B \tag{2.6}$$

where $\delta \tilde{\mathbf{w}}_{l,i}^A$ and $\delta \tilde{\mathbf{w}}_{l,j}^B$ are the adjustments in the weights of i and j to merge the two neurons; $\tilde{\mathbf{H}}_{l,i}^A = \partial^2 E_l / (\partial \tilde{\mathbf{w}}_{l,i}^A)^2$ and $\tilde{\mathbf{H}}_{l,j}^B = \partial^2 E_l / (\partial \tilde{\mathbf{w}}_{l,j}^B)^2$ denote the *layer-wise* Hessian matrices. Similarly to [11], the layer-wise Hessian matrices can be calculated as

$$\tilde{\mathbf{H}}_{l,i}^A = \frac{\alpha}{n_A} \sum \mathbf{x}_{i-1}^A \cdot (\mathbf{x}_{i-1}^A)^\top \tag{2.7}$$

$$\tilde{\mathbf{H}}_{l,j}^B = \frac{1-\alpha}{n_B} \sum \mathbf{x}_{j-1}^B \cdot (\mathbf{x}_{j-1}^B)^\top \tag{2.8}$$

where $\mathbf{x}_{i-1}^A$ and $\mathbf{x}_{j-1}^B$ are the outputs of layer $(l-1)$ in M^A and M^B, respectively.

When sharing the i-th and j-th neurons in the l-th layers of M^A and M^B, our aim is to minimize δE_l, which can be formulated as the optimization problem below:

$$\min_{(i,j)}\{\min_{(\delta\tilde{\mathbf{w}}^A_{l,i},\delta\tilde{\mathbf{w}}^B_{l,j})}\delta E_l\}\,\text{s.t.}\,\tilde{\mathbf{w}}^A_{l,i}+\delta\tilde{\mathbf{w}}^A_{l,i}=\tilde{\mathbf{w}}^B_{l,j}+\delta\tilde{\mathbf{w}}^B_{l,j} \tag{2.9}$$

For the inner minimization problem:

$$\min_{(\delta\tilde{\mathbf{w}}^A_{l,i},\delta\tilde{\mathbf{w}}^B_{l,j})}\delta E_l \quad \text{s.t.}\,\tilde{\mathbf{w}}^A_{l,i}+\delta\tilde{\mathbf{w}}^A_{l,i}=\tilde{\mathbf{w}}^B_{l,j}+\delta\tilde{\mathbf{w}}^B_{l,j} \tag{2.10}$$

we form Lagrange multipliers with the second order approximation in (2.2):

$$\begin{aligned}L =&\frac{1}{2}(\delta\tilde{\mathbf{w}}^A_{l,i})^\top\cdot\tilde{\mathbf{H}}^A_{l,i}\cdot\delta\tilde{\mathbf{w}}^A_{l,i}+\frac{1}{2}(\delta\tilde{\mathbf{w}}^B_{l,j})^\top\cdot\tilde{\mathbf{H}}^B_{l,j}\cdot\delta\tilde{\mathbf{w}}^B_{l,j}\\&+\boldsymbol{\lambda}^\top\cdot(\tilde{\mathbf{w}}^A_{l,i}+\delta\tilde{\mathbf{w}}^A_{l,i}-\tilde{\mathbf{w}}^B_{l,j}-\delta\tilde{\mathbf{w}}^B_{l,j})\end{aligned} \tag{2.11}$$

where $\boldsymbol{\lambda}$ is the vector of Lagrange undetermined multipliers. By taking functional derivatives and employing the constraints of Eq.(2.9), we have closed-form solutions:

$$\begin{aligned}\delta\tilde{\mathbf{w}}^{A,opt}_{l,i} =&(\tilde{\mathbf{H}}^A_{l,i})^{-1}\cdot\left((\tilde{\mathbf{H}}^A_{l,i})^{-1}+(\tilde{\mathbf{H}}^B_{l,j})^{-1}\right)^{-1}\\&\cdot(\tilde{\mathbf{w}}^B_{l,j}-\tilde{\mathbf{w}}^A_{l,i})\end{aligned} \tag{2.12}$$

$$\begin{aligned}\delta\tilde{\mathbf{w}}^{B,opt}_{l,j} =&(\tilde{\mathbf{H}}^B_{l,j})^{-1}\cdot\left((\tilde{\mathbf{H}}^A_{l,i})^{-1}+(\tilde{\mathbf{H}}^B_{l,j})^{-1}\right)^{-1}\\&\cdot(\tilde{\mathbf{w}}^A_{l,i}-\tilde{\mathbf{w}}^B_{l,j})\end{aligned} \tag{2.13}$$

$$\begin{aligned}\delta E^{opt}_l =&\frac{1}{2}(\tilde{\mathbf{w}}^A_{l,i}-\tilde{\mathbf{w}}^B_{l,j})^\top\cdot\left((\tilde{\mathbf{H}}^A_{l,i})^{-1}+(\tilde{\mathbf{H}}^B_{l,j})^{-1}\right)^{-1}\\&\cdot(\tilde{\mathbf{w}}^A_{l,i}-\tilde{\mathbf{w}}^B_{l,j})\end{aligned} \tag{2.14}$$

We define the neuron functional difference metric as:

$$d[\tilde{\mathbf{w}}^A_{l,i},\tilde{\mathbf{w}}^B_{l,j}]=\delta E^{opt}_l \tag{2.15}$$

and the weight update function as:

$$f(\tilde{\mathbf{w}}^A_{l,i},\tilde{\mathbf{w}}^B_{l,j})=\tilde{\mathbf{w}}^A_{l,i}+\delta\tilde{\mathbf{w}}^{A,opt}_{l,i}=\tilde{\mathbf{w}}^B_{l,j}+\delta\tilde{\mathbf{w}}^{B,opt}_{l,j}. \tag{2.16}$$

2.3.4 MTZ Framework

Algorithm 1 outlines the process of MTZ on two tasks of the same input domain, *e.g.*, images. We first construct a joint input layer. In case the input layer dimensions are not equal in both tasks, the dimension of the joint input layer equals the larger dimension of the two original input layers, and fictive connections (*i.e.*, weight 0) are added to the model whose original input layers are smaller. Afterwards we begin layer-wise neuron sharing and weight matrix updating from the first hidden layer. The two networks are "zipped" layer by layer till the last hidden layer and we obtain a combined network. After merging each layer, the networks are retrained to re-boost the accuracy.

Practical Issues. We make the following notes on the practicability of MTZ.

Algorithm 1: Multi-task Zipping via Layer-wise Neuron Sharing

input : $\{\mathbf{W}_l^A\}$, $\{\mathbf{W}_l^B\}$: weight matrices of M^A, M^B; $\mathbf{X}^A, \mathbf{X}^B$: training datum of task A and B (including labels); α: coefficient to balance M^A and M^B; $\{\tilde{N}_l\}$: number of neurons to be shared in layer l

1 **for** $l = 1, \ldots, L-1$ **do**
2 Calculate inputs for the current layer $\mathbf{x}_{l-1}^A$ and $\mathbf{x}_{l-1}^B$ using training data from $\mathbf{X}^A$ and $\mathbf{X}^B$ and forward propagation
3 $\tilde{\mathbf{H}}_{l,i}^A \leftarrow \frac{\alpha}{n_A} \sum \mathbf{x}_{i-1}^A \cdot (\mathbf{x}_{i-1}^A)^\top$
4 $\tilde{\mathbf{H}}_{l,j}^B \leftarrow \frac{1-\alpha}{n_B} \sum \mathbf{x}_{j-1}^B \cdot (\mathbf{x}_{j-1}^B)^\top$
5 Select $\tilde{N}_l$ pairs of neurons $\{(i_k, j_k)\}$ with the smallest $d[\tilde{\mathbf{w}}_{l,i}^A, \tilde{\mathbf{w}}_{l,j}^B]$
6 **for** $k \leftarrow 1, \ldots, \tilde{N}_l$ **do**
7 $\tilde{\mathbf{w}}_{l,k} \leftarrow f(\tilde{\mathbf{w}}_{l,i_k}^A, \tilde{\mathbf{w}}_{l,j_k}^B)$
8 Re-organize $\mathbf{W}_l^A$ and $\mathbf{W}_l^B$ into $\tilde{\mathbf{W}}_l$, $\hat{\mathbf{W}}_l^A$, $\tilde{\mathbf{W}}_l^A$, $\hat{\mathbf{W}}_l^B$ and $\tilde{\mathbf{W}}_l^B$
9 Permute the order of rows in $\mathbf{W}_{l+1}^A$ and $\mathbf{W}_{l+1}^B$ to maintain correct connections
10 Conduct a light retraining on task A and B to re-boost accuracy of the joint model

output: $\{\hat{\mathbf{W}}_l^A\}, \{\tilde{\mathbf{W}}_l^A\}, \{\tilde{\mathbf{W}}_l\}, \{\tilde{\mathbf{W}}_l^B\}, \{\hat{\mathbf{W}}_l^B\}$: weights of the zipped multi-task model M^C

■ *How to set the number of neurons to be shared?* One can directly set $\tilde{N}_l$ neurons to be shared for the l-th layers, or set a layer-wise threshold ε_l instead. Given a threshold ε_l, MTZ shares pairs of neurons where $\{(i_k, j_k) | d[\tilde{\mathbf{w}}_{l,i_k}^A, \tilde{\mathbf{w}}_{l,j_k}^B] < \varepsilon_l\}$. In this case $\tilde{N}_l = |\{(i_k, j_k)\}|$. One can set $\{\tilde{N}_l\}$ if there is a hard constraint on storage or memory. Otherwise $\{\varepsilon_l\}$ can be set if accuracy is of higher priority. Note that $\{\varepsilon_l\}$ controls the layer-wise error δE_l, which correlates to the accumulated errors of the outputs in layer L $\tilde{\varepsilon}^A = \frac{1}{\sqrt{n_A}} \sum \|\tilde{\mathbf{x}}_L^A - \mathbf{x}_L^A\|$ and $\tilde{\varepsilon}^B = \frac{1}{\sqrt{n_B}} \sum \|\tilde{\mathbf{x}}_L^B - \mathbf{x}_L^B\|$ [11].

■ *How to execute the combined model for each task?* During inference, only task-related connections in the combined model are enabled. For instance, when performing inference on task A, we only activate $\{\hat{\mathbf{W}}_l^A\}$, $\{\tilde{\mathbf{W}}_l^A\}$ and $\{\tilde{\mathbf{W}}_l\}$, while $\{\tilde{\mathbf{W}}_l^B\}$ and $\{\hat{\mathbf{W}}_l^B\}$ are disabled (*e.g.*, by setting them to zero).

■ *How to zip more than two neural networks?* MTZ is able to zip more than two models by sequentially adding each network into the joint network, and the calculated Hessian matrices of the already zipped joint network can be reused. Therefore, MTZ is scalable in regards to both the depth of each network and the number of tasks to be zipped. Also note that since calculating the Hessian matrix of one layer requires only its layer input, only one forward pass in total from each model is needed for the merging process (excluding retraining).

Complexity Analysis. We only analyse the complexity of the main merging process (Line 3 to 9) and ignore the complexity of the forward and backward propagation of neural networks (Line 2 and 10). For simplicity, we consider fully merging two layers from two networks, which both have n neurons in every hidden layer. Line 3 and 4 take $O(n^2)$ time and $O(n^2)$ memory. Line 5 takes $O(n^5)$ time and $O(n^2)$ memory (using in-place matrix inversion) to calculate all the paring distances, and then $O(n^4)$ to sort them. This is the most time consuming step, as the calculation of the merging criterion (2.15) involves inversion of the Hessian matrices. There are $O(n)$ iterations in Line 6 and 7, and line 7 takes $O(n^3)$ time and $O(n^2)$ memory. Line 8 and 9 take

$O(n^2)$ time and $O(n^2)$ memory. Therefore, the total time cost is $O(n^5)$ and memory cost is $O(n^2)$.

2.3.5 Propagation of Layer-wise Error

Note that we define layer-wise error function Eq.(2.5) to avoid calculating the entire Hessian matrix. In this subsection, we demonstrate the effectiveness of such a layer-wise formulation by proving that the accumulated error at the output layer is bounded.

To analyse the accumulated error at the output layer, we investigate how the error Eq.(2.5) propagates from layer l to the final output layer. Note that the error function in layer l consists of two parts, E_l^A and E_l^B, as defined in Eq.(2.3) and Eq.(2.4), which are defined with pre-activation outputs $\mathbf{y}_l^A$ and $\tilde{\mathbf{y}}_l^A$. In order to understand the propagation of errors, however, we need to take activation function into consideration. We define:

$$\mathcal{E}_l^A = \frac{1}{n_A} \sum \|\tilde{\mathbf{z}}_l^A - \mathbf{z}_l^A\|^2 \tag{2.17}$$

$$\mathcal{E}_l^B = \frac{1}{n_B} \sum \|\tilde{\mathbf{z}}_l^B - \mathbf{z}_l^B\|^2 \tag{2.18}$$

$$\mathcal{E}_l = \alpha \cdot \mathcal{E}_l^A + (1-\alpha) \cdot \mathcal{E}_l^B \tag{2.19}$$

where $\mathbf{z}_l^A = \sigma(\mathbf{y}_l^A)$, $\tilde{\mathbf{z}}_l^A = \sigma(\tilde{\mathbf{y}}_l^A)$, $\mathbf{z}_l^B = \sigma(\mathbf{y}_l^B)$ and $\tilde{\mathbf{z}}_l^B = \sigma(\tilde{\mathbf{y}}_l^B)$ are *post-activation* layer outputs with activation function $\sigma(\cdot)$. In this chapter, we consider the widely adopted activation function: rectified linear unit (ReLU).

After merging the l-th layer, there are three groups of neurons: $\hat{N}_l^A$ task-A-specific neurons, $\hat{N}_l^B$ task-B-specific neurons, and $\tilde{N}_l$ shared neurons. When task A is performed, only task-A-specific and shared neurons are activated. The connections between the task-A-specific and shared neurons in the $l-1$-th and l-th layer have weights $\mathbf{W}_l'^A$:

$$\mathbf{W}_l'^A = \left[\begin{array}{c|c} \hat{\mathbf{W}}_l^A & \begin{matrix}\tilde{\mathbf{W}}_l^A \\ \tilde{\mathbf{W}}_l\end{matrix} \end{array}\right] \tag{2.20}$$

Similarly, we can define $\mathbf{W}_l'^B$. Furthermore, we denote the vectorisation of the weight matrices $\mathbf{W}_l'^A$ and $\mathbf{W}_l'^B$ as $\mathcal{V}_l^A$ and $\mathcal{V}_l^B$, respectively.

Adapting the conclusions in [11] to multiple neural networks, the propagation of the layer-wise error in MTZ can be described by the following theorem:

Theorem 2.1. *For a multi-task network merged via Algorithm 1 with L layers, the accumulated error of the last layer output is upper-bounded by:*

$$\begin{aligned} \mathcal{E}_L \leq \sum_{i=1}^{L-1} \Bigg(& \alpha \cdot \prod_{j=i+1}^{L-1} \|\mathcal{V}_j^A\| \sqrt{\delta E_i^A} \\ & + (1-\alpha) \cdot \prod_{j=i+1}^{L-1} \|\mathcal{V}_j^B\| \sqrt{\delta E_i^B} \Bigg) \end{aligned} \tag{2.21}$$

Proof. From similar derivation as in [11], we have:

$$\mathcal{E}_L^A \leq \sum_{i=1}^{L-1} \left(\prod_{j=i+1}^{L} \|\mathcal{V}_j^A\| \sqrt{\delta E_i^A} \right) + \sqrt{\delta E_L^A} \tag{2.22}$$

and the same holds if we switch A with B. However, as the last layer, *i.e.*, the output layer is untouched, we have $\sqrt{\delta E_L^A} = \sqrt{\delta E_L^B} = 0$. Therefore:

$$\mathcal{E}_L^A \leq \sum_{i=1}^{L-1} \left(\prod_{j=i+1}^{L-1} \|\mathcal{V}_j^A\| \sqrt{\delta E_i^A} \right) \tag{2.23}$$

which also holds if we swap A and B. Finally, since $\mathcal{E}_l = \alpha \cdot \mathcal{E}_l^A + (1-\alpha) \cdot \mathcal{E}_l^B$, Eq.(2.21) holds. ■

2.4 MTZ Extensions

In this section, we explain how to extend MTZ to support sparse models (Sec. 2.4.1), other commonly used layers in computer vision *e.g.*, convolutional (CONV) layers, batch normalisation (BN) layers and residual blocks (Sec. 2.4.2).

2.4.1 Support for Sparse Models

Since the pre-trained neural networks may have already been sparsified via weight pruning, we also extend MTZ to support sparse models. Specifically, we use sparse matrices, where zeros indicate no connections, to represent such sparse models. Then the incoming weights from the previous shared neurons $\tilde{\mathbf{w}}_{l,i}^A, \tilde{\mathbf{w}}_{l,j}^B$ still have the same dimension. Therefore $d[\tilde{\mathbf{w}}_{l,i}^A, \tilde{\mathbf{w}}_{l,j}^B]$, $f(\tilde{\mathbf{w}}_{l,i}^A, \tilde{\mathbf{w}}_{l,j}^B)$ can be calculated as before. However, we also calculate two mask vectors $\tilde{\mathbf{m}}_{l,i}^A$ and $\tilde{\mathbf{m}}_{l,j}^B$, whose elements are 0 when the corresponding elements in $\tilde{\mathbf{w}}_{l,i}^A$ and $\tilde{\mathbf{w}}_{l,j}^B$ are 0, and 1 otherwise. We pick the mask vector with more $1's$ and apply it to $\tilde{\mathbf{w}}_l$. This way the combined model always have a smaller number of weights than the sum of the original two models.

2.4.2 Extension to Other Layers

This subsection introduces how to extend MTZ from FC layers to CONV layers, BN layers and residual blocks.

2.4.2.1 Extensions to Convolutional Layers

The layer zipping procedure of two convolutional layers are similar to that of two fully connected layers. The only difference is that sharing is performed on *kernels* rather than *neurons*. Take the i-th kernel of size $k_l \times k_l$ in layer l

of M^A as an example. Its incoming weights from the previous shared kernels are $\tilde{\mathbf{W}}_{l,i}^{A,in} \in \mathbb{R}^{k_l \times k_l \times \tilde{N}_{l-1}}$. The weights are then flatten into a vector $\tilde{\mathbf{w}}_{l,i}^{A}$ to calculate functional differences. As with in Sec. 2.3.2, after layer zipping in the l-th layers, the weight matrices in the $(l+1)$-th layers need careful permutations regarding the flattening ordering to maintain correct connections among neurons, especially when the next layers are fully connected layers.

2.4.2.2 Extensions to Batch Normalisation Layers

BN layers are typically applied on the pre-activation outputs of CONV layers. After training, the output of the BN layer applied on the i-th channel of layer l is:

$$BN(y_{l,i}) = \gamma_{l,i} \cdot \frac{y_{l,i} - \mu_{l,i}}{\sqrt{\sigma_{l,i}^2 + \epsilon}} + \beta_{l,i} \tag{2.24}$$

where $y_{l,i}$ is the pre-activation output of the CONV layer, $\gamma_{l,i}$ and $\beta_{l,i}$ are the two learnable parameters (scaling and shifting) for the BN layer, $\mu_{l,i}$ and $\sigma_{l,i}$ are the pre-calculated mean and standard deviation.

Since all the parameters are fixed after training, the effect of the BN layer can be replaced by multiplying the incoming weight $\mathbf{w}_{l,i}$ by a scalar $\frac{\gamma_{l,i}}{\sigma_{l,i}}$ and adding $\beta_{l,i} - \frac{\gamma_{l,i} \cdot \mu_{l,i}}{\sigma_{l,i}}$ to the bias $b_{l,i}$. The calculation of the Hessian matrices (2.7) and (2.8) remains the same, and in the closed-form solutions Eq.(2.12), Eq.(2.13) and Eq.(2.14) the new weights and bias should be used. Later in the retraining phase, newly initialised BN layers need to be applied.

2.4.2.3 Extensions to Residual Blocks

At the end of each residual block, the output vector of the last CONV layer is added with the identity shortcut vector. This addition can be considered as a layer of neurons (channels) with binary weights (1 or 0) fully connected to the last convolutional layer and the last shortcut addition layer (or in the case of the first residual block, it connect to the pre-convolutional layer). However, in order to continue the chain of MTZ, the neurons at this addition layer should be marked as shared/unshared. Since the neuron sharing situation of the last CONV layer in the current residual block can be different from of the addition layer of the last residual block, there might be conflicts. We propose an exact and an approximate method to combine residual blocks.

Exact Method. We illustrate the exact method via an example residual block with output dimension of three. In the last CONV layer of the current block, the first neuron in model A is shared with the second neuron in model B, and the third neuron in model A is shared with the third neuron in model B. In the addition layer of the last block, the second neuron in model A is shared with the second neuron in model B. Fig. 2.3 illustrates the weight matrices of the current addition layers of model A and B, where the red column indicates

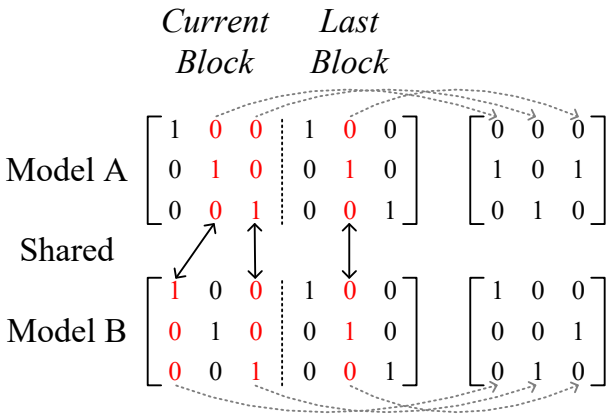


Figure 2.3 An example of the weight matrices when merging residual blocks with output dimension of three.

the weights from merged neurons, and the weight matrices from the merged neurons.

As shown in this example, we mark the neurons in the addition layer as shared/unshared as follows:

- If a neuron in the addition layer is not connecting (*i.e.*, having non-zero weights) to any shared convolutional neuron or shared shortcut neuron, *e.g.*, the first row in the matrices on the right of Fig. 2.3, this neuron is marked as unshared.
- In other cases, these addition neurons are merged by analysing their incoming weights from merged neurons, *e.g.*, the matrices on the right of Fig. 2.3.

Approximate Method. The exact method above requires an additional layer to be processed by MTZ, and actually adds more weights to the model. To avoid this problem, we propose an approximate method to merge residual blocks. The idea is to use the neuron sharing scheme of the last CONV layer in the current residual block as the reference. In other words, in the last CONV layer in current residual block, if the i-th neuron in model A is merged with j-th neuron in model B, then their corresponding neurons in the addition layer are also marked as merged.

2.5 Evaluations

We first evaluate the performance of MTZ on zipping two networks pre-trained for the same task (Sec. 2.5.1) and different tasks (Sec. 2.5.2). We mainly assess the test errors of each task after network zipping and the retraining overhead involved. We then show that MTZ can merge models for different input domains (Sec. 2.5.3). Finally we show that MTZ is scalable and reduces the execution time of neural networks on resource-constrained mobile devices (Sec. 2.5.4).

MTZ is implemented with TensorFlow. All experiments are conducted on a workstation equipped with Nvidia Titan X Maxwell GPU. The execution time of neural networks is measured on the Jetson Nano embedded platform [57] equipped with a 128-core Maxwell GPU, a Quad-core ARM A57 (1.43GHz) CPU, and 4GB 64-bit LPDDR4 (25.6GB/s).

2.5.1 Zipping Two Networks for the Same Task

This experiment validates the effectiveness of MTZ by merging two differently trained models for the same task. Ideally, two models trained to different local optimums should function the same on the test data. Therefore their hidden layers can be fully merged without incurring any accuracy loss. This experiment aims to show that, by finding the correct pairs of neurons which shares the same functionality, MTZ can achieve the theoretical limit of compression ratio *i.e.*, 100%, even without any retraining involved.

Dataset and Settings. We test on MNIST dataset with the LeNet-300-100 and LeNet-5 networks [24] to recognise handwritten digits from 0 to 9. LeNet-300-100 is a fully connected network with two hidden layers (300 and 100 neurons each), reporting an error from 1.6% to 1.76% on MNIST [11], [24]. LeNet-5 is a convolutional network with two convolutional layers and two fully connected layers, which achieves an error from 0.8% to 1.27% on MNIST [11], [24].

We train two LeNet-300-100 networks of our own with errors of 1.57% and 1.60%; and two LeNet-5 networks with errors of 0.89% and 0.95%. All the networks are initialised randomly with different seeds, and the training data are also shuffled before every training epoch. After training, the ordering of neurons/kernels in all hidden layers is once more randomly permuted. Therefore the models have completely different parameters (weights). The training of LeNet-300-100 and LeNet-5 networks requires 1.05×10^4 and 1.1×10^4 iterations in average, respectively.

For sparse networks, we apply one iteration of L-OBS [11] to prune the weights of the four LeNet networks. We then enforce all neurons to be shared in each hidden layer of the two dense LeNet-300-100 networks, sparse LeNet-300-100 networks, dense LeNet-5 networks, and sparse LeNet-5 networks, using MTZ.

Results. Fig. 2.4a plots the average error after sharing different amounts of neurons in the first layers of two dense LeNet-300-100 networks. Fig. 2.4b shows the error by further merging the second layers. We compare MTZ with a random sharing scheme, which shares neurons by first picking (i_k, j_k) at random, and then choosing randomly between $\tilde{\mathbf{w}}^A_{l,i_k}$ and $\tilde{\mathbf{w}}^B_{l,j_k}$ as the shared weights $\tilde{\mathbf{w}}_{l_k}$. When all the 300 neurons in the first hidden layers are shared, there is an increase of 0.95% in test error (averaged over the two models) even without retraining, while random sharing induces an error of 33.47%. We also use MTZ to fully merge the hidden layers in the two LeNet-300-100 networks without any

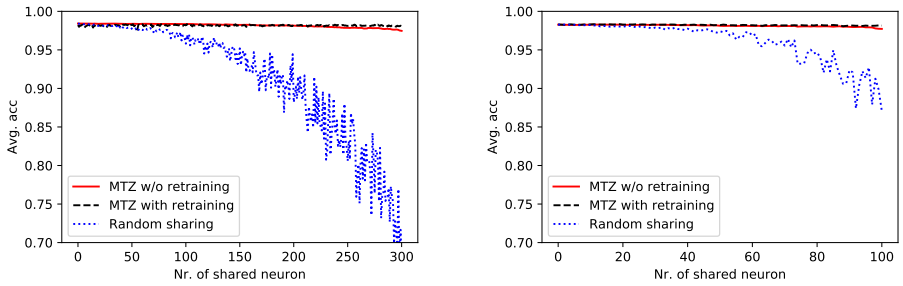


Figure 2.4 Test error on MNIST by continually sharing neurons in (a) the first and (b) the second fully connected layers of two dense LeNet-300-100 networks till the merged layers are fully shared.

Table 2.1 Test errors on MNIST by sharing all neurons in two LeNet networks.

Model	err_A	err_B	re-err_C	# re-iter
LeNet-300-100-Dense	1.57%	1.60%	1.64%	550
LeNet-300-100-Sparse	1.80%	1.81%	1.83%	800
LeNet-5-Dense	0.89%	0.95%	0.93%	600
LeNet-5-Sparse	1.27%	1.28%	1.29%	1200

retraining *i.e.*, without line 10 in Algorithm 1. The averaged test error increases by only 1.50%.

Table 2.1 summarises the errors of each LeNet pair before zipping (err_A and err_B), after fully merged with retraining (re-err_C) and the number of retraining iterations involved (# re-iter). MTZ consistently achieves lossless network zipping on FC and CONV networks, either they are dense or sparse, with 100% parameters of hidden layers shared. Meanwhile, the number of retraining iterations is approximately $19.0\times$ and $18.7\times$ fewer than that of training a dense LeNet-300-100 network and a dense LeNet-5 network, respectively.

2.5.2 Zipping Two Networks for Different Tasks

This experiment evaluates the performance of MTZ to automatically share information among two neural networks for different tasks. We investigate: *(i)* what the accuracy loss is when all hidden layers of two models for different tasks are fully shared (in purpose of maximal size reduction); *(ii)* how much neurons and parameters can be shared between the two models by MTZ with at most 0.5% increase in test errors allowed (in purpose of minimal accuracy loss); *(iii)* how MTZ performs compared with the state-of-the-art cross-model compression schemes [6], [22].

Dataset and Settings. We first test the performance of MTZ for either *maximal size reduction* or *minimal accuracy loss*. We merge two VGG-16 networks trained on the ImageNet ILSVRC-2012 dataset [40] for object

Table 2.2 Test errors and retraining iterations of sharing all neurons (output layer fc8 excluded) in two VGG-16 networks for ImageNet and CelebA.

Layer	N_l^A	ImageNet (Top-5 Error)		CelebA (Error)		# re-iter
		w/o-re-err$_C$	re-err$_C$	w/o-re-err$_C$	re-err$_C$	
conv1_1	64	10.59%	10.61%	8.45%	8.43%	50
conv1_2	64	11.19%	10.78%	8.82%	8.77%	100
conv2_1	128	10.99%	10.68%	8.91%	8.82%	100
conv2_2	128	11.31%	11.03%	9.23%	9.07%	100
conv3_1	256	11.65%	11.46%	9.16%	9.04%	100
conv3_2	256	11.92%	11.83%	9.17%	9.05%	100
conv3_3	256	12.54%	12.41%	9.46%	9.34%	100
conv4_1	512	13.40%	12.28%	10.18%	9.69%	400
conv4_2	512	13.02%	12.62%	10.65%	10.25%	400
conv4_3	512	13.11%	12.97%	12.03%	10.92%	400
conv5_1	512	13.46%	13.09%	12.62%	11.68%	400
conv5_2	512	13.77%	13.20%	12.61%	11.64%	400
conv5_3	512	36.07%	13.35%	13.10%	12.01%	1×10^3
fc6	4096	15.08%	15.17%	12.31%	11.71%	2×10^3
fc7	4096	15.73%	14.07%	11.98%	11.09%	1×10^4

classification and the CelabA dataset [41] for facial attribute classification. The ImageNet dataset contains images of $1,000$ object categories. The CelebA dataset consists of 200 thousand celebrity face images labelled with 40 attribute classes. VGG-16 has 13 convolutional layers and 3 fully connected layers. We adopt the pre-trained weights from the original VGG-16 model [2] for the object classification task, which has a 10.31% error. For the facial attribute classification task, we train a second VGG-16 model following a similar process as in [26]. We initialise the convolutional layers of a VGG-16 model using the pre-trained parameters from imdb-wiki [25], and train the remaining 3 fully connected layers till the model yields an error of 8.50%, which matches the accuracy of the VGG-16 model in [26] on CelebA.

We conduct two experiments with the two VGG-16 models. *(i)* All hidden layers in the two models are 100% merged using MTZ. *(ii)* Each pair of layers in the two models are adaptively merged using MTZ allowing an increase ($< 0.5\%$) in test errors on the two datasets.

Results. Table 2.2 summarises the performance when each pair of hidden layers are 100% merged. The test errors of both tasks gradually increase during the zipping procedure from layer conv1_1 to conv5_2 and then the error on ImageNet surges when conv5_3 are 100% shared. After $1,000$ iterations of retraining, the accuracies of both tasks are resumed. When 100% parameters of all hidden layers are shared between the two models, the joint model yields test errors of 14.07% on ImageNet and 11.09% on CelebA, *i.e.*, increases of 3.76% and 2.59% in the original test errors.

Table 2.3 shows the performance when each pair of hidden layers are adaptively merged. MTZ achieves an increase in test errors of 0.44% on ImageNet and 0.45% on CelebA. Approximately 39.61% of the parameters in the two models are shared (56.94% in the 13 CONV layers and 38.17% in the 2 FC

Table 2.3 Test errors, number of shared neurons, and retraining iterations of adaptively zipping two VGG-16 networks for ImageNet and CelebA.

Layer	N_l^A	$\tilde{N}_l$	ImageNet (Top-5 Error)		CelebA (Error)		# re-iter
			w/o-re-err$_C$	re-err$_C$	w/o-re-err$_C$	re-err$_C$	
conv1_1	64	64	10.28%	10.37%	8.39%	8.33%	50
conv1_2	64	64	10.93%	10.50%	8.77%	8.54%	100
conv2_1	128	96	10.74%	10.57%	8.62%	8.46%	100
conv2_2	128	96	10.87%	10.79%	8.56%	8.47%	100
conv3_1	256	192	10.83%	10.76%	8.62%	8.48%	100
conv3_2	256	192	10.92%	10.71%	8.52%	8.44%	100
conv3_3	256	192	10.86%	10.71%	8.83%	8.63%	100
conv4_1	512	384	10.69%	10.51%	9.39%	8.71%	400
conv4_2	512	320	10.43%	10.46%	9.06%	8.80%	400
conv4_3	512	320	10.56%	10.36%	9.36%	8.93%	400
conv5_1	512	436	10.42%	10.51%	9.54%	9.15%	400
conv5_2	512	436	10.47%	10.49%	9.43%	9.16%	400
conv5_3	512	436	10.49%	10.24%	9.61%	9.07%	1×10^3
fc6	4096	1792	11.46%	11.33%	9.37%	9.18%	2×10^3
fc7	4096	4096	11.45%	10.75%	9.15%	8.95%	1.5×10^4

layers). The zipping procedure involves $20,650$ iterations of retraining. For comparison, at least 3.7×10^5 iterations are needed to train a VGG-16 network from scratch [2]. That is, MTZ is able to inherit information from the pre-trained models and construct a combined model with an increase in test errors of less than 0.5%. And the process requires at least $17.9\times$ fewer (re)training iterations than training a joint network from scratch.

For comparison, we also trained a fully shared multi-task VGG-16 with two split classification layers jointly on both tasks. The test errors are 14.88% on ImageNet and 13.29% on CelebA. This model has exactly the same topology and amount of parameters as our model constructed by MTZ, but performs slightly worse on both tasks.

Comparison with State-of-the-Arts. We compare our MTZ against two recent cross-model compression schemes, Neural Weight Virtualisation [6] (NWV for short) and ZipperNet [22] in the *fully shared* setting, *i.e.*, in purpose of *maximal size reduction*. We choose the fully shared setting because NWV adopts hard parameter sharing, *i.e.*, parameters are shared across all tasks. ZipperNet allows partial parameter sharing but all the weights *within* a layer are fully shared.

To compare with ZipperNet [22], we apply it to merge two VGG-16 networks pre-trained on ImageNet and CelabA as above. Specifically, we perform filter alignment by Hungarian algorithm for each CONV layer and then retrain the merged network as [22]. The merging process starts with the first CONV layer and continues until all CONV layers are merged. Since ZipperNet does not apply to FC layers, we leave the two FC layers in the VGG-16 separate. For fair comparison, the number of retraining iterations for each CONV layer is set to the same as our MTZ (detailed numbers see Table 2.2). Fig. 2.5 plots the test errors after fully merging each CONV layer in the two VGG-16 networks.

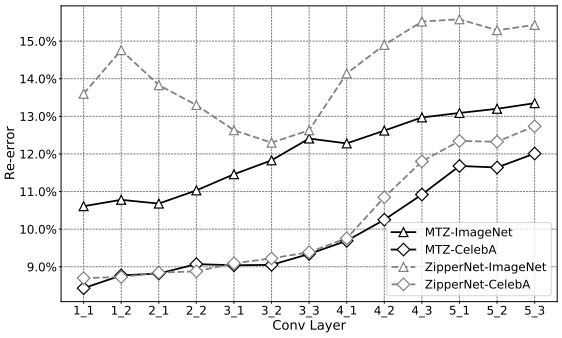


Figure 2.5 Test errors after merging each CONV layer in two VGG-16 networks for ImageNet and CelebA using ZipperNet [22].

Table 2.4 Test errors of the joint network merged by NWV [6] and MTZ. The joint model is fully shared except the last classification layer.

	GTSRB	SVHN	Average
NWV	7.04%	7.65%	7.35%
MTZ	1.05%	6.86%	3.96%

In general, ZipperNet performs worse than MTZ in terms of accuracy after merging. The difference gets increasingly evident after merging the `conv4_2` layer. In the end, ZipperNet only achieves a test errors of 15.43% on ImageNet and 12.74% on CelebA, 2.08% and 0.73% worse than MTZ.

For comparison with NWV [6], we merge two ResNet-28 networks [58] to cover diverse model architectures. The two ResNet-28 networks are pre-trained on German Traffic Sign Recognition Benchmark (GTSRB) [59] and Street View House Numbers (SVHN) [60]. Table 2.7 provides a summary of the datasets. When merging the two ResNet-28 networks with NWV, the weights are organised into $5,822$ weight-pages with a page size of $1,000$. Then the weight-pages are retrained as in NWV to recover the inference accuracy. Table 2.4 lists the test errors of the fully shared ResNet-28 model merged by NWV and MTZ. MTZ achieves better accuracy than NMV on both tasks after merging. Compared with NWV, MTZ yields 5.99% lower error on GTSRB and 0.79% on SVHN.

To demonstrate the feasibility of MTZ on merging models of similar architectures yet different depth, we further merge a ResNet-28 network and a ResNet-34 network, which are pretrained respectively on GTSRB and SVHN. The two networks are merged layer-by-layer from the first layers and the extra layers in the ResNet-34 are left independent. Table 2.5 shows the test errors of the merged model. We can see that the merged model performs well on both tasks, showing the effectiveness of MTZ merging networks with different depth.

Table 2.5 Test errors of the joint network by merging ResNet-28 (pretrained on GTSRB) and ResNet-34 (pretrained on SVHN) using MTZ. The joint model is merged layer-by-layer from the first layers.

	GTSRB	SVHN	Average
MTZ	1.14%	6.80%	3.97%

Table 2.6 Test errors of audio emotion classification network, visual emotion classification network and the joint network merged by MTZ. $1\times$ is the number of parameters of one single ResNet excluding the last classification layer.

Model	ResNet-28-Audio	ResNet-28-Video	ResNet-28-Fusion	MTZ
#par.	1.0×	1.0 ×	2.0 ×	1.1 ×
Test Error	43.02%	39.65%	24.74%	28.42%

2.5.3 Zipping Two Networks for Different Input Domains

This experiment evaluates the performance of MTZ to merge two models for different input domains. The aim is to show the potential memory saving to share information among different models, even if they are designed for different input domains, *e.g.*, one for audio and the other for visual input. This is common in mobile and ubiquitous computing with multiple sensing modalities.

Dataset and Settings. We experiment with an audio-visual emotion classification task, where we perform emotion recognition from speech and facial expression [28]. We use the same models as [28], where one ResNet-28 network is used to extract audio representations and the other ResNet-28 network is used to extract facial features from video frames. The audio and video features are concatenated and fed to full connected layers for emotion recognition. The performance of emotion recognition is assessed on the RML audio-visual dataset [30]. The RML database contains 720 utterances from 8 participants with 6 emotions: anger, disgust, fear, joy, sadness, and surprise.

We first train the two ResNet-28 networks for audio emotion classification (ResNet-28-Audio) and video emotion classification (ResNet-28-Video) respectively. The two models are then merged via full connected layers and fine-tuned for audio-emotion classification (ResNet-28-Fusion). Finally we enforce sharing 90% of the neurons in the two networks (the last classification layer excluded) via MTZ, which leads to a joint model of $1.1\times$ the size of a single ResNet-28 network.

Results. Table 2.6 shows the accuracy of different models on emotion classification. Comparing ResNet-28-Fusion with ResNet-28-Audio and ResNet-28-Video, the emotion recognition error drops significantly from around 40% to about 25%. However, this accuracy gain is at the cost of double the size of the model, *i.e.*, having twice the number of parameters of a single ResNet-28 network. Our MTZ method is able to enforce information sharing between ResNet-28-Audio and ResNet-28-Video. Specifically, compared with ResNet-

Table 2.7 A brief description of the datasets that each ResNet-28 network is trained on.

Dataset	Brief Description
CIFAR100 [61]	$60,000$ colour images of 100 different object categories
GTSRB [59]	$50,000+$ images for 43 traffic sign classes in different resolutions
Omniglot [62]	$1,623$ different handwritten characters from 50 different alphabets
SVHN [60]	$70,000$ images of digits cropped from street views
UCF101 [63]	$13,320$ videos clips collected from YouTube for 101 action categories
Flowers102 [64]	$8,189$ images of 102 categories of flowers
DPed [65]	$50,000$ grey-scale images of pedestrians and non-pedestrians
DTD [66]	$5,640$ texture images of 47 terms (categories) *e.g.*, bubbly
FGVC-Aircraft [67]	$10,000$ images of 100 different aircraft models *e.g.*, Airbus A310

28-Fusion, our joint model only has 1.1 times the number of the parameters (in contrast to 2 times), with only a 3.68% drop in emotion recognition accuracy. The results indicate that *inter-redundancy* is not limited to models with the same input domain and MTZ is able to suppress inter-redundancy among models for different input domains.

2.5.4 Zipping More Than Two Networks

This experiment evaluates the scalability of MTZ and the benefit of cross-model compression for running multiple models on embedded platforms. We investigate: *(i)* what the accuracy loss is if more than two models are merged; and *(ii)* what is the reduction in the model-switching time on resource-limited devices if multiple models are merged.

Dataset and Settings. We adopt the models and datasets in [23], a recent work on multi-task learning with ResNets. Specifically, nine ResNet-28 networks [58] are trained for diverse image recognition tasks, including CIFAR100 [61], German Traffic Sign Recognition Benchmark (GTSRB) [59], Omniglot [62], Street View House Numbers (SVHN) [60], UCF101 [63], Flowers102 [64], Daimler Mono Pedestrian Classification Benchmark (DPed) [65], Describable Texture Dataset (DTD) [66] and FGVC-Aircraft [67]. Table 2.7 provides a brief summary of the datasets.

To test the scalability of MTZ, we enforce sharing 90% of the neurons in a single ResNet-28 network with the other eight models, and evaluate the accuracy of the joint model on each of the nine task.

To show the benefit of executing a compact joint model, we measure the delay when switching between the nine inference tasks on the Jetson Nano embedded platform [57]. A 32GB Sandisk microSD is connected to the platform to storage the neural networks. To perform inference tasks on-device, the corresponding model should be loaded from the microSD to the memory. When a new task is performed, the parameters of the new model are loaded and the old parameters in the memory are overwritten, as the memory resource is limited. Fig. 2.6 illustrates the setup to measure the execution time of the visual inference tasks

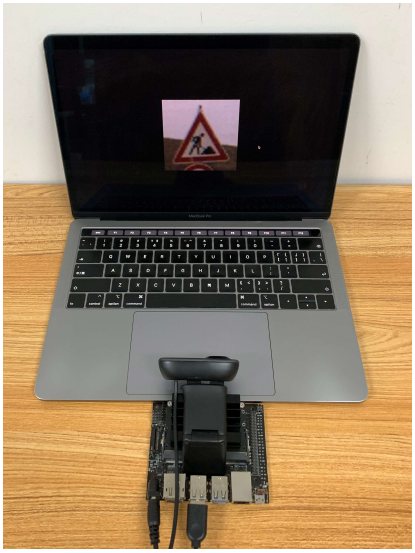


Figure 2.6 Hardware setup to measure the execution time of the nine inference tasks (the photo was for the traffic sign recognition task, *i.e.*, the GTSRB dataset [59]) on the Jetson Nano embedded platform.

Table 2.8 Test errors of pre-trained single ResNets and the joint network merged by MTZ. The joint model is compressed to $1.8\times$ of a single model. $1\times$ is the number of parameters of a single ResNet-28 excluding the last classification layer. Without MTZ the joint model would have a size of $9\times$.

	CIFAR100	GTSRB	Omniglot	SVHN	UCF101	Flowers102	DPed	DTD	FGVC	**Average**
Single	28.97%	0.56%	14.97%	6.04%	36.68%	37.45%	0.56%	67.61%	58.75%	**27.96%**
Joint	31.88%	0.51%	16.94%	6.70%	36.22%	37.35%	0.51%	67.71%	58.30%	**28.42%**

on the embedded platform. To measure the model-switching time, tasks are performed in a random sequence but each one from the 9 tasks is performed 10 times.

Results. Table 2.8 shows the accuracy of each individual pre-trained model and the joint model on the nine tasks. Compared with each individual model, the accuracy of the joint model only drops by 0.46% (averaged across the nine tasks). However, the total storage for the nine models decreases from $9\times$ to only $1.8\times$ of a single ResNet-28 network. The results show that MTZ is able to enforce neuron sharing among dozens of models for diverse tasks while retaining the inference accuracy on each task.

Fig. 2.7 shows the averaged time needed to update the parameters in the memory for new tasks. As 90% of the parameters are shared, only 10% of the parameters in the memory are needed to be updated when we use the joint model. Hence the model-switching time when using the joint model is significantly lower than that when using individual models. In general, the joint model is able to achieve $8.71\times$ lower model-switching time.

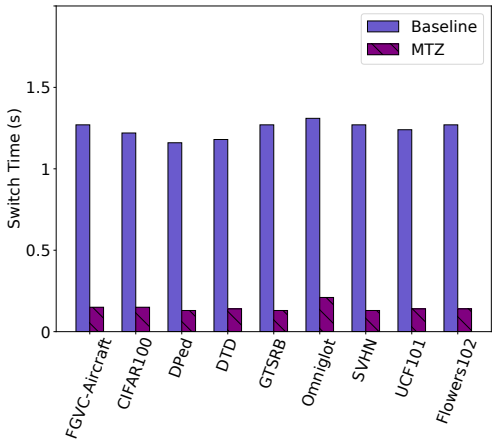


Figure 2.7 Averaged model-switching time between the nine tasks.

2.6 Conclusion

We propose MTZ, a framework to automatically merge multiple correlated, well-trained deep neural networks for cross-model compression via neuron sharing. It selectively shares neurons and optimally updates their incoming weights on a layer basis to minimise the errors induced to each individual task. Only light retraining is necessary to resume the accuracy of the joint model on each task. Evaluations show that MTZ can fully merge two VGG-16 networks with an error increase of 3.76% and 2.59% on ImageNet for object classification and CelebA for facial attribute classification, or share 39.61% parameters between the two models with $< 0.5\%$ error increase. The number of iterations to retrain the combined model is $17.9\times$ lower than that of training a single VGG-16 network. Meanwhile, MTZ can share 90% of the parameters among nine ResNets on nine different visual recognition tasks while inducing negligible loss in accuracy. The joint model also reduces the model-switching time between these inference tasks on memory-constrained devices by $8.71\times$. Experiments show that MTZ also applies to sparse networks and models for different input domains.

MTS is an effective weight sharing method for the first and second type of MMDL system, discussed in Sec. 1.4.1 and Sec. 1.4.2, respectively. And weight sharing is effective in reducing the number of parameters. However, as those models merged via MTZ must be concurrently executed on-device, the parallel execution of multiple models is necessary for optimal latency and energy consumption. As we will see in the next chapter, weight sharing among models creates challenges for such parallel execution. Existing parallel execution schemes do not fully support weight-shared models, and the resulted memory usage and latency are therefore suboptimal due to the forced duplication of shared weights. In Chapter 3, we will investigate this problem in detail and

provide a novel solution for the efficient execution of MTZ-merged models.

3

Multi-Task Stitching: Efficient On-Device Execution of Weight-Shared Models

With the help of our Multi-Task Zipping framework introduced in Chapter 2, we can construct weight-shared neural networks for multiple different yet correlated tasks for edge devices. Naturally, the next step is to try deploying these models on-device. However, as discussed in Sec. 1.4.2, it is a non-trivial problem to efficiently execute weight-shared neural networks, such as the networks merged by MTZ, on GPU enabled mobile and embedded edge devices. Most deep neural network (DNN) graph rewriters are blind for multi-DNN compression, while GPU vendors provide inefficient APIs for parallel multi-DNN execution at runtime. A few prior graph rewriters suggest cross-model graph fusion for low-latency multi-DNN execution. Yet they request duplication of the shared weights, erasing the memory saving of weight-shared DNNs.

In this chapter, We design Multi-Task Stitching (MTS), a novel graph rewriter for efficient multitask inference with weight-shared DNNs. MTS adopts a model stitching algorithm which outputs a single computational graph for weight-shared DNNs without duplicating any shared weight. MTS also utilises a model grouping strategy to avoid overwhelming the GPU when co-running tens of DNNs. Extensive experiments show that MTS accelerates multitask inference by up to $6.0\times$ compared to sequentially executing multiple weight-shared DNNs. MTS also yields up to $2.5\times$ lower latency and $3.7\times$ less memory usage than NETFUSE, a state-of-the-art multi-DNN graph rewriter.

3.1 Introduction

Deep learning empowered ubiquitous applications increasingly demand the co-execution of *multiple* deep neural networks (DNNs), known as *multitask inference*, for complex cognitive analysis [5], [6], [21], [68], [69]. In multitask

inference, multiple DNNs, each pre-trained for a single inference task, run concurrently on resource-constrained platforms ranging from edge servers [69], [70] to embedded devices [5], [6], [21], [29] for *correlated* inference tasks. Such multitask inference is critical for future applications such as smart glasses that identify user attributes *e.g.*, age, gender, face, and recognise objects [5], [22], [29], [69], [71], personal robots that classify places and sounds [6], autonomous vehicles that perceive the surroundings with front, side, rear camera views [68], home hubs that recognise emotions from speech and facial expression [28], [72] etc.

For efficient execution on low-resource platforms, DNNs often undergo multiple levels of optimisations. At the *model* level, over-parameterised DNNs can be compressed without loss in inference accuracy [8], [73]. The compressed DNNs, typically represented as computational directed acyclic graphs (DAGs), are then optimised at the *graph* level via sub-graph fusion and substitution to generate functionally equivalent yet faster DAGs for the target hardware platform [18], [74]. The DAGs can be further optimised at the *runtime* level for better resource utilisation via hardware-aware scheduling [75], [76]. Mainstream deep learning development frameworks such as TensorFlow [77] and PyTorch [78] support automatic and customised model- or graph-level optimisations whereas hardware vendors like NVIDIA also provide APIs for user-specified runtime-level accelerations. Despite extensive research on efficient DNN execution [13], [46], [73], [79]–[81], most efforts focus on accelerations *within a single model*, overlooking the potential gains from *cross-model* optimisation.

An emerging technique for efficient multitask inference is cross-model weight sharing [6], [20], [22], [29], [72], [82]. Sharing weights across DNNs pre-trained for correlated inference tasks reduces the memory footprint to deploy them on low-memory devices. Task correlation is pervasive since multiple DNNs may take the same input to generate different labels, or augment complimentary inputs to jointly output a single label. For example, DNNs that identify user age and faces from the same input image may extract similar low-level features, while DNNs for video- and audio-based emotion recognition may share similar high-level features. As illustrated in Fig. 3.1(b), cross-model weight-sharing methods automatically identify correlated weights (coloured in green) among weight matrices pre-trained for different tasks (see Fig. 3.1(a)). Such weight-shared DNNs significantly save the storage for multitask inference.

However, the memory saving of weight-shared DNNs fails to translate into efficient execution with existing graph- and runtime-level optimisations. On the one hand, popular graph rewriters such as TVM [18] and NVIDIA TensorRT [74] optimise each DAG in isolation (see Fig. 3.1(c)). Such graph rewriting duplicates the shared weights to create independent DAGs for each task. On the other hand, native runtime APIs such as CUDA Stream [83] and NVIDIA MPS [84] offer limited multi-DNN parallelism support. Executing individual DAGs as multiple streams leads to not only high memory cost, but also latency almost as large as executing these DAGs sequentially (see Fig. 3.1(f)-(g)). In

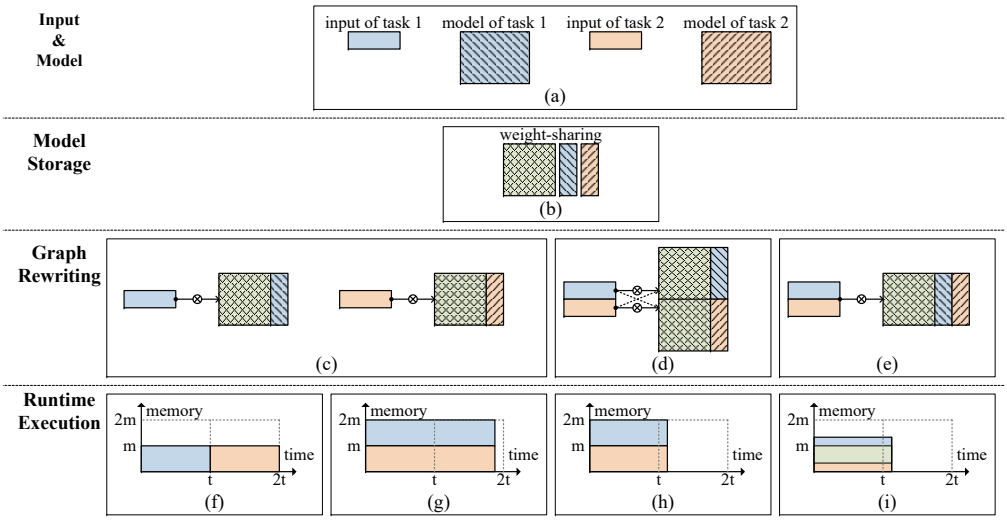


Figure 3.1 Executing weight-shared neural networks on a GPU. (a) Input and pre-trained single-task models for two correlated tasks 1 and 2. (b) Cross-model weight-sharing for storage saving of models. The green portion represents the shared weights uncovered by techniques such as [6], [22], [29]. Multitask inference on weight-shared networks can be compiled as (c) two separate computational graphs; (d) a single computational graph by duplicating shared weights; or (e) a single computational graph without duplicating shared weights. At runtime, (f) sequential execution of two graphs incurs long latency; (g) parallel execution of two graphs leads to both high memory footprint and long latency due to inefficient GPU runtime support; (h) the state-of-the-art executes a combined graph with a single stream, but duplicates the shared weights. (i) Our objective is to achieve both low memory and low latency.

fact, the state-of-the-art multi-DNN graph rewriters [32], [85] suggest cross-model fusion into a single DAG (see Fig. 3.1(d)) to comply with the default one-DNN-per-stream execution logic in most deep learning frameworks [68]. Yet these multi-DNN graph rewriters duplicate the shared weights, thus erasing the memory saving of weight-shared DNNs (see Fig. 3.1(h)).

In this chapter, we explore graph rewriting strategies dedicated to weight-shared DNNs for efficient multitask inference. Specifically, we aim to generate a single DAG for weight-shared DNNs without duplicating the shared weights (see Fig. 3.1(e)) to achieve both low latency and memory at runtime when executed on GPU (see Fig. 3.1(i)). We focus on graph-level optimisation to induce minimal changes and dependency to the runtime. Advanced multi-DNN runtime optimisations [70], [71], [86] are often complex to implement and rely on hardware-specific APIs such as CUDA Stream [83], which are inaccessible on platforms like mobile GPUs [87].

We design Mulit-Task Stitching (MTS), a novel cross-model graph rewriting framework for efficient multitask inference with weight-shared DNNs. The core of MTS is a model stitching algorithm which outputs a single DAG for multiple DNNs without duplicating their shared weights, which minimises the runtime memory. MTS also incorporates a model grouping strategy to organise

multiple models in groups to saturate, yet not overwhelm the GPU. Our main contributions and results are as follows.

- We are the first to address the problem of runtime memory saving for cross-model weight sharing.
- We propose MTS, which preserves the benefits of cross-model weight sharing to minimise runtime memory usage, and achieves pseudo-parallelism for low latency.
- Experiments are conducted on different hardware platforms, numbers of tasks, network architectures, pruning ratios, sharing ratios, batch sizes and heterogeneity. Results show that MTS is able to accelerate up to $6.0\times$ compare to sequentially executing multiple weight-shared DNNs. MTS also yields up to $2.5\times$ lower latency and $3.7\times$ less memory usage compared with state-of-the-art multi-DNN graph rewriter [32].

In the rest of this chapter, we state our problem in Sec. 3.2, introduce the MTS overview in Sec. 3.3, and elaborate on its model stitching and grouping schemes in Sec. 3.4 and Sec. 3.5, respectively. We present the evaluations in Sec. 3.6, review related work in Sec. 3.7, and finally conclude in Sec. 3.8.

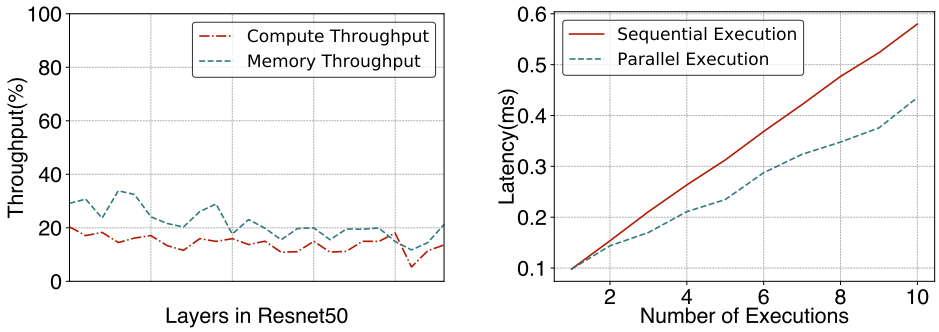
3.2 Problem Statement

We focus on graph rewriting of weight-shared DNNs for efficient multitask inference on single-GPU platforms. We justify our objectives and scope in details below.

Objectives. We use *runtime memory* and *overall latency* to assess the efficiency of multitask inference. Specifically, we would like to preserve the memory saving of cross-model weight sharing *i.e.*, the benefits of weight-shared DNNs [6], [20], [22], [29], [72], [82], while achieving low latency when performing multiple inference tasks.

Scope. We target at efficient multitask inference on devices equipped with a single GPU (either desktop- or mobile-grade) by implementing *pseudo-parallelism* at the graph-level optimisation of DNNs.

- We focus on GPUs because they are common hardware accelerators widely deployed to even low-resource devices. However, inference with mult[88] or heterogeneous resources [5], [68] is out of our scope.
- We aim at high *parallelism* to execute multiple DNNs for low overall latency. Improving parallelism is a tangible strategy because DNN inference often under-utilises the GPUs due to low computation density of operations and too few inputs for batching [70], [85], [87], [89]. Such under-utilisation exists in both desktop-[70] and mobile-grade [87], [90] GPUs. As an example, the NVIDIA GEFORCE RTX 2080 Ti GPU suffers from severe resource under-utilisation



(a) **(b)**

Figure 3.2 Opportunities and challenges of parallel DNN execution. (a) GPU utilisation with ResNet50. (b) Inference latency of a single fully connected layer using parallelism support available on commodity GPU runtime.

when executing ResNet50, a representative convolutional neural network (see Fig. 3.2a).

■ We resort to implement *pseudo-parallelism* at the *graph* level rather than the *runtime* level due to the limited parallelism API support provided by GPU vendors. Parallelism APIs for desktop GPUs such as CUDA Stream [83] and NVIDIA MPS [84] allow parallel execution of multiple inference tasks, with each model running in a different stream. Nevertheless, such runtime-level parallelism incurs non-trivial contentions [70], scheduling overhead [86], and kernel launch overhead [32], which dominate the overall latency of multitask inference. Even worse, such parallelism APIs are unavailable in mobile GPUs [87]. As a toy example, we measure the latency of executing a single fully connected layer for multiple times. As shown in Fig. 3.2b, parallel execution with multiple CUDA streams fails to deliver the expected acceleration over sequential execution.

3.3 MTS Overview

This section presents the overview of Multi-Task Stitching (MTS), a graph rewriting scheme for low latency, low runtime memory multitask inference on GPU-enabled devices. We illustrate our solution with MTZ [29], [72], a recent method to generate weight-shared DNNs. Our solution also applies to other cross-model weight-sharing schemes [6], [22], [82].

3.3.1 Notations

For ease of presentation, we explain our methods with fully-connected (FC) layers and extend to other layers in Sec. 3.4.2.

Consider T models $\{M_t\}$ where $1 \leq t \leq T$. Each model is a well-trained DNN for an inference task t. Let $\mathbf{F}_t^l$ and $\mathbf{A}_t^l$ be the feature map (weight matrix for

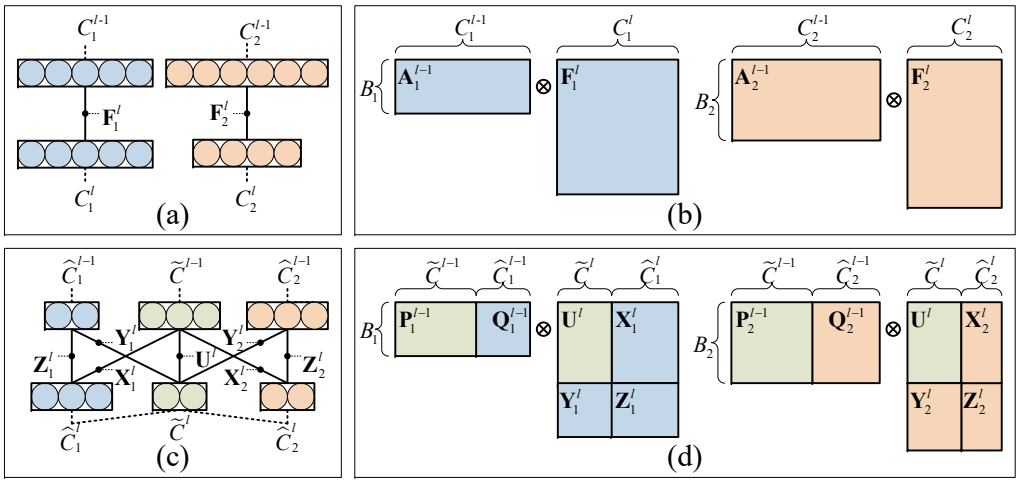


Figure 3.3 Notations for two tasks 1 and 2. (a) Layer $(l-1)$ and l without weight-sharing as well as (b) the corresponding dimensions of the activations at layer $(l-1)$ and the weights at layer l. (c) Layer $(l-1)$ and l with weight-sharing as well as (d) the corresponding dimensions of the activations at layer $(l-1)$ and the weights at layer l.

FC layers) and the output activation in layer l $(0 \leq l \leq L)$ of model M_t. Then the input activation of layer l is $\mathbf{A}_t^{l-1}$, *i.e.*, the output activation of layer $(l-1)$. Accordingly, $\mathbf{A}_t^0$ represents the input of M_t. Further assume for M_t, the input batch size is B_t and the number of neurons in layer l is C_t^l. Then in layer l, the input activation $\mathbf{A}_t^{l-1} \in \mathbb{R}^{B_t \times C_t^{l-1}}$, and the weight matrix $\mathbf{F}_t^l \in \mathbb{R}^{C_t^{l-1} \times C_t^l}$.

Without cross-model weight-sharing, the weight matrices $\{\mathbf{F}_t^l\}$ of multiple models $\{M_t\}$ are stored separately (see Fig. 3.3a for layer l of two models). The corresponding computations are also performed as separate computational graphs, *i.e.*, $\mathbf{A}_t^{l-1} \times \mathbf{F}_t^l$ at layer l for each M_t, where $\times$ is matrix multiplication, as shown in Fig. 3.3b.

With cross-model weight-sharing like [29], [72], each model shares certain amount of neurons on a layer basis while keeping other neurons exclusive. Concretely, the weight matrix $\mathbf{F}_t^l$ at layer l of model M_t is split into four portions: $\mathbf{U}^l \in \mathbb{R}^{\widetilde{C}^{l-1} \times \widetilde{C}^l}$, $\mathbf{X}_t^l \in \mathbb{R}^{\widetilde{C}^{l-1} \times \widehat{C}_t^l}$, $\mathbf{Y}_t^l \in \mathbb{R}^{\widehat{C}_t^{l-1} \times \widetilde{C}^l}$, and $\mathbf{Z}_t^l \in \mathbb{R}^{\widehat{C}_t^{l-1} \times \widehat{C}_t^l}$, where $\widetilde{C}^l$ and $\widehat{C}_t^l$ are the number of shared and task-t-exclusive neurons at layer l, which are automatically determined by cross-model weight-sharing algorithms. Matrix $\mathbf{U}^l$ is shared among models and thus only one copy is stored, as shown in Fig. 3.3c. During computation, the input activation $\mathbf{A}_t^{l-1}$ is split into two portions: $\mathbf{P}_t^{l-1} \in \mathbb{R}^{B_t \times \widetilde{C}^{l-1}}$ and $\mathbf{Q}_t^{l-1} \in \mathbb{R}^{B_t \times \widehat{C}_t^{l-1}}$. The $\mathbf{P}_t^{l-1}$ contains all shared neurons of activation $\mathbf{A}^{l-1}$, while $\mathbf{Q}_t^{l-1}$ contains the remaining exclusive neurons. Fig. 3.3d illustrates the dimensions of each matrix. Specifically, $\mathbf{U}^l$ transfers input activation's shared neurons to output activation's shared neurons. Similarly, $\mathbf{X}_t^l$ transfers shared neurons to exclusive ones, $\mathbf{Y}_t^l$ transfers exclusive neurons to shared ones, and $\mathbf{Z}_t^l$ transfers exclusive neurons to exclusive ones. Table 3.1 summarises the major notations.

Table 3.1 Summary of major notations.

Notation	Explanation
t, M_t, B_t	task t, its model M_t with input batch size B_t
$\mathbf{F}_t^l$, $\mathbf{A}_t^l$	kernels/weights, output activations of layer l of M_t
C_t^l, H_t^l, W_t^l	# neurons/channels, height, width for $\mathbf{A}_t^l$
K_t^l	kernel size for $\mathbf{F}_t^l$
$\widetilde{C}^l$, $\widehat{C}_t^l$	# shared, exclusive neurons of layer l of M_t

Takeaways. As shown in Fig. 3.3c, cross-model weight-sharing saves storage of weight matrices, *i.e.*, $\mathbf{U}^l$. However, such storage saving does not readily translate into runtime GPU memory saving, due to the lack of a cross-model computational graph rewriter. Naive execution of weight-shared DNNs as Fig. 3.3d duplicates the shared matrix $\mathbf{U}^l$, and regards weight-shared DNNs as multiple separate computational graphs. Such execution may incur high runtime memory footprint and long latency. Our solution is to *stitch* both the activations and weight matrices of multiple models as a single computational graph without duplicating the shared matrix $\mathbf{U}^l$ for efficient execution on GPU, as explained in detail next.

3.3.2 Functional Modules

MTS aims at pseudo-parallelism of weight-shared DNNs at the graph level. This is realised by stitching the activations and weight matrices of multiple tasks into a single stream to better utilise the GPU. The core design of MTS consists of two complementary schemes (see Fig. 3.4).

- A **model stitching** scheme (Sec. 3.4) that reconstructs multiple computational graphs into a single one without duplicating shared weights, for spatial multiplexing of the GPU among multiple models in parallel.
- A **model grouping** scheme (Sec. 3.5) that determines which models to be grouped for stitching without overwhelming the available resources, where groups of models are sequentially executed with high utilisation, *i.e.*, temporal multiplexing the GPU among model groups.

3.4 Model Stitching

The model stitching scheme combines the separate computational graphs into a single one without duplicating the shared weights. It demands cross-model stitching of both the input/output activations and the weight matrices. We present the stitching methods for two FC layers (Sec. 3.4.1), other common layer types (Sec. 3.4.2), and finally discuss the scalability to stitch more than two models (Sec. 3.4.3).

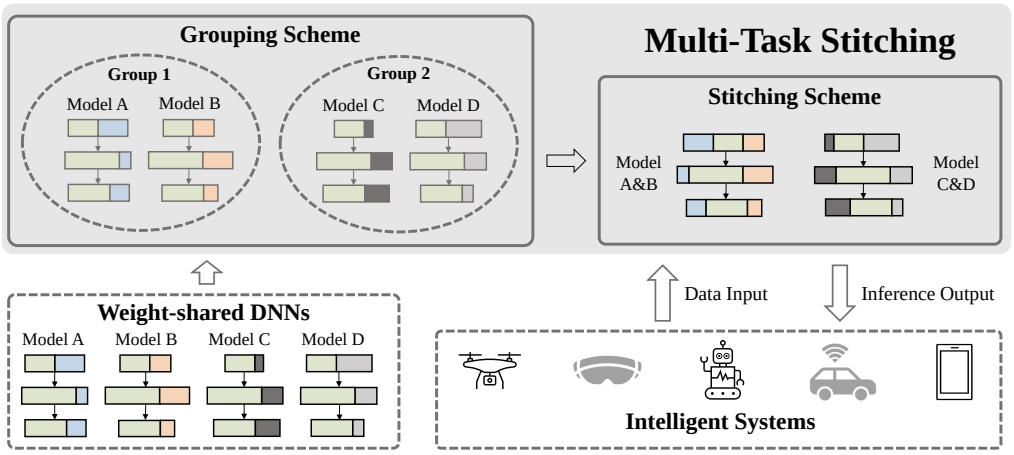
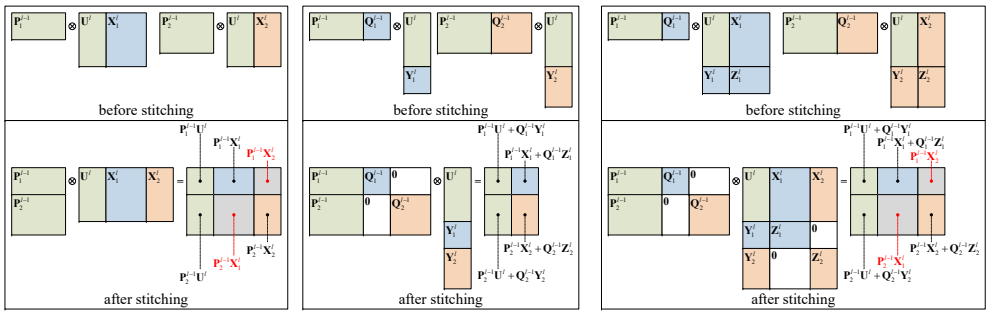


Figure 3.4 Workflow of MTS.



(a) **(b)** **(c)**

Figure 3.5 Illustrations of model stitching for (a) special case 1; (b) special case 2; and (c) the general case.

3.4.1 Stitching Two Fully Connected Layers

Consider the two fully connected (FC) layers in Fig. 3.3d. Our objective is to reconstruct the input activations $\mathbf{A}_1^{l-1}$ and $\mathbf{A}_2^{l-1}$ as $\mathbf{A}_{Stitch}^{l-1}$, and the weight matrices $\mathbf{F}_1^l$ and $\mathbf{F}_2^l$ as $\mathbf{F}_{Stitch}^l$, which still results in valid matrix multiplication $\mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^l$, without duplicating the shared weights $\mathbf{U}^l$.

We start with the two special cases which inspire our stitching method for the general case.

3.4.1.1 Special Case 1

Layers l of the two models share all input neurons, *i.e.*, $\widehat{C}_1^{l-1} = \widehat{C}_2^{l-1} = 0$. This is the case when the last layer is fully merged. Accordingly, the total number of input neurons is $C_1^{l-1} = C_2^{l-1} = \widetilde{C}^{l-1}$. The input activations $\mathbf{A}_1^{l-1} = \mathbf{P}_1^{l-1}$ and $\mathbf{A}_2^{l-1} = \mathbf{P}_2^{l-1}$. The weight matrices are simplified as $\mathbf{F}_1^l = \begin{bmatrix}\mathbf{U}^l & \mathbf{X}_1^l\end{bmatrix}$ and $\mathbf{F}_2^l = \begin{bmatrix}\mathbf{U}^l & \mathbf{X}_2^l\end{bmatrix}$ (Fig. 3.5a top).

We can concatenate input activation along the batch-size dimension (Fig. 3.5a

bottom).

$$\mathbf{A}_{Stitch}^{l-1} = \begin{bmatrix} \mathbf{P}_1^{l-1} \\ \mathbf{P}_2^{l-1} \end{bmatrix} \tag{3.1}$$

For valid matrix multiplication, we can concatenate the weight matrices as (Fig. 3.5a bottom):

$$\mathbf{F}_{Stitch}^{l} = \begin{bmatrix} \mathbf{U}^l & \mathbf{X}_1^l & \mathbf{X}_2^l \end{bmatrix} \tag{3.2}$$

Multiplying the two stitched matrices, we have

$$\mathbf{A}_{Stitch}^{l} = \mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^{l} = \begin{bmatrix} \mathbf{P}_1^{l-1}\mathbf{U}^l & \mathbf{P}_1^{l-1}\mathbf{X}_1^l & \mathbf{P}_1^{l-1}\mathbf{X}_2^l \\ \mathbf{P}_2^{l-1}\mathbf{U}^l & \mathbf{P}_2^{l-1}\mathbf{X}_1^l & \mathbf{P}_2^{l-1}\mathbf{X}_2^l \end{bmatrix}$$

Comparing the original calculations without stitching:

$$\mathbf{A}_1^l = \mathbf{A}_1^{l-1} \times \mathbf{F}_1^l = \begin{bmatrix} \mathbf{P}_1^{l-1}\mathbf{U}^l & \mathbf{P}_1^{l-1}\mathbf{X}_1^l \end{bmatrix}$$
$$\mathbf{A}_2^l = \mathbf{A}_2^{l-1} \times \mathbf{F}_2^l = \begin{bmatrix} \mathbf{P}_2^{l-1}\mathbf{U}^l & \mathbf{P}_2^{l-1}\mathbf{X}_2^l \end{bmatrix}$$

From special case 1, we make the following observations.

- We can obtain the output activations of the two models from $\mathbf{A}_{Stitch}^{l}$ with simple matrix rearrangements.
- The stitching strategy introduces certain redundant calculations, *i.e.*, $\mathbf{P}_1^{l-1}\mathbf{X}_2^l$ and $\mathbf{P}_2^{l-1}\mathbf{X}_1^l$.

3.4.1.2 Special Case 2

Layers l of the two models share all output neurons, *i.e.*, $\widehat{C}_1^l = \widehat{C}_2^l = 0$. This may take place when the next layer is fully merged. In this case, we cannot simplify the input activations. However, since the weight matrices can be represented by $\mathbf{F}_1^l = \begin{bmatrix} \mathbf{U}^{l^T} & \mathbf{Y}_1^{l^T} \end{bmatrix}^T$, $\mathbf{F}_2^l = \begin{bmatrix} \mathbf{U}^{l^T} & \mathbf{Y}_2^{l^T} \end{bmatrix}^T$ (Fig. 3.5b top). Naturally, we may concatenate the weight matrices along the the output-neuron dimension (Fig. 3.5b bottom) since we have $C_1^l = C_2^l = \widetilde{C}^l$.

$$\mathbf{F}_{Stitch}^{l} = \begin{bmatrix} \mathbf{U}^l \\ \mathbf{Y}_1^l \\ \mathbf{Y}_2^l \end{bmatrix} \tag{3.3}$$

Stitching of the input activations, however, is slightly more difficult due to the unequal numbers of input neurons. An intuitive solution is to expand $\mathbf{A}_1^{l-1}$ and $\mathbf{A}_2^{l-1}$ with additional zeros. Thus, the stitched input activation is (Fig. 3.5b bottom).

$$\mathbf{A}_{Stitch}^{l-1} = \begin{bmatrix} \mathbf{P}_1^{l-1} & \mathbf{Q}_1^{l-1} & \mathbf{0} \\ \mathbf{P}_2^{l-1} & \mathbf{0} & \mathbf{Q}_2^{l-1} \end{bmatrix} \tag{3.4}$$

The stitched output activation is calculated as

$$\mathbf{A}_{Stitch}^{l} = \mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^{l} = \begin{bmatrix} \mathbf{P}_1^{l-1}\mathbf{U}^l + \mathbf{Q}_1^{l-1}\mathbf{Y}_1^l \\ \mathbf{P}_2^{l-1}\mathbf{U}^l + \mathbf{Q}_2^{l-1}\mathbf{Y}_2^l \end{bmatrix}$$

Note that the top half of $\mathbf{A}^l_{stitch}$ is exactly the layer l output activation of M_1, *i.e.*, $\mathbf{A}^{l-1}_1 \times \mathbf{F}^l_1$, while the bottom half is exactly the layer l output activation of M_2, *i.e.*, $\mathbf{A}^{l-1}_2 \times \mathbf{F}^l_2$.

From special case 2, we make the following observations.

- We can obtain the output activations of the two models from $\mathbf{A}^l_{Stitch}$ without redundant calculations.
- Stitching introduces extra zeros in the input activations.

3.4.1.3 General Case

Now we consider the general case to stitch two FC layers. The input activations and the weight matrices are as follows (Fig. 3.5c top):

$$\mathbf{A}^{l-1}_1 = \begin{bmatrix}\mathbf{P}^{l-1}_1 & \mathbf{Q}^{l-1}_1\end{bmatrix}, \quad \mathbf{A}^{l-1}_2 = \begin{bmatrix}\mathbf{P}^{l-1}_2 & \mathbf{Q}^{l-1}_2\end{bmatrix},$$
$$\mathbf{F}^l_1 = \begin{bmatrix}\mathbf{U}^l & \mathbf{X}^l_1 \\ \mathbf{Y}^l_1 & \mathbf{Z}^l_1\end{bmatrix}, \quad \mathbf{F}^l_2 = \begin{bmatrix}\mathbf{U}^l & \mathbf{X}^l_2 \\ \mathbf{Y}^l_2 & \mathbf{Z}^l_2\end{bmatrix}$$

The input activations are in the same form as special case 2, and we can stitch inputs following Eq.(3.4). It is less obvious how to stitch the weight matrices. We can deduce from the two special cases that the stitched weight matrix $\mathbf{F}^l_{Stitch}$ should follow the structure below.

$$\mathbf{F}^l_{Stitch} = \begin{bmatrix}\mathbf{U}^l & \mathbf{X}^l_1 & \mathbf{X}^l_2 \\ \mathbf{Y}^l_1 & ? & ? \\ \mathbf{Y}^l_2 & ? & ?\end{bmatrix}$$

Taking the matrix size and the output values into account, we stitch $\mathbf{Z}^l_1$ and $\mathbf{Z}^l_2$ into $\mathbf{F}^l_{Stitch}$ into the bottom-right as follows (Fig. 3.5c bottom).

$$\mathbf{F}^l_{Stitch} = \begin{bmatrix}\mathbf{U}^l & \mathbf{X}^l_1 & \mathbf{X}^l_2 \\ \mathbf{Y}^l_1 & \mathbf{Z}^l_1 & \mathbf{0} \\ \mathbf{Y}^l_2 & \mathbf{0} & \mathbf{Z}^l_2\end{bmatrix} \tag{3.5}$$

Accordingly, the stitched output activation is calculated as

$$\mathbf{A}^l_{Stitch} = \mathbf{A}^{l-1}_{Stitch} \times \mathbf{F}^l_{Stitch} = \tag{3.6}$$
$$\begin{bmatrix}\mathbf{P}^{l-1}_1\mathbf{U}^l + \mathbf{Q}^{l-1}_1\mathbf{Y}^l_1 & \mathbf{P}^{l-1}_1\mathbf{X}^l_1 + \mathbf{Q}^{l-1}_1\mathbf{Z}^l_1 & \mathbf{P}^{l-1}_1\mathbf{X}^l_2 \\ \mathbf{P}^{l-1}_2\mathbf{U}^l + \mathbf{Q}^{l-1}_2\mathbf{Y}^l_2 & \mathbf{P}^{l-1}_2\mathbf{X}^l_1 & \mathbf{P}^{l-1}_2\mathbf{X}^l_2 + \mathbf{Q}^{l-1}_2\mathbf{Z}^l_2\end{bmatrix}$$

Discussions. We make the following notes on our scheme to stitch two FC layers, *i.e.*, Eq.(3.4) and Eq.(3.5).

- The stitched output activation involves redundant calculations, *i.e.*, $\mathbf{P}^{l-1}_1\mathbf{X}^l_2$ and $\mathbf{P}^{l-1}_2\mathbf{X}^l_1$ in Eq.(3.6). We quantify the impact of these redundant calculations in Sec. 3.4.3.
- If we set the redundant elements in $\mathbf{A}^l_{Stitch}$, *i.e.*, $\mathbf{P}^{l-1}_1\mathbf{X}^l_2$ and $\mathbf{P}^{l-1}_2\mathbf{X}^l_1$, to zeros, it just becomes the stitched input activation of the layer $l+1$. That is, we do not need to split and restitch the activations between layers.

3.4.2 Stitching More Than FC Layers

Now we extend our stitching strategy to layer types in representative convolutional neural networks (CNNs).

3.4.2.1 Stitching Convolutional Layers

Convolutional (CONV) layers are the primary building blocks for convolutional neural networks. Matrix multiplication can be considered as a special convolution with input feature map size and kernel size of $(1, 1)$. Therefore, neurons in a FC layer correspond to *channels* in a CONV layer. In a CONV layer, the input $\mathbf{A}_i^{l-1} \in \mathbb{R}^{B_t \times C_t^{l-1} \times H_t^l \times W_t^l}$ and weight $\mathbf{F}_t^l \in \mathbb{R}^{C_t^l \times C_t^{l-1} \times K_t^l \times K_t^l}$ are non-degenerate 4D tensors. Therefore, we can stitch CONV layers in the same way as FC layers. The only changes is to replace all 2D matrices with 4D tensors. This ensures the flexibility and efficiency of MTS.

Note that algorithms like im2col can convert convolution into matrix multiplication. Hence another solutions is to stitch the FC layers transformed from CONV layers. However, $im2col$ is only efficient when the kernel size is small and it may notably increase the memory footprint [91].As a result, we do not use this solution in the final implementation.

3.4.2.2 Stitching Residual Blocks

Residual blocks are the main components of residual networks. Different from FC Layers and CONV Layers, the residual block is a mixture of multiple layers. It primarily consists of several convolution operators and one addition operator. The major difference is that the residual block is a two-branch network instead of a series of base layers strung sequentially. Existing cross-model weight-sharing methods [72] merge residual blocks by sharing the same input/output neurons across these two branches. Therefore, we can stitch CONV layers along each branch independently and exploit the stitched addition operator (more details in the Sec. 3.4.2.3) to combine two branch outputs.

3.4.2.3 Stitching Other Layers/Operators

There are a few other common layers or operators in mainstream DNNs. We briefly discuss their stitching below.

- *Element-wise Binary Operators*. Element-wise binary operators are common in DNN models, *e.g.*, the residual block uses an addition operator to merge two branches. Note that we only stitch tensors/matrices over the neurons dimension. Therefore, our stitching scheme does not affect element-wise operators. We can directly apply these element-wise operators over stitched inputs.
- *Batch Normalisation (BN) Layers*. The BN layer is applied to the outputs

to keep their mean close to 0 and standard deviation close to 1. Assume the output of the previous CONV layer is x. The output of the BN layer can be written as

$$BN(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta$$

where μ, σ are the pre-calculated mean and standard deviation. γ, β are two learnable parameters. Since the parameters in the BN layer vary across models, it is impossible to stitch these BN layers into one. However, since the parameters are fixed after training, we can fuse BN layers into their respective previous CONV layer by modifying the weights and bias of this CONV layer. Thus stitching BN layers is transformed into stitching CONV layers, which has been solved in Sec. 3.4.2.1.

- *Activation/Pooling Layers*. Activation layers are applied for better fitting non-linear functions, while pooling layers are used to decrease data dimensions. Since the activation function and pooling function take a single element or a cluster of elements as input, our stitching algorithm does not interfere with these layers. The only trifle is that we add additional zeros in inputs and outputs, while activation functions like $Sigmoid(\cdot)$ map zeros to 0.5's. These zeros need to be preserved for correct feed-forward computation.

3.4.3 Stitching Multiple Layers

So far we have shown how to stitch layers of two models. We now use the FC layers to illustrate how $T > 2$ models are stitched. We can extend Eq.(3.4) to stitch input activations and Eq.(3.5) to stitch weight matrices as:

$$\mathbf{A}^{l-1}_{Stitch} = \begin{bmatrix} \mathbf{P}^{l-1}_1 & \mathbf{Q}^{l-1}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{P}^{l-1}_2 & \mathbf{0} & \mathbf{Q}^{l-1}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}^{l-1}_T & \mathbf{0} & \mathbf{0} & \dots & \mathbf{Q}^{l-1}_N \end{bmatrix} \tag{3.7}$$

$$\mathbf{F}^{l}_{Stitch} = \begin{bmatrix} \mathbf{U}^{l} & \mathbf{X}^{l}_1 & \mathbf{X}^{l}_2 & \dots & \mathbf{X}^{l}_T \\ \mathbf{Y}^{l}_1 & \mathbf{Z}^{l}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{Y}^{l}_2 & \mathbf{0} & \mathbf{Z}^{l}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{Y}^{l}_T & \mathbf{0} & \mathbf{0} & \dots & \mathbf{Z}^{l}_T \end{bmatrix} \tag{3.8}$$

Hence $\mathbf{A}^{l}_{Stitch}$, the result of $\mathbf{A}^{l-1}_{Stitch} \times \mathbf{F}^{l}_{Stitch}$, has a similar structure to $\mathbf{A}^{l-1}_{Stitch}$, from which we can obtain $\mathbf{A}^{l}_{t}$.

Analysis of Stitched Model. As mentioned in Sec. 3.4.1, the stitching scheme introduces redundant computation. We now estimate the overall computation of a stitched model.

Assume $\mathbf{A}^{l}_{Stitch}$ and $\mathbf{F}^{l}_{Stitch}$ have dimensions of $B_S \times C^{l-1}_S$ and $C^{l-1}_S \times C^{l}_S$.

Then B_S, C_S^{l-1}, and C_S^l are computed as:

$$B_S = \sum_{t=1}^{T} B_t \qquad C_S^{l-1} = \widetilde{C}^{l-1} + \sum_{t=1}^{T} \widehat{C}_t^{l-1} \qquad C_S^l = \widetilde{C}^l + \sum_{t=1}^{T} \widehat{C}_t^l$$

The floating point operations (FLOPs) of the stitched model is then estimated as

$$G_S^l = B_S \times C_S^{l-1} \times C_S^l$$

For simplicity, assume

$$\begin{aligned} B_1 = B_1 = \cdots = B_T = B \\ \widehat{C}_1^{l-1} = \widehat{C}_2^{l-1} = \cdots = \widehat{C}_T^{l-1} = (1-\alpha^{l-1}) \cdot C_t^{l-1} = (1-\alpha^{l-1}) \cdot C^{l-1} \\ \widehat{C}_1^{l} = \widehat{C}_2^{l} = \cdots = \widehat{C}_T^{l} = (1-\alpha^{l}) \cdot C_t^{l} = (1-\alpha^{l}) \cdot C^{l} \end{aligned} \tag{3.9}$$

where $\alpha^{l-1}, \alpha^l \in [0,1]$ are the sharing ratios in cross-model weight-sharing [29], [72]. Substituting with Eq.(3.6), the stitched weight/model size and the total FLOPs can be estimated as

$$\begin{aligned} \text{Model-Size} &= C_S^{l-1} \times C_S^l \\ &= \left(\alpha^{l-1} + T \cdot (1-\alpha^{l-1})\right)\left(\alpha^{l} + T \cdot (1-\alpha^{l})\right) \\ &\quad \cdot C^{l-1} C^l \\ \text{FLOPs} &= B_S \times C_S^{l-1} \times C_S^l \\ &= T\left(\alpha^{l-1} + T \cdot (1-\alpha^{l-1})\right)\left(\alpha^{l} + T \cdot (1-\alpha^{l})\right) \\ &\quad \cdot C^{l-1} C^l B \end{aligned}$$

Hence the stitched model size is $\mathcal{O}(T^2)$ with $\mathcal{O}(T^3)$ FLOPs.

Discussions. We make the following notes.

- Since the GPU is under-utilised [70], [87], [90] for inference, the redundant computation improves the throughput and reduces the overall latency of multitask inference compared with executing without stitching, as empirically validated in Sec. 3.6 (MTS against Sequential).

- The redundant computation does impair the scalability to more *e.g.*, tens of tasks, which motivates the model grouping design in Sec. 3.5 for temporal multiplexing the GPU without overwhelming. Experimental results show that with model grouping, our method achieves comparable latency with NETFUSE [32], the state-of-the-art multi-DNN graph rewriter, with notably lower runtime GPU memory usage.

- Note that the model size and FLOPs decrease with the sharing ratios α^{l-1} and α^l. It indicates that when the sharing ratios comes to 1, *i.e.*, all neurons shared, all the T models are exactly the same and MTS simply batches inputs. Therefore, MTS can be considered as a novel extension of input batching.

Algorithm 2: Model Grouping Algorithm

input : weight-shared models $M_t (1 \le t \le T)$, corresponding inputs A_t^0
output: model grouping scheme $\mathcal{G}$, $s.t.\ \forall M_t \in \{M_t \mid 1 \le t \le T\},\ \exists! \mathcal{G}_i \in \mathcal{G},\ M_t \in \mathcal{G}_i$

1 **if** *models have the same structure and input batch size* **then**
2 $\mathcal{F}_0 \leftarrow 0$;
3 **foreach** $n \in [2, T]$ **do**
4 $L_n \leftarrow$ latency to assign n models in one group;
5 $\mathcal{F}_n \leftarrow \min_{1 \le j \le n} \{L_n,\ \mathcal{F}_{n-i} + L_i\}$;
6 extract $\mathcal{G}$ from the derivation of $\mathcal{F}_T$;
7 **else if** T *is small* **then**
8 $\mathcal{H}_\emptyset \leftarrow 0$;
9 **foreach** $\mathcal{S} \subseteq \{M_t \mid 1 \le t \le T\}$ **do**
10 $L_\mathcal{S} \leftarrow$ latency to assign all models in $\mathcal{S}$ in one group;
11 $\mathcal{H}_\mathcal{S} \leftarrow \min_{\mathcal{S}' \subset \mathcal{S}} \{L_\mathcal{S},\ \mathcal{H}_{\mathcal{S}'} + \mathcal{H}_{\mathcal{S} \setminus \mathcal{S}'}\}$;
12 extract $\mathcal{G}$ from the derivation of $\mathcal{H}_{\{M_t | 1 \le t \le T\}}$;
13 **else**
14 **foreach** $G \in [1, T]$ **do**
15 construct empty groups $\hat{\mathcal{G}} \leftarrow \{\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_G\}$;
16 set group-queue $[\mathcal{G}_1, \cdots, \mathcal{G}_{G-1}, \mathcal{G}_G, \mathcal{G}_G, \mathcal{G}_{G-1}, \cdots, \mathcal{G}_1, \mathcal{G}_1, \mathcal{G}_2, \cdots]$;
17 sort models in descending order of latency;
18 group models in the order of group-queue;
19 get $\hat{\mathcal{G}}$'s latency, update $\mathcal{G}$ if $\hat{\mathcal{G}}$ has shorter latency;
20 **return** $\mathcal{G}$

3.5 Model Grouping

In this section, we present the model grouping scheme of MTS. It facilitates efficient multitask inference with even tens of models. Specifically, the T models are organised into a groups $\mathcal{G} = \{\mathcal{G}_i \mid 1 \le i \le G\}$. The groups are executed sequentially, whereas models within a group are stitched and executed in parallel. For efficient grouping, we propose a greedy-based grouping scheme as illustrated in Algorithm 2. Its details are explained below.

Same Model Structure and Batch Size (Line 1-6). In this case, each model can be considered as the same inference task. Thus the latency of a group is only dependent on the number of tasks grouped. In line 2-6, we use dynamic programming to get the optimal grouping. Specifically, let $\mathcal{F}_n$ as the minimum latency of the first n models. The recursive definition of $\mathcal{F}_n$ in line 5 chooses the optimal scheme by assigning j tasks to one group, and the remaining $n - j$ tasks to other groups. Finally, we construct an optimal group scheme according to the computing progress of $\mathcal{F}_T$ (line 6).

Small Number of Tasks T (Line 7-12). In this case, we use state compression dynamic programming to get the optimal grouping scheme. Let $\mathcal{H}_\mathcal{S}$ be the minimum latency of all models in set $\mathcal{S}$. The recursive definition in line 11 explains how to find the optimal inference: either assign all models and execute in one time, or pick some ($\mathcal{S}' \subset \mathcal{S}$) models to execute first. The optimal group scheme can be constructed by the computing progress of $\mathcal{H}_{\{M_t | 1 \le t \le T\}}$ (line 12).

General Case (Line 14-19). In the most general case, we group the models

greedily. Specifically, we enumerate the total number of groups from 1 to T, and put all models into groups in a balanced way (Line 15-18). The best grouping scheme is chosen. The criterion is to balance the latency among groups.

Discussions. We make two notes on the model grouping.

- Despite the greedy grouping, it finds the optimal solution if the model size and batch sizes are the same, or with a small number of tasks (empirically set as 15).
- The algorithm requires inference latency estimates (Line 4, 10, 17). The latency can be estimated offline by direct measurements, or more efficiently, by exploiting latency estimators such as nn-Meter [92]. In our implementation, we measure the latency directly.

3.6 Evaluation

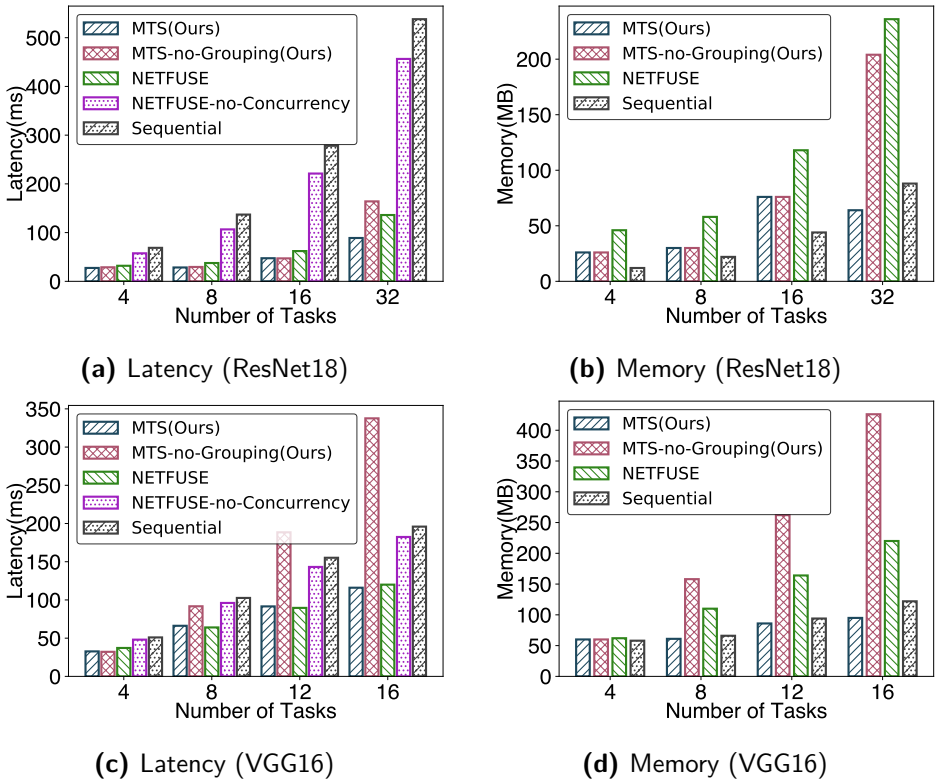
3.6.1 Evaluation Setup

Platforms and Implementation. We conduct experiments on two platforms: Jetson Tx2 and an edge server. Jetson Tx2 is a mobile computing platform equipped with a Quad-Core ARM Cortex-A57 MPCore (NVIDIA Denver 2 CPU was disabled during experiments) and 8GB RAM, as well as a 256-core Pascal-based GPU. The edge server is equipped with an 32-core Intel Xeon E5-2620@2.10GHz processor and 256GB RAM, as well as a NVIDIA GEFORCE RTX 2080 Ti. All the algorithms are implemented with PyTorch in Python.

Inference Workload. We experiment with three representative CNNs: ResNet18, ResNet50 and VGG16. Due to the limited resources on mobile and edge devices, we also considered pruned versions of these models. Note that cross-model weight-sharing schemes [29], [72] can also merge pruned models. To generate multitask inference workload, we merge a given number of the three CNN types, either pruned or unpruned, by MTZ [29]. We test different model number, pruning ratios, and sharing ratios. We consider a batch size of 1 since most inference tasks demand real-time processing. The detailed configurations are deferred to each experiment.

Baselines. We compare MTS with the following baselines.

- Sequential: It selects one model from all in a round-robin fashion and performs the inference one by one.
- NETFUSE [32]: It is the state-of-the-art multi-DNN graph rewriter. It leverages operations like group convolution, batch matrix multiplication and group normalisation for cross-model fusion.
- NETFUSE-no-Concurrency: It is the original NETFUSE with multi-stream execution disabled. It is to simulate the mobile GPU runtime because multi-stream execution is not supported by most mobile GPUs [87].



(a) Latency (ResNet18) **(b)** Memory (ResNet18)

(c) Latency (VGG16) **(d)** Memory (VGG16)

Figure 3.6 Performance comparisons on mobile platforms (Jetson Tx2).

- MTS-no-Grouping: It is MTS without the grouping.

Metrics. We use overall inference latency and peak memory footprint to assess the performance of each algorithm. The memory footprint of NETFUSE-no-Concurrency is omitted because it has the same result as NETFUSE. We measure the inference latency on GPU by calculating the average of 500 inference latency after 2 warmups, and we capture the peak memory footprint during execution with NVIDIA Nsight Systems[93]. Since the reconstruction of weights induces extra delay in inference, the models in Sequential, NETFUSE, and NETFUSE-no-Concurrency are not merged by cross-model weight-sharing schemes for fair comparison.

3.6.2 Evaluation Results

We now present our evaluation results with various platforms and workload settings.

3.6.2.1 Performance on Mobile Devices

In this experiment, we compare different algorithms on mobile devices using Jetson Tx2 as the evaluation platform.

Settings. We choose pruned ResNet18 and VGG16 as the model types. Each model has 90% neurons pruned (pruning ratio $= 0.9$) and all of them share 90% of the remaining neurons with each other (sharing ratio $= 0.9$). In this experiment, we set the input batch size of all models to 1, *i.e.*, each model receives exactly one image at a time. We vary the number of weight-shared models from 4 to 32.

Results. Fig. 3.6 shows the inference latency and peak memory footprint of ResNet18 and VGG16 on Jetson Tx2.

For ResNet18 (see Fig. 3.6a and Fig. 3.6b), MTS yields about $2.5\times$, $4.8\times$, $5.8\times$, $6.0\times$ speedup against Sequential, when executing 4, 8, 16 or 32 models, respectively. MTS consumes however more memory than Sequential when there are 4, 8 and 16 models. This is caused by the the temporary runtime memory dominating in total memory usage. When the number of tasks becomes larger than 32, MTS's memory footprint is reduced significantly and surpasses Sequential ($1.4\times$ memory saving), benefiting from the grouping scheme. Compared to NETFUSE, MTS is $1.2\times$, $1.3\times$, $1.3\times$ and $1.5\times$ faster. The latency is realtive close, but NETFUSE consumes much more memory: $1.8\times$, $1.9\times$, $1.6\times$, and $3.7\times$, respectively. As for MTS-no-Grouping, when there are 4, 8 or 16 tasks in total, it has the same performance as MTS because there is no grouping activated. When there are 32 tasks, the grouping scheme is activated and the tasks are grouped into three. Thus MTS saves $1.84\times$ latency and $3.18\times$ memory usage. At last, the latency of NETFUSE-no-Concurrency is almost the same as Sequential ($2.1\times$, $3.7\times$, $4.7\times$ and $5.1\times$ slower than MTS, respectively), indicating that NETFUSE is unfit for mobile platforms without multi-stream APIs.

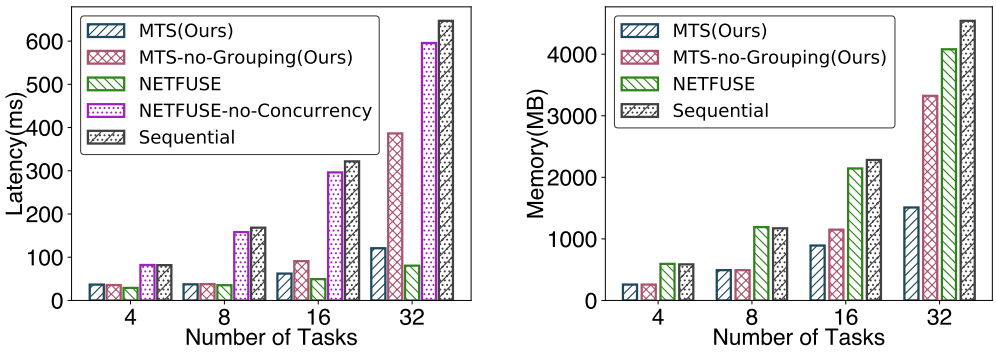
For VGG16 (see Fig. 3.6c and Fig. 3.6d), MTS achieves $1.5\times$ speedup and $1.3\times$ memory saving than Sequential. Compared with the speedup for ResNet18, the lower speedup is due to VGG16 being more computational intensive. For the same reason, the latency of MTS-no-Grouping even exceeds that of Sequential. The memory footprint of MTS-no-Grouping is also drastically larger than that of NETFUSE since the all-in-one stitched VGG16 model consumes large amounts of runtime memory.

In a word, MTS achieves the best latency-memory balance than the baselines. For comparison, MTS accelerates up to $6.0\times$ and saving $1.4\times$ memory compared to Sequential. As for NETFUSE, MTS is $1.5\times$ faster and $3.7\times$ memory saving. Furthermore, the results of MTS-no-Grouping demonstrate the necessity of model grouping.

3.6.2.2 Performance on Edge Servers

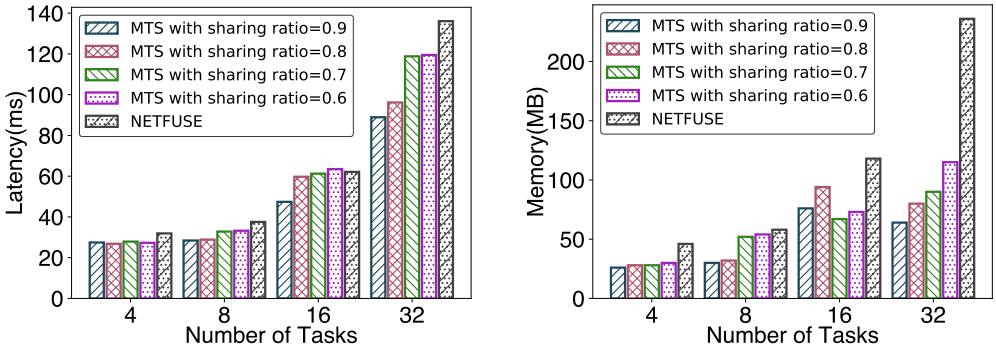
In this experiment, we test the algorithms on a desktop-grade GPU for edge servers.

Settings. We use a server equipped with an RTX 2080 Ti GPU as the platform,



(a) Latency (ResNet50) **(b)** Memory (ResNet50)

Figure 3.7 Performance comparisons on edge servers.

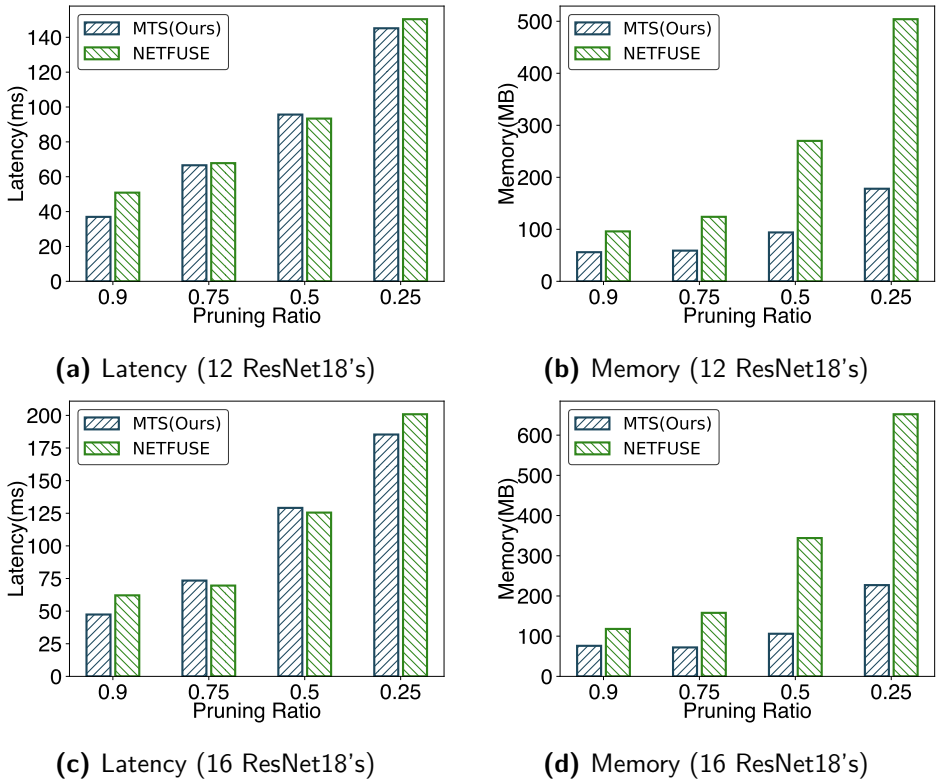


(a) Latency **(b)** Memory

Figure 3.8 Impact of sharing ratios using ResNet18 models on Jetson Tx2.

and ResNet50 as the model type. Due to the adequate resource on the desktop-grade GPU, we directly adopt the unpruned ResNet50 as the model for inference tasks. The sharing ratio remains 0.9 and the input batch size is also set to 1 as in Sec. 3.6.2.1.

Results. Fig. 3.7 plots the inference latency and peak memory footprint of each algorithm when executing 4 to 32 weight-shared ResNet50 models on the edge server platform. As with the results on Jetson TX2, MTS is $5.3\times$ faster than Sequential. However, MTS is slightly slower than NETFUSE. This may stem from the larger bandwidth and more CUDA cores, allowing faster execution of group convolution and batch matrix multiplication used by NETFUSE. In terms of memory usage, MTS still notably outperforms NETFUSE, consuming only 37% of memory NETFUSE does. Sequential's peak memory footprint grows drastically because the pruning ratio is 1 and it is the models' parameters rather than temporary runtime memory that dominates in the total memory footprint.



(a) Latency (12 ResNet18's) **(b)** Memory (12 ResNet18's)

(c) Latency (16 ResNet18's) **(d)** Memory (16 ResNet18's)

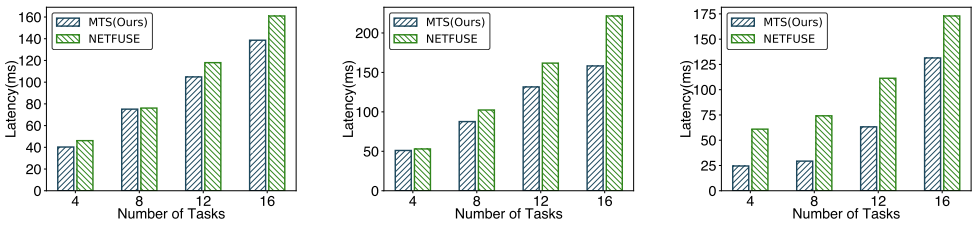
Figure 3.9 Performance with different pruning ratios.

3.6.2.3 Impact of Pruning Ratios

In this experiment, we assess the impact of pruning ratios on different algorithms. This is because DNNs tend to be compressed before deployment to mobile devices [8]. We use pruning ratios to simulate the inference workload of diverse pruned DNNs. Note that the pruning ratio is a configurable hyperparameter in many network pruning methods [8], [73].

Settings. We choose ResNet18 on Jetson Tx2 for this experiment. We set the pruning ratios for ResNet18 as 0.9, 0.75, 0.5 and 0.25. After pruning, the ResNet18 models are merged by MTZ [29] with a sharing ratio of 0.9 and the input batch size is 1, as in Sec. 3.6.2.1. We test the inference latency to execute 12 and 16 models. As our aim is to prove MTS is applicable across different pruning ratios, we only select the state-of-the-art algorithm NETFUSE as the baseline.

Results. Fig. 3.9 shows the performance of MTS and NETFUSE. The two methods have similar overall inference latency (see Fig. 3.9a and Fig. 3.9c), but MTS greatly outperforms NETFUSE in terms of memory usage (see Fig. 3.9b and Fig. 3.9d), similar to the results in Sec. 3.6.2.1. The results suggest that MTS can hold its efficient feed-forward and low memory footprint across DNNs with different widths.



(a) Pruning ratios=0.9/0.88 **(b)** Pruning ratios=0.9/0.85 **(c)** Batch sizes=1/2/3

Figure 3.10 Impact of model heterogeneity. (a) and (b): latency of VGG16's with mixed pruning ratios; (c) latency of ResNet18's with mixed batch sizes.

3.6.2.4 Impact of Sharing Ratios

Another hyperparameter of the inference workload is the sharing ratio among tasks. A large sharing ratio is set in case of high task relatedness or limited resources [29].

Settings. We use Jetson Tx2 as the platform, the pruned ResNet18 with a pruning ratio of 0.9 as the model type and set the input batch size to 1. We experiment with four sharing ratios: 0.9, 0.8, 0.7 and 0.6 and vary the number of tasks from 4, 8, 16 to 32.

Results. Fig. 3.8 shows the inference latency and peak memory footprint of ResNet18 with different sharing ratios on Jetson Tx2. When operating the same grouping scheme (number of tasks is 4, 8, or 32), the inference latency and the memory footprint of MTS decrease with the increase of sharing ratio, which is consistent with our analysis in Sec. 3.4.3. MTS achieves the lowest latency and memory cost with all sharing ratios. The gain is more notable with more models *e.g.*, 32. This is because the model grouping scheme is activated due to an excessively high computing need with large amounts of models. Then both the inference latency and runtime peak memory will be greatly reduced by model grouping of MTS.

3.6.2.5 Impact of Heterogeneity in Inference Workload

In this experiment, we test the impact of heterogeneity in inference workload on the performance of MTS. We consider two types of heterogeneity: mixed layer widths and input batch sizes.

Settings. We conduct two experiments on Jetson Tx 2.

- In the first experiment, we force MTS and NETFUSE to combine multiple VGG16's compressed with different pruning ratios. We experiment with two sets of pruning ratios: $0.9/0.88$, and $0.9/0.85$. MTS naturally functions with unequal layer widths. For NETFUSE to work with unequal layer widths, it should extend the narrower layers with redundant neurons. The sharing ratio is 0.9 and the input batch size is 1.

- In the second experiment, we vary the batch size from 1 to 3 and use ResNet18 as the model type. The stitching strategy of MTS is designed to handle different batch sizes. Nevertheless, NETFUSE needs to group these models according to the batch size and execute them one by one. We set the pruning ratio of the ResNet18 models to 0.9 and the sharing ratio to 0.9.

Results. Fig. 3.10a and Fig. 3.10b show that the inference latency of VGG16s with mixed pruning ratios. Comparing Fig. 3.10a,b with Fig. 3.6c, where a fixed pruning ratio of 0.9 is adopted for all VGG16 models, the gain in latency of MTS over NETFUSE becomes more notable. This advantage in latency gets clearer if the difference in pruning ratios is larger *i.e.*, by comparing Fig. 3.10a and Fig. 3.10b. The increased gain is because NETFUSE has to pad redundant neurons for combining models of different layer widths, whereas MTS incurs no extra overhead. Fig. 3.10c shows that the inference latency of ResNet18s with mixed input batch sizes. Compared with Fig. 3.6a, MTS is much more efficient than NETFUSE, especially when number of tasks is relatively small, *e.g.*, 4, 8. The forcing sequential execution of NETFUSE leads to severe resource under-utilisation.

3.6.2.6 Summary of Experimental Results

We summarise our main experimental findings as follows.

- MTS can accelerate Sequential up to 6.0 times.
- With the model grouping scheme, MTS's latency is improved by 3.2 times and the memory footprint is saved by 4.5 times than MTS-no-Grouping.
- Overall, MTS outperforms NETFUSE [32], the state-of-the-art multi-DNN graph rewriter in both inference latency (up to $1.5\times$) and runtime memory (over 3.7 times of saving). MTS advantages over NETFUSE is more notable with heterogeneity in inference workload *e.g.*, mixed model widths and input batch sizes, achieving up to $2.5\times$ speedup while retaining the same memory saving.

3.7 Related Work

Our work is related to the following categories of research.

3.7.1 Weight-Shared DNNs

Cross-model weight sharing facilitates DNN deployment to low-memory devices. It applies to DNNs for either a *single* task or *multiple* tasks. In single-task weight-shared DNNs, each DNN is often a model variant for the same inference task, yet of a different complexity-accuracy trade-off. Examples include early-exits [94], slimmable networks [95], nested architectures [21], etc.

In multitask weight-shared DNNs, each DNN is trained for a different inference task. Multitask weight sharing is feasible for correlated tasks. Given multiple DNNs well-trained for correlated tasks, weight sharing can be achieved by cross-model quantisation [20] or fine-tuning [6], [22], [29], [72]. We apply MTZ [29] to generate weight-shared DNNs, for it allows adaptive weight sharing and supports diverse layer types. Other studies either enforce full weight sharing [6] or support convolutional layers only [22].

Although weight-shared DNNs save storage, their execution may not speed up for multitask inference. Naive execution of such networks results in multiple data flows [96]. This leads to the same delay as running unshared DNNs sequentially, and erases the memory saving of weight sharing. MTS is the first attempt at efficient weight-shared DNN execution for multitask inference while retaining the memory saving.

3.7.2 Multi-DNN Graph Rewriting

Deep learning frameworks such as TensorFlow [77] and PyTorch [78] represent DNNs as computational directed acyclic graphs (DAGs). Graph rewriters such as TVM [18] and NVDIA TensorRT [74] apply automatically or manually configured graph substitution rules to output mathematically equivalent DAGs that run faster on the given hardware platform. Yet these rewriters optimise each DAG in isolation and are ineffective for multi-DNN graph rewriting [32], [85].

HiveMind [85] and NETFUSE [32] are two state-of-the-art multi-DNN graph rewriters. The idea is to perform cross-model layer fusion to increase the computational intensity of operations, and thus the GPU utilisation [85]. HiveMind [85] assumes the same input for the weight-shared DNNs. NETFUSE [32] eliminates such restrictions and support cross-model layer fusion of DNNs with different inputs and outputs. However, neither HiveMind [85] nor NETFUSE [32] retains the memory saving of weight sharing during layer fusion, and they pose constraints such as the same channel or input sizes on the DNNs. These drawbacks motivates the model stitching strategy in our MTS.

3.7.3 Multi-DNN Runtime Scheduling

Given DAGs as input, a multi-DNN runtime schedules the DAG operations to maximise the pipelined or parallel execution on the targeting platform. Despite multi-tenancy APIs such as CUDA stream [83] and NVIDIA MPS [84], multi-DNN runtime scheduling is still challenging because most deep learning frameworks adopts one-DNN-per-process execution model by default [68]. DeepEye [5] interleaves the executions of convolutional and fully-connected layers of multiple DNNs for latency hiding. DART [32] is a pipelined multi-DNN scheduling framework under real-time constraints. MASA [71] is the latest memory-aware multi-DNN runtime scheduler. For multi-DNN runtime

scheduling on mobile GPU, ParallelFusion [87] extends kernel fusion to multi-DNN scenarios to maximum the utilisation of mobile GPU. Yu *et al.* [70] propose an automated, fine-grained concurrency control and scheduling framework for multiple DNNs.

Our MTS is complementary. We use a single CUDA stream [83] as the runtime for weight-shared DNNs to avoid the API's inefficient multi-DNN parallelism support [70], [85], [86] and because current mobile GPUs only allows a single stream [87].

3.8 Conclusion

This chapter introduced MTS, a graph rewriting framework for efficient multitask inference with weight-shared DNNs. MTS uses a model stitching scheme to output a single DAG for multiple DNNs with shared weights. The runtime memory usage is minimised by avoiding the duplication of shared weights. With the help of a dedicated model grouping strategy, it also achieves a near-optimal runtime latency. We conducted extensive experiments on different hardware platforms, numbers of tasks, network architectures, pruning ratios, sharing ratios, batch sizes and model heterogeneity. Results show that MTS accelerates up to $6.0\times$ compared to sequentially executing multiple weight-shared DNNs. MTS also yields up to $2.5\times$ lower latency and $3.7\times$ less memory usage compared with `NETFUSE`, a state-of-the-art multi-DNN graph rewriter. We envision our work as a critical step towards full-stack optimisation for efficient multi-DNN execution.

Now combining the MTZ introduced in Chapter 2 and MTS introduced in this chapter, we have a complete solution for the multi-model compression of the first and second types of MMDL system, discussed in Sec. 1.4.1 and Sec. 1.4.2, respectively. In the next chapter, we will investigate the third and also the last type of MMDL system, discussed in Sec. 1.4.3.

4

Pruning-Aware Merging: Multi-Model Compression via Neuron Merging

With the MTZ introduced in Chapter 2 and the MTS introduced in Chapter 3, we have covered the multi-model compression of the first and second type of MMDL system, discussed in Sec. 1.4.1 and Sec. 1.4.2, respectively. Now we investigate the third and also the last type of MMDL system, discussed in Sec. 1.4.3. This type of MMDL system requires a different multi-model compression technique: neuron merging, which should be able to reduce both the number of parameters and the total computation cost. Moreover, there is also the asynchronous execution challenge discussed in Sec. 1.4.3, which requires that the MMDL can be partially deactivated in case some of the tasks are temporarily not required.

This chapter designs a novel neuron merging method for the third type of MMDL system. Many mobile applications demand selective execution of multiple correlated deep learning inference tasks on resource-constrained platforms. Given a set of deep neural networks, each pre-trained for a single task, it is desired that executing arbitrary combinations of tasks yields minimal computation cost. Pruning each network separately yields suboptimal computation costs due to task relatedness. A promising remedy is to merge the networks into a multitask network to eliminate redundancy across tasks before network pruning. However, pruning a multitask network combined with existing network merging schemes cannot minimise the computation cost of every task combination because they do not consider such a future pruning. To this end, we theoretically identify the conditions such that pruning a multitask network minimises the computation of all task combinations. On this basis, we propose Pruning-Aware Merging (PAM), a heuristic network merging scheme to construct a multitask network that approximates these conditions. The merged network is then ready to be further pruned by existing network pruning methods. Evaluations with different pruning schemes, datasets, and network architectures

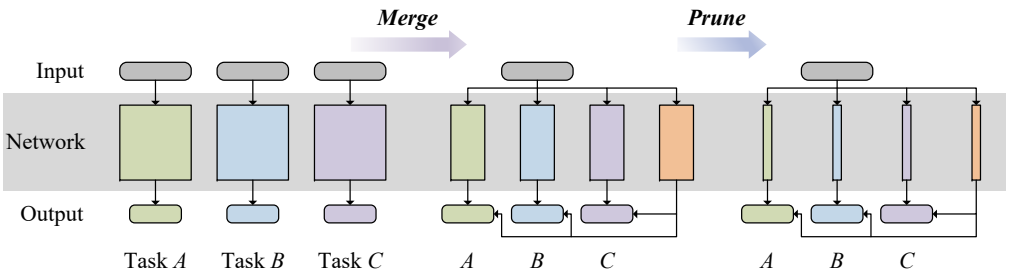


Figure 4.1 Efficient multitask inference by "merge & prune". Three networks pre-trained for tasks A, B and C are first merged into a multitask network and then pruned.

show that PAM achieves up to $4.87\times$ less computation against the baseline without network merging and up to $2.01\times$ less computation against the baseline with a state-of-the-art network merging scheme.

4.1 Introduction

Deep neural networks that can run locally on resource-constrained devices hold potential for various emerging applications such as autonomous drones and social robots [6], [21]. These applications often simultaneously perform a set of correlated inference tasks based on the current context to deliver accurate and adaptive services. Although deep neural networks pre-trained for individual tasks are readily available [2], [24], deploying multiple such networks easily overwhelms the resource budget.

To support these applications on low-resource platforms, we investigate *efficient multitask inference*. Given a set of correlated inference tasks and deep neural networks (each network pre-trained for an individual task), we aim to minimise the computation cost when *any subset of tasks* is performed at inference time.

One naive solution to efficient multitask inference is to *prune* each network for individual tasks *separately*. A deep neural network is typically over-parameterised [97]. Network pruning [8], [10], [45], [98], [99] can radically reduce the number of operations within a network without accuracy loss in the inference task. This solution, however, is only optimal if *a single task* is executed at a time. When multiple correlated tasks are running concurrently, this solution is unable to save computation cost by exploiting tasks relatedness and sharing intermediate results among networks.

A more promising solution framework is *"merge & prune"*, which merges multiple networks into a *multitask network*, before pruning it (Fig. 4.1). A few pioneer studies [20], [29] have explored network merging schemes to eliminate the redundancy among multiple networks pre-trained for correlated tasks. However, pruning a multitask network merged via these schemes can

only minimise computation cost when *all tasks* are executed at the same time.

In this chapter, we propose Pruning-Aware Merging (PAM), a new network merging scheme for efficient multitask inference. By applying existing network pruning methods on the multitask network merged by PAM, the computation cost when performing *any subset* of tasks can be reduced. Extensive experiments show that "PAM & Prune" consistently achieves solid advantages over the state-of-the-art network merging scheme across tasks, datasets, network architectures and pruning methods.

Our main contributions and results are as follows:

- We theoretically show that pruning a multitask network may not simultaneously minimise the computation cost of all task combinations in the network. We then identify conditions such that minimising the computation of all task combinations via network pruning becomes feasible. To the best of our knowledge, this is the first explicit analysis on the applicability of network pruning in multitask networks.
- We propose Pruning-Aware Merging (PAM), a heuristic network merging scheme to construct a multitask network that approximately meets the conditions in our analysis and enables "merge & prune" for efficient multitask inference.
- We evaluate PAM with various pruning schemes, datasets and architectures. PAM achieves up to $4.87\times$ less computation cost against the baseline without network merging, and up to $2.01\times$ less computation cost against the baseline with the state-of-the-art network merging scheme [29].

In the rest of this chapter, we review related work in Sec. 4.2, introduce our problem statement in Sec. 4.3, theoretical analysis in Sec. 4.4 and our solution in Sec. 4.5. We present the evaluations of our methods in Sec. 4.6 and finally conclude in Sec. 4.7.

4.2 Related Work

Our work is related to the following categories of research.

Network Pruning. Network pruning reduces the number of operations in a deep neural network without loss in accuracy [8], [99]. Unstructured pruning removes unimportant weights [11], [98], [100]. However, customised hardware [101] is compulsory to exploit such irregular sparse connections for acceleration. Structured pruning enforces sparsity at the granularity of channels/filters/neurons [10], [45], [102], [103]. The resulting sparsity is fit for acceleration on general-purpose processors. Prior pruning proposals implicitly assume a single task in the given network. We identify the challenges to prune a multitask network and propose a network merging scheme such that pruning the merged multitask network minimises computation cost of all task combinations

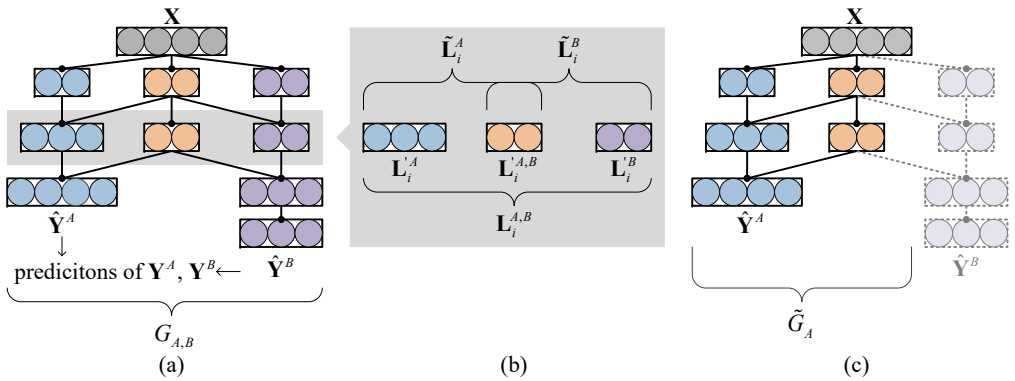


Figure 4.2 Important notations: (a) graph representation $G_{A,B}$ of a multitask network for tasks A and B, with $N_A = 2$ hidden layers for task A and $N_B = 3$ hidden layers for task B; (b) layer outputs for the i-th layer; (c) subgraph $\tilde{G}_A$ for task A.

in the network.

Multitask Networks. A multitask network can be either constructed from scratch via Multi-Task Learning (MTL) or merged from multiple networks pre-trained for individual tasks. MTL joint trains multiple tasks for better generalisation [36], while we focus on the computation cost of running multiple tasks at inference time. Network merging schemes [20], [29] aim to construct a compact multitask network from networks pre-trained for individual tasks. Both MTZ [29] and NeuralMerger [20] enforce weight sharing among networks to reduce their overall storage. In contrast, we account for the computation cost of a multitask network. Although constructing a multitask network using these schemes [20], [29] and pruning it via existing pruning methods can reduce the computation when *all tasks* are concurrently executed, they cannot minimise the computation cost for *every combination of tasks*.

4.3 Problem Statement

We define and analyse our problem based on the graph representation of neural networks. The graph representation reflects the computation cost of neural networks (see below) and facilitates an information theoretical understanding on network pruning (see Sec. 4.4). Fig. 4.2 shows important notations used throughout this chapter. For ease of illustration, we explain our analysis using two tasks. Extensions to more than two tasks are in Sec. 4.5.4.

4.3.1 Graph Representation of Neural Networks

Task. Consider three sets of random variable $\mathbf{X} \in \mathcal{X}$, $\mathbf{Y}^A \in \mathcal{Y}^A$, and $\mathbf{Y}^B \in \mathcal{Y}^B$. Task A outputs $\widehat{\mathbf{Y}}^A$, a prediction of $\mathbf{Y}^A$, by learning the conditional

Algorithm 3: Organise vertices in the graph representation of a neural network into layers.

Input: A neural network graph G^A
Output: $N+1$ layers $\mathbf{v}_i^A$ with $i = 1, \cdots, N+1$.

$\mathbf{v}_0^A \leftarrow \mathbf{v}_X^A$;
$i \leftarrow 0$;
while $N^+(\mathbf{v}_i^A) \neq \mathbf{v}_Y^A$ **do**
  $\mathbf{v}_{i+1}^A \leftarrow \emptyset$;
  for *each node* $v_{i,j}^A \in \mathbf{v}_i^A$ **do**
    if $N^+(v_{i,j}^A) \cap \mathbf{v}_Y^A \neq \emptyset$ **then**
      $\mathbf{v}_{i+1}^A \leftarrow \mathbf{v}_{i+1}^A \cup \{v_{i,j}^A\}$;
    end
  end
  $\mathbf{v}_{i+1}^A \leftarrow \mathbf{v}_{i+1}^A \cup \left(N^+(\mathbf{v}_i^A) \setminus \mathbf{v}_i^A\right)$;
  $i \leftarrow i+1$;
end
$N \leftarrow i$;
$\mathbf{v}_{N+1}^A \leftarrow \mathbf{v}_Y^A$;

distribution $\Pr\{\mathbf{Y}^A = \mathbf{y}|\mathbf{X} = \mathbf{x}\}$. Task B outputs $\widehat{\mathbf{Y}}^B$, a prediction of $\mathbf{Y}^B$, by learning $\Pr\{\mathbf{Y}^B = \mathbf{y}|\mathbf{X} = \mathbf{x}\}$.

Single-Task Network. For task A, a neural network without feedback loops can be represented by an acyclic directed graph $G_A = \{V^A, E^A\}$. Each vertex represents a neuron. There is an edge between two vertices if two neurons are connected. The vertex set V_A can be categorised into three types of nodes: source, internal and sink node. $\deg^-(v)/\deg^+(v)$ is the indegree/outdegree of a vertex v.

- Source node set $\mathbf{v}_X^A = \{v|v \in V^A \wedge \deg^-(v) = 0\}$ represents the *input layer*. Each source node represents an *input neuron* and outputs a random variable $X_i \in \mathbf{X}$. The output of the input layer is the input random variable set $\mathbf{X}$.
- Internal nodes $v_i \in \{v|v \in V \wedge \ \deg^-(v) \neq 0 \wedge \deg^+(v) \neq 0\}$ represents the *hidden neurons*. The output of each hidden neuron is generated by calculating the weighted sum of its inputs and then applying an activation function.
- Sink node set $\mathbf{v}_Y^A = \{v|v \in V \wedge \deg^+(v) = 0\}$ represents the *output layer*. Each sink node represents an *output neuron* and the output is calculated in the same way as the hidden neurons. The output of the output layer is the prediction $\widehat{\mathbf{Y}}^A$ of ground-truth labels $\mathbf{Y}^A$.

We organise the hidden neurons v_i of G^A into layers $\mathbf{v}_i^A$ by Algorithm 3. $N^+(\mathbf{v})$ represents the out-coming neighbours of the vertex set $\mathbf{v}$. Algorithm 3 can organise any acyclic single-task network into layers and the layer outputs satisfy the Markov property.

Multitask Network. For task A and B, a multitask network without feedback loops can be represented by an acyclic directed graph $G_{A,B}$. All paths from the input neurons to the output neurons for task A form a subgraph $\widetilde{G}_A$ (see Fig. 4.2(c)), which is in effect the same as a single-task network. When only

task A is performed, only $\widetilde{G}_A$ is activated. Subgraph $\widetilde{G}_B$ is defined similarly. We also organise vertices of $\widetilde{G}_A$ and $\widetilde{G}_B$ into layers with Algorithm 3. Layer outputs of $\widetilde{G}_A$ and $\widetilde{G}_B$ are denoted as $\widetilde{\mathbf{L}}_i^A$ and $\widetilde{\mathbf{L}}_i^B$. Suppose $\widetilde{G}_A$ and $\widetilde{G}_B$ have respectively N_A and N_B hidden layers. We assume $N_A \leq N_B$ w.l.o.g.. Then the i-th layer output of $G_{A,B}$ is defined as $\mathbf{L}_i^{A,B} = \widetilde{\mathbf{L}}_i^A \cup \widetilde{\mathbf{L}}_i^B$ with $i = 0, \cdots, N_A$. As shown in Fig. 4.2(b), $\mathbf{L}_i^{A,B}$ consists of three sets of neurons: $\mathbf{L}_i'^A$, $\mathbf{L}_i'^B$ and $\mathbf{L}_i'^{A,B}$.

Remarks. The above definitions have two benefits. *(i)* The computation cost of a neural network is an *increasing function* of the size of the graph, *i.e.*, the number of edges plus vertices. Reducing the computation cost of the network is transformed into removing edges or vertices in the graph. *(ii)* For a single-task network with N_A hidden layers, its layer outputs form a Markov chain: $\mathbf{Y}^A \rightarrow \mathbf{L}_0^A \rightarrow \cdots \rightarrow \mathbf{L}_{N_A+1}^A$. All layer outputs $\mathbf{L}_i^{A,B}$ in a multitask network also form a Markov chain. The Markov property allows an information theoretical analysis on neural networks [104], [105].

4.3.2 Problem Definition

Given two single-task networks G_A and G_B pre-trained for task A and B, we aim to construct a multitask network $G_{A,B}$ such that pruning on $G_{A,B}$ can minimise the number of vertices and edges in $G_{A,B}$, $\widetilde{G}_A$ and $\widetilde{G}_B$ while preserving inference accuracy on A and B. To ensure minimal computation of *any subset* of tasks, we need to minimise the number of vertices and edges in *any subgraph*. For two tasks, $G_{A,B}$ corresponds to running task A and B concurrently; $\widetilde{G}_A$ ($\widetilde{G}_B$) corresponds to running task A (B) only. Next, we show the difficulty to optimise all subgraphs simultaneously.

4.4 Theoretical Understanding

This section presents a theoretical understanding on the challenges to prune a multitask network and identifies conditions such that minimising the computation cost of all task combinations via pruning becomes feasible (Theorem 4.1).

4.4.1 Why Pruning a Single-task Network Work

Pruning a single-task network reduces the computation cost of a neural network while retaining task inference accuracy by suppressing *redundancy* in the network [8], [99]. From the information theoretical perspective [104], [105], since the layer outputs form a Markov chain, the inference accuracy for a given task A is positively correlated to the task related information transmitted through the network at each layer, measured by $I(\mathbf{L}_i^A; \mathbf{Y}^A)$. All other information is irrelevant for the task. Hence *the redundancy within a*

single-task network can be defined as below.

Definition 4.1. For the i-th layer in the single-task neural network G_A, the redundancy of the layer is defined as $\mathcal{R}_A(\mathbf{L}_i^A) = \sum_{L_{i,j}^A \in \mathbf{L}_i^A} H(L_{i,j}^A) - I(\mathbf{L}_i^A; \mathbf{Y}^A)$.

$\sum_{L_{i,j}^A \in \mathbf{L}_i^A} H(L_{i,j}^A)$ measures *the maximal amount of information* the layer can express. $I(\mathbf{L}_i^A; \mathbf{Y}^A)$ measures *the amount of task A related information* in the layer output. By definition, $\mathcal{R}_A(\mathbf{L}_i^A) \geq 0$.

Remarks. $\sum_{L_{i,j}^A \in \mathbf{L}_i^A} H(L_{i,j}^A)$ is positively correlated to the number of vertices and incoming edges of the i-th layer. Therefore, in a well trained network where $I(\mathbf{L}_i^A; \mathbf{Y}^A)$ can no longer increase, the computation cost can be minimised by reducing $\mathcal{R}_A(\mathbf{L}_i^A)$.

Accordingly, pruning a single-task network can be formalised as an optimisation problem

$$\text{minimise} \sum_{i=1}^{N_A+1} \left(\mathcal{R}_A(\mathbf{L}_i^A) - \xi_i \cdot I(\mathbf{L}_i^A; \mathbf{Y}^A) \right) \tag{4.1}$$

where $\xi_i > 0$ controls the trade-off between inference accuracy and computation cost.

Remarks. Existing pruning methods implicitly assume a single-task network. That is, they are all designed to solve optimisation problem (4.1), even though the concrete strategies vary. We now show the problems that occur when these pruning methods are applied to a multitask network.

4.4.2 Why Pruning a Multitask Network Fail

As mentioned in Sec. 4.3.2, we aim to minimise the computation cost of any subset of tasks, which is a *multi-objective* optimisation problem. As we will show below, existing network pruning methods are unable to handle these objectives simultaneously.

We first define redundancy when performing two tasks at the same time, similarly as in Definition 4.1.

Definition 4.2. For a multitask network $G_{A,B}$, the redundancy of its i-th layer is $\mathcal{R}_{A,B}(\mathbf{L}_i^{A,B}) = \sum_{L_{i,j}^{A,B} \in \mathbf{L}_i^{A,B}} H(L_{i,j}^{A,B}) - I(\mathbf{L}_i^{A,B}; \mathbf{Y}^A, \mathbf{Y}^B)$.

Following the above definitions of redundancy, our objective in Sec. 4.3.2 is equivalent to minimising the redundancy in $G_{A,B}$ as well as in its two subgraphs $\widetilde{G}_A$ and $\widetilde{G}_B$, which leads to the following three-objective optimisation (still, we

assume $N_A \leq N_B$ w.l.o.g.):

$$\begin{aligned} &\text{minimise} \\ &\sum_{i=1}^{N_A+1} \left(\mathcal{R}_A(\widetilde{\mathbf{L}}_i^A) - \tilde{\xi}_i^A \cdot I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A)\right), \\ &\sum_{i=1}^{N_B+1} \left(\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B) - \tilde{\xi}_i^B \cdot I(\widetilde{\mathbf{L}}_i^B; \mathbf{Y}^B)\right), \\ &\sum_{i=1}^{N_A} \left(\mathcal{R}_{A,B}(\mathbf{L}_i^{A,B}) - \xi_i^A \cdot I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A) - \xi_i^B \cdot I(\widetilde{\mathbf{L}}_i^B; \mathbf{Y}^B)\right) \end{aligned} \tag{4.2}$$

Reducing $\mathcal{R}_A(\widetilde{\mathbf{L}}_i^A)$, $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ and $\mathcal{R}_{A,B}(\mathbf{L}_i^{A,B})$ decreases the number of vertices and edges in $\widetilde{G}_A$, $\widetilde{G}_B$ and $G_{A,B}$, respectively. $\xi_i^A, \xi_i^B, \tilde{\xi}_i^A, \tilde{\xi}_i^B > 0$ are parameters to control the trade-off between computation cost and inference accuracy, as well as to balance task A and B.

To solve optimisation problem (4.2) with prior network pruning methods, we observe two problems.

Problem 1: The first two objectives in (4.2) may conflict. This is because reducing $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ may decrease $I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A)$ (see Lemma 4.1 below). In other words, when pruning subgraph $\widetilde{G}_B$, it is possible that some information related to task A is removed from the shared vertices between $\widetilde{G}_A$ and $\widetilde{G}_B$. Hence $I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A)$ decreases and the inference accuracy of task A deteriorates.

Problem 2: It is unclear how to minimise the third objective in (4.2). As mentioned in Sec. 4.4.1, most pruning methods are designed with a single-task network in mind. It is unknown how to apply them to a multitask network $G_{A,B}$ with architecture in Fig. 4.2 (a).

Here we prove the cause for Problem 1:

Lemma 4.1. *Reducing $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ may decrease $I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A)$.*

Proof. We decompose $I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A)$:

$$\begin{aligned} &I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A) = I(\mathbf{L}_i'^A; \mathbf{Y}^A) + \\ &I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) + I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A; \mathbf{Y}^B | \mathbf{L}_i'^A) \end{aligned} \tag{4.3}$$

where $I(A; B; C) = I(A; B) - I(A; B|C)$ is the *co-information* [106]. From Definition 4.1, we have:

$$\begin{aligned} \mathcal{R}_B(\widetilde{\mathbf{L}}_i^B) &= \sum_{\widetilde{L}_{i,j}^B \in \widetilde{\mathbf{L}}_i^B} H(\widetilde{L}_{i,j}^B) - I(\widetilde{\mathbf{L}}_i^B; \mathbf{Y}^B) \\ &= \sum_{\widetilde{L}_{i,j}^B \in \widetilde{\mathbf{L}}_i^B} H(\widetilde{L}_{i,j}^B) - H(\widetilde{\mathbf{L}}_i^B) + H(\widetilde{\mathbf{L}}_i^B | \mathbf{Y}^B) \end{aligned} \tag{4.4}$$

For the last term, we have:

$$H(\widetilde{\mathbf{L}}_i^B | \mathbf{Y}^B)$$

$$
\begin{aligned}
&= H(\mathbf{L}_i'^B, \mathbf{L}_i'^{A,B} | \mathbf{Y}^B) && (4.5)\\
&= H(\mathbf{L}_i'^{A,B} | \mathbf{Y}^B) + H(\mathbf{L}_i'^B | \mathbf{L}_i'^{A,B}, \mathbf{Y}^B) && (4.6)\\
&= I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{Y}^B) + H(\mathbf{L}_i'^{A,B} | \mathbf{Y}^A, \mathbf{Y}^B) + H(\mathbf{L}_i'^B | \mathbf{L}_i'^{A,B}, \mathbf{Y}^B) && (4.7)\\
&= I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) + I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A; \mathbf{L}_i'^A | \mathbf{Y}^B) + \\
&\quad H(\mathbf{L}_i'^{A,B} | \mathbf{Y}^A, \mathbf{Y}^B) + H(\mathbf{L}_i'^B | \mathbf{L}_i'^{A,B}, \mathbf{Y}^B) && (4.8)
\end{aligned}
$$

Hence, $H(\widetilde{\mathbf{L}}_i^B | \mathbf{Y}^B)$ includes $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B)$. Reducing $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ may decrease $I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A)$. ■

4.4.3 When Pruning a Multitask Network Work

The two problems in Sec. 4.4.2 show that not all multitask networks can be pruned for efficient multitask inference. However, a multitask network can be effectively pruned if it meets the conditions stated by the following theorem.

Theorem 4.1. *If* $\forall\, 1 \leq i \leq N_A$*, the conditions below are satisfied:*

$$
\begin{aligned}
I(\mathbf{L}_i'^A; \mathbf{L}_i'^B; \mathbf{Y}^A; \mathbf{Y}^B) &= 0\\
I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) &= 0\\
I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^B | \mathbf{L}_i'^B, \mathbf{Y}^A) &= 0
\end{aligned}
\tag{4.9}
$$

where $I(\mathbf{L}_i'^A; \mathbf{L}_i'^B; \mathbf{Y}^A; \mathbf{Y}^B)$ *is the co-information [106], then the three-objective optimisation problem* (4.2) *can be reduced to two non-conflicting optimisation problems that can be solved independently:*

$$
\begin{aligned}
&\textit{minimise} \sum_{i=1}^{N_A+1} \mathcal{R}_A(\widetilde{\mathbf{L}}_i^A) - \tilde{\xi}_i^A \cdot I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A),\\
&\textit{minimise} \sum_{i=1}^{N_B+1} \mathcal{R}_B(\widetilde{\mathbf{L}}_i^B) - \tilde{\xi}_i^B \cdot I(\widetilde{\mathbf{L}}_i^B; \mathbf{Y}^B)
\end{aligned}
\tag{4.10}
$$

Proof. Here we shows that the conditions in Theorem 4.1 solve Problem 1 and Problem 2 in Sec. 4.4.2.

Solving Problem 1 in Sec. 4.4.2.

From (4.3) we have the following if $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) = 0$:

$$
I(\widetilde{\mathbf{L}}_i^A; \mathbf{Y}^A) = I(\mathbf{L}_i'^A; \mathbf{Y}^A) + I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A; \mathbf{Y}^B | \mathbf{L}_i'^A) \tag{4.11}
$$

$\mathbf{L}_i'^A$ is not in $\widetilde{\mathbf{L}}_i^B$. Hence $I(\mathbf{L}_i'^A; \mathbf{Y}^A)$ is unaffected when $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ is reduced. $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A; \mathbf{Y}^B | \mathbf{L}_i'^A)$ is included in $I(\widetilde{\mathbf{L}}_i^B; \mathbf{Y}^B)$. Thus minimising $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B) - \tilde{\xi}_i^B \cdot I(\widetilde{\mathbf{L}}_i^B; \mathbf{Y}^B)$ will not reduce $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A; \mathbf{Y}^B | \mathbf{L}_i'^A)$ with a proper $\tilde{\xi}_i^B$. All still hold if we swap A and B in the above equations. Consequently, if $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) = I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^B | \mathbf{L}_i'^B, \mathbf{Y}^A) = 0$, the first two objectives in optimisation problem (4.2) become non-conflicting.

Solving Problem 2 in Sec. 4.4.2.

We first decompose $\mathcal{R}_{A,B}(\mathbf{L}_i^{A,B})$:

$$\begin{aligned}
&\mathcal{R}_{A,B}(\mathbf{L}_i^{A,B}) \\
=&\sum_{L_{i,j}^{A,B}\in\mathbf{L}_i^{A,B}} H(L_{i,j}^{A,B}) - H(\widetilde{\mathbf{L}}_i^A,\widetilde{\mathbf{L}}_i^B) + H(\widetilde{\mathbf{L}}_i^A,\widetilde{\mathbf{L}}_i^B|\mathbf{Y}^A,\mathbf{Y}^B) && (4.12)\\
=&\sum_{L_{i,j}^{A}\in\widetilde{\mathbf{L}}_i^A} H(L_{i,j}^{A}) - I(\widetilde{\mathbf{L}}_i^A;\mathbf{Y}^A,\mathbf{Y}^B) + \sum_{L_{i,j}^{B}\in\widetilde{\mathbf{L}}_i^B} H(L_{i,j}^{B}) - I(\widetilde{\mathbf{L}}_i^B;\mathbf{Y}^A,\mathbf{Y}^B) \\
&+I(\widetilde{\mathbf{L}}_i^A;\widetilde{\mathbf{L}}_i^B;\mathbf{Y}^A,\mathbf{Y}^B) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.13)\\
=&\sum_{L_{i,j}^{A}\in\widetilde{\mathbf{L}}_i^A} H(L_{i,j}^{A}) - I(\widetilde{\mathbf{L}}_i^A;\mathbf{Y}^A) - I(\widetilde{\mathbf{L}}_i^A;\mathbf{Y}^B|\mathbf{Y}^A) + \sum_{L_{i,j}^{B}\in\widetilde{\mathbf{L}}_i^B} H(L_{i,j}^{B}) - I(\widetilde{\mathbf{L}}_i^B;\mathbf{Y}^B) \\
&-I(\widetilde{\mathbf{L}}_i^B;\mathbf{Y}^A|\mathbf{Y}^B) + I(\widetilde{\mathbf{L}}_i^A;\widetilde{\mathbf{L}}_i^B;\mathbf{Y}^A,\mathbf{Y}^B) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.14)\\
=&\mathcal{R}_A(\widetilde{\mathbf{L}}_i^A) + \mathcal{R}_B(\widetilde{\mathbf{L}}_i^B) - I(\widetilde{\mathbf{L}}_i^A;\mathbf{Y}^B|\mathbf{Y}^A) \\
&-I(\widetilde{\mathbf{L}}_i^B;\mathbf{Y}^A|\mathbf{Y}^B) + I(\widetilde{\mathbf{L}}_i^A;\widetilde{\mathbf{L}}_i^B;\{\mathbf{Y}^A,\mathbf{Y}^B\}) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.15)
\end{aligned}$$

Then from (4.15), we have:

$$\begin{aligned}
&\mathcal{R}_{A,B}(\mathbf{L}^{A,B}) - \left(\mathcal{R}_A(\widetilde{\mathbf{L}}_i^A) + \mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)\right) \\
\leq& I(\widetilde{\mathbf{L}}_i^A;\widetilde{\mathbf{L}}_i^B;\{\mathbf{Y}^A,\mathbf{Y}^B\}) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.16)\\
\leq& I(\widetilde{\mathbf{L}}_i^A;\widetilde{\mathbf{L}}_i^B) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.17)\\
=& I(\mathbf{L}_i'^A,\mathbf{L}_i'^{A,B};\mathbf{L}_i'^B,\mathbf{L}_i'^{A,B}) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.18)\\
\leq& I(\mathbf{L}_i'^A;\mathbf{L}_i'^B) + H(\mathbf{L}_i'^{A,B}) - \sum_{L_{i,j}\in\mathbf{L}_i'^{A,B}} H(L_{i,j}) && (4.19)\\
\leq& I(\mathbf{L}_i'^A;\mathbf{L}_i'^B) && (4.20)
\end{aligned}$$

Further,

$$\begin{aligned}
&I(\mathbf{L}_i'^A;\mathbf{L}_i'^B) \\
=&I(\mathbf{L}_i'^A;\mathbf{L}_i'^B;\mathbf{Y}^A;\mathbf{Y}^B) + I(\mathbf{L}_i'^A;\mathbf{L}_i'^B;\mathbf{Y}^A|\mathbf{Y}^B) + I(\mathbf{L}_i'^A;\mathbf{L}_i'^B|\mathbf{Y}^A) && (4.21)\\
\leq& I(\mathbf{L}_i'^A;\mathbf{L}_i'^B;\mathbf{Y}^A;\mathbf{Y}^B) + H(\mathbf{L}_i'^A|\mathbf{Y}^A) + H(\mathbf{L}_i'^B|\mathbf{Y}^B) && (4.22)\\
\leq& I(\mathbf{L}_i'^A;\mathbf{L}_i'^B;\mathbf{Y}^A;\mathbf{Y}^B) + \mathcal{R}_A(\widetilde{\mathbf{L}}_i^A) + \mathcal{R}_B(\widetilde{\mathbf{L}}_i^B) && (4.23)
\end{aligned}$$

This is a loose upper bound. However, since $\mathcal{R}_{A,B}(\mathbf{L}^{A,B})$, $\mathcal{R}_A(\widetilde{\mathbf{L}}_i^A)$ and $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ are lower bounded by 0, it suffices to show that when $I(\mathbf{L}_i'^A;\mathbf{L}_i'^B;\mathbf{Y}^A;\mathbf{Y}^B)=0$, minimising $\mathcal{R}_A(\widetilde{\mathbf{L}}_i^A)$ and $\mathcal{R}_B(\widetilde{\mathbf{L}}_i^B)$ will minimise $\mathcal{R}_{A,B}(\mathbf{L}^{A,B})$.

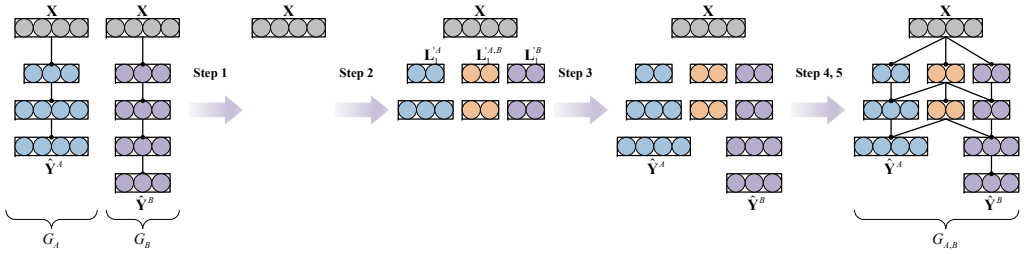


Figure 4.3 PAM workflow to construct a multitask network ($G_{A,B}$) from two single-task networks (G_A and G_B).

In summary, when

$$\begin{aligned} I(\mathbf{L}_i'^A; \mathbf{L}_i'^B; \mathbf{Y}^A; \mathbf{Y}^B) &= 0 \\ I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) &= 0 \\ I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^B | \mathbf{L}_i'^B, \mathbf{Y}^A) &= 0 \end{aligned} \tag{4.24}$$

the optimisation problem (4.2) is reduced to two non-conflicting optimisation problems (4.10). ■

Remarks. Each of the two optimisation problems (4.10) are in effect optimisation problem similar to single-task pruning problem (4.1), which can be effectively solved by prior pruning proposals. Theorem 4.1 provides important guidelines to design the network merging scheme for our problem in Sec. 4.3.2. Specifically, if G_A and G_B can be merged into a a multitask network $G_{A,B}$ such that conditions (4.9) are satisfied, we can simply apply existing network pruning on the two subgraphs $\widetilde{G}_A$ and $\widetilde{G}_B$ to minimise the computation cost when performing any subset of tasks.

4.5 Pruning-Aware Merging

Based on the above analysis, we propose Pruning-Aware Merging (PAM), a novel network merging scheme that constructs a multitask network from pre-trained single task networks. PAM approximately meets the conditions in Theorem 4.1 such that the merged multitask network can be effectively pruned for efficient multitask inference.

4.5.1 PAM Workflow

Given two single-task networks G_A and G_B pre-trained for task A and B ($N_A \leq N_B$), PAM constructs a multitask network $G_{A,B}$ with the steps below (see Fig. 4.3).

1. Assign $\mathbf{L}_0^{A,B} = \mathbf{X}$, as $G_{A,B}$, G_A and G_B use the same inputs.

2. For $i = 1, \cdots, N_A$, regroup the neurons from $\mathbf{L}_i^A$ and $\mathbf{L}_i^B$ into $\mathbf{L}_i'^A$, $\mathbf{L}_i'^B$ and $\mathbf{L}_i'^{A,B}$ by the regrouping algorithm in Sec. 4.5.2.

3. Take over the output layer for task A: $\widetilde{\mathbf{L}}_{N_A+1}^A = \mathbf{L}_{N_A+1}^A$. For $i = N_A + 1, \cdots, N_B + 1$, take over the remaining layers from G_B: $\widetilde{\mathbf{L}}_i^B = \mathbf{L}_i^B$.

4. Reconnect the neurons as in Fig. 4.3. If a connection exist before merging, it preserves its original weight. Otherwise it is initialised with a zero.

5. Finetune $G_{A,B}$ on A and B to learn the newly added connections. For the shared connections, $\mathbf{L}_{i-1}'^{A,B} \rightarrow \mathbf{L}_i'^{A,B}$. The gradients are first calculated separately on A and B, and then averaged before weight updating.

Now the multitask network $G_{A,B}$ is ready to be pruned. From Theorem 4.1, we can apply network pruning on the two subgraphs $\widetilde{G}_A$ and $\widetilde{G}_B$ independently and achieve a minimal computation cost for all combinations of tasks. However, since we only approximate the conditions in (4.9), pruning $\widetilde{G}_A$ and $\widetilde{G}_B$ is not perfectly independent in practice. Hence we prune $\widetilde{G}_A$ and $\widetilde{G}_B$ *in an alternating manner* to balance between task A and B.

4.5.2 Regrouping Algorithm

The core of PAM is the regrouping algorithm in the second step in Sec. 4.5.1. It regroups the neurons from $\mathbf{L}_i^A$ and $\mathbf{L}_i^B$ into three sets: $\mathbf{L}_i'^A$, $\mathbf{L}_i'^B$ and $\mathbf{L}_i'^{A,B}$, such that the conditions (4.9) in Theorem 4.1 are satisfied. However, it is computation-intensive to estimate the co-information and conditional mutual information in (4.9) precisely. We rely on the following theorem to approximate the conditions.

Theorem 4.2. *The conditions in* (4.9) *can be achieved by minimising* $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$, $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$, *and maximising* $I(\mathbf{L}_i'^A; \mathbf{Y}^A)$, $I(\mathbf{L}_i'^B; \mathbf{Y}^B)$.

Proof. For co-information between four random variables, we have from [106]:

$$0 \leq I(\mathbf{L}_i'^A; \mathbf{L}_i'^B; \mathbf{Y}^A; \mathbf{Y}^B) \leq \min\{I(\mathbf{L}_i'^A; \mathbf{Y}^B), I(\mathbf{L}_i'^B; \mathbf{Y}^A)\} \tag{4.25}$$

Therefore, the first condition in Theorem 4.1, *i.e.*, $I(\mathbf{L}_i'^A; \mathbf{L}_i'^B; \mathbf{Y}^A; \mathbf{Y}^B) = 0$, is achieved by minimising $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ to 0.

Then for the second condition in Theorem 4.1, *i.e.*, $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) = 0$:

$$\begin{aligned}
& I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) \\
\leq & H(\mathbf{Y}^A | \mathbf{L}_i'^A, \mathbf{Y}^B) && (4.26)\\
= & H(\mathbf{Y}^A | \mathbf{Y}^B) - I(\mathbf{Y}^A; \mathbf{L}_i'^A) + I(\mathbf{Y}^A; \mathbf{L}_i'^A; \mathbf{Y}^B) && (4.27)\\
\leq & H(\mathbf{Y}^A | \mathbf{Y}^B) - I(\mathbf{Y}^A; \mathbf{L}_i'^A) + I(\mathbf{L}_i'^A; \mathbf{Y}^B) && (4.28)
\end{aligned}$$

Algorithm 4: Regroup algorithm.

Input: $\mathbf{L}_i^A$, $\mathbf{L}_i^B$, $\mathbf{X}$, $\mathbf{Y}^A$, $\mathbf{Y}^B$, α
Output: $\mathbf{L}_i'^A$, $\mathbf{L}_i'^B$, $\mathbf{L}_i'^{A,B}$

1 $N = \min\{N^A, N^B\}$;
2 **for** $i \leftarrow 1$ **to** N **do**
3 $\quad \mathbf{F}^A \leftarrow \mathbf{F}^B \leftarrow \mathbf{L}_i^A \cup \mathbf{L}_i^B$;
4 $\quad \mathbf{L}_i'^A \leftarrow \emptyset$;
5 $\quad$ **while** $I(\mathbf{L}_i'^A; \mathbf{Y}^B) \leq \alpha$ **do**
6 $\quad\quad L_{i,\cdot} \leftarrow \arg\min_{L_{i,j} \in \mathbf{F}_i^A} I(\{L_{i,j}\} \cup \mathbf{L}_i'^A; \mathbf{Y}^B)$;
7 $\quad\quad$ move the neuron $L_{i,\cdot}$ from $\mathbf{F}^A$ to $\mathbf{L}_i'^A$
8 $\quad$ **end**
9 $\quad \mathbf{L}_i'^B \leftarrow \emptyset$;
10 $\quad$ **while** $I(\mathbf{L}_i'^B; \mathbf{Y}^A) \leq \alpha$ **do**
11 $\quad\quad L_{i,\cdot} \leftarrow \arg\min_{L_{i,j} \in \mathbf{F}_i^B} I(\{L_{i,j}\} \cup \mathbf{L}_i'^B; \mathbf{Y}^A)$;
12 $\quad\quad$ move the neuron $L_{i,\cdot}$ from $\mathbf{F}^B$ to $\mathbf{L}_i'^B$
13 $\quad$ **end**
14 $\quad$ The remaining neurons join $\mathbf{L}_i'^{A,B}$: $\mathbf{L}_i'^{A,B} \leftarrow \mathbf{L}_i^A \cup \mathbf{L}_i^B \setminus \left(\mathbf{L}_i'^A \cup \mathbf{L}_i'^B\right)$;
15 $\quad$ If a neuron exists in both $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$, remove the neuron from them both.
16 **end**

Given A and B, $H(\mathbf{Y}^A|\mathbf{Y}^B)$ is constant. The second condition in Theorem 4.1 is achieved by minimising $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ to 0 and maximising $I(\mathbf{Y}^A; \mathbf{L}_i'^A)$ to $H(\mathbf{Y}^A|\mathbf{Y}^B)$.

The same holds if we swap A and B. The third condition in Theorem 4.1, *i.e.*, $I(\mathbf{L}_i'^{A,B}; \mathbf{Y}^B|\mathbf{L}_i'^B, \mathbf{Y}^A) = 0$, is achieved by minimising $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ and maximising $I(\mathbf{Y}^B; \mathbf{L}_i'^B)$. ■

Remarks. $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ describe the "misplaced" information, i.e., the information that is useful for one task, but contained in neurons that are not connected to the outputs of this task. Therefore such information is redundant and needs to be minimised. $I(\mathbf{L}_i'^A; \mathbf{Y}^A)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^B)$ measure the "relevant" information, i.e., the information useful for one task and contained in neurons connected to this task. Note that this information may not be simply maximised, because it includes the information that is useful for both tasks. It requires simultaneously minimising the "misplaced" information and maximising the "correct" information to achieve the conditions in (4.9).

Based on Theorem 4.2, we propose an algorithm to regroup the neurons such that conditions (4.9) are approximately met. It constructs the largest possible set $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$ from all the neurons in $\mathbf{L}_i^A$ and $\mathbf{L}_i^B$ while $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ remain close to zero, such that $I(\mathbf{L}_i'^A; \mathbf{Y}^A)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^B)$ are approximately maximised. To estimate $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$, we use a Kullback-Leibler-based mutual information upper bound estimator from [107].

Algorithm 4 illustrates the pseudocode to regroup the neurons such that the conditions in Theorem 4.1 are approximated met. Central in Algorithm 4 is a greedy search in Lines 5-8 and 10-13. In Lines 5-8, we search for the largest possible set of neuron $\mathbf{L}_i'^A$ while $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ remains approximately zero (smaller than a pre-defined threshold α), such that $I(\mathbf{L}_i'^A; \mathbf{Y}^A)$ is approximately

maximised. Similarly, in Lines 10-13, we approximately maximise $I(\mathbf{L}_i'^B; \mathbf{Y}^B)$ while keeping $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ close to zero. According to Theorem 4.2, the conditions in Theorem 4.1 are approximately met.

Practical Issue: How to Estimate Mutual Information. We use a Kullback-Leibler-based mutual information upper bound estimator from [107] to estimate the upper bounds of $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$. Since the upper bounds are approximate, it is impossible to request them to be exactly zero. Hence, we use a threshold parameter α to keep $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ close to zero.

Practical Issue: How to Tune Threshold α. The parameter α affects the performance of "PAM & prune". A larger α results in more neurons in $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$ and fewer shared neurons in $\mathbf{L}_i'^{A,B}$. In this case, the multitask network after "PAM & prune" performs worse in terms of efficiency when both tasks are executed concurrently, but better when only one task is executed (similar to "baseline 1 & prune"). Conversely, a smaller α results in more shared neurons. In this case, the multitask network after "PAM & prune" performs worse when only one task is executed, but better when both tasks are executed concurrently, (similar to "baseline 2 & prune").

The parameter α can be empirically tuned as follows:

1. Execute Algorithm 4 with a small α.
2. Increase the value of α slightly and rerun Algorithm 4. Since Lines 5-8 and 10-13 are greedy search, the results for the smaller α in Step 1 (*i.e.*, the already constructed neuron sets $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$) can be reused, instead of starting with empty sets as in Line 4 and 9.
3. Iterate Step 2 till a satisfying balance among task combinations. In each iteration of Step 2, we can reuse the neuron sets $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$ from the last iteration.

The impact of α is shown in Sec. 4.6.3.1.

4.5.3 Extensions to ResNets

In order to support merging Residual Networks [108], PAM needs to be slightly modified. As illustrated in Fig. 4.4, the regrouping of the last layer in each residual block happens not directly after the weighted summation, but after the superposition with the shortcut connection and just before the vector is passed as inputs to the first layer in the next block. This input vector of the first layer in each block is also regrouped using Algorithm 4 and then pruned at a later stage. This special treatment for the last layer in each residual block is consistent with ResNet compatible pruning methods such as [45], which can also prune the block outputs just before it is fed into the first layer in the next block.

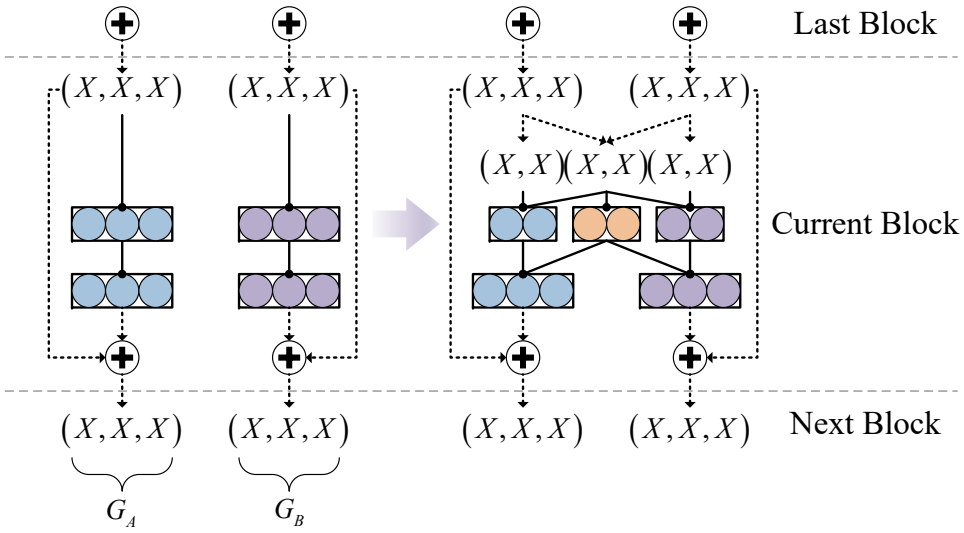


Figure 4.4 Applying PAM on residual blocks. Vectors are denoted as $(X, \cdots, X)$. Dotted lines are identical connections, and firm lines represent weighted connections for neurons.

4.5.4 Extension to Three or More Tasks

When there are $K \geq 3$ tasks, we define the set of all the task as $\upsilon = \{t_1, \cdots, t_K\}$. The merged multitask network can be divided into subgraphs $\widetilde{G}_\tau$, where $\tau \subseteq \upsilon$ and $\tau \neq \emptyset$ is a nonempty subset of tasks. Each vertex in $\widetilde{G}_\tau$ has paths to all the outputs $\widehat{\mathbf{Y}}^t$ with $t \in \tau$. When a task combination (*i.e.*, a subset of tasks) τ is executed, only subgraph $\widetilde{G}_\tau$ is activated. Layers in $\widetilde{G}_\tau$ is denoted as $\widetilde{\mathbf{L}}_i^\tau$. The output layer for task combination τ is denoted as $\widehat{\mathbf{Y}}^\tau = \bigcup_{t\in\tau} \widehat{\mathbf{Y}}^t$, which is the prediction of ground-truth labels $\mathbf{Y}^\tau = \bigcup_{t\in\tau} \mathbf{Y}^t$.

Extension of Theorem 4.1. For any pair of non-overlapped nonempty subsets of task τ_A and τ_B ($\tau_A \cap \tau_B = \emptyset$), define:

$$\mathbf{A}_i = \widetilde{\mathbf{L}}_i^{\tau_A} \setminus \widetilde{\mathbf{L}}_i^{\tau_B} \tag{4.29}$$

$$\mathbf{B}_i = \widetilde{\mathbf{L}}_i^{\tau_B} \setminus \widetilde{\mathbf{L}}_i^{\tau_A} \tag{4.30}$$

$$\mathbf{M}_i = \widetilde{\mathbf{L}}_i^{\tau_A} \cap \widetilde{\mathbf{L}}_i^{\tau_B} \tag{4.31}$$

Then Theorem 4.1 is extended into:

Theorem 4.3. *If for all $i = 1, \cdots, N$ with $N = \min_{t\in\upsilon} N_t$, and for any pair of non-overlapped nonempty subsets of task τ_A and τ_B, the following conditions are satisfied:*

$$\begin{aligned} I(\mathbf{A}_i; \mathbf{B}_i; \mathbf{Y}^{\tau_A}; \mathbf{Y}^{\tau_B}) &= 0 \\ I(\mathbf{M}_i; \mathbf{Y}^{\tau_A} | \mathbf{A}_i, \mathbf{Y}^{\tau_B}) &= 0 \\ I(\mathbf{M}_i; \mathbf{Y}^{\tau_B} | \mathbf{B}_i, \mathbf{Y}^{\tau_A}) &= 0 \end{aligned} \tag{4.32}$$

then the computation cost of executing all task combinations can be minimised by the following K non-conflicting optimisation problems that can be solved

Algorithm 5: Extending Algorithm 4 to over two tasks

Input: $\mathbf{X}$, α, $\mathbf{L}_i^t$, and $\mathbf{Y}^t$ for all $t \in \upsilon$
Output: $\mathbf{L}_i'^{\tau}$ for all $\tau \subseteq \upsilon$ and $\tau \neq \emptyset$

$N \leftarrow \min_{t\in\upsilon} N_t$;
$K \leftarrow |\upsilon|$;
for $i \leftarrow 1$ **to** N **do**
  $\mathbf{S} \leftarrow \bigcup_{t\in\upsilon} \mathbf{L}_i^t$;
  for $n \leftarrow 1$ **to** $K-1$ **do**
    for *any* τ *with* $|\tau| = n$ **do**
      $\mathbf{F} \leftarrow \mathbf{S}$;
      $\mathbf{L}_i'^{\tau} \leftarrow \emptyset$;
      $\mathbf{Y}^{\notin\tau} \leftarrow \bigcup_{t\notin\tau} \mathbf{Y}^t$
      while $I(\mathbf{L}_i'^{\tau}; \mathbf{Y}^{\notin\tau}) \leq \alpha$ **do**
        $L_{i,\cdot} \leftarrow \arg\min_{L_{i,j}\in\mathbf{F}} I(\{L_{i,j}\} \cup \mathbf{L}_i'^{\tau}; \mathbf{Y}^{\notin\tau})$
        move the neuron $L_{i,\cdot}$ from $\mathbf{F}$ to $\mathbf{L}_i'^{\tau}$
      end
    end
    Remove all selected neurons from $\mathbf{S}$: $\mathbf{S} \leftarrow \mathbf{S} \setminus \bigcup_{|\tau|=n} \mathbf{L}_i'^{\tau}$
    Among all $\mathbf{L}_i'^{\tau}$, if a neuron exists in more than one set, remove the neuron from them all
  end
  $\mathbf{L}_i'^{\upsilon} \leftarrow \mathbf{S}$
end

independently:

$$\textit{For every } t \in \upsilon\textit{: minimise } \sum_{i=1}^{N+1} \mathcal{R}_t(\widetilde{\mathbf{L}}_i^t) - \tilde{\xi}_i^t \cdot I(\widetilde{\mathbf{L}}_i^t; \mathbf{Y}^t) \tag{4.33}$$

Theorem 4.3 can be proven by recursively applying Theorem 4.1.

Extension of PAM. The neuron sets $\mathbf{L}_i'^{A}$, $\mathbf{L}_i'^{B}$ and $\mathbf{L}_i'^{A,B}$ are extended to:

$$\mathbf{L}_i'^{\tau} = \bigcap_{t\in\tau} \widetilde{\mathbf{L}}_i^t \setminus \bigcup_{t\notin\tau} \widetilde{\mathbf{L}}_i^t \tag{4.34}$$

Note that neurons in $\mathbf{L}_i'^{\tau}$ are activated iff any task $t \in \tau$ is executed. Now Algorithm 4 is extended to Algorithm 5. And at step 5 of the PAM workflow in Sec. 4.5.1, we connect $\mathbf{L}_{i-1}'^{\tau_1} \rightarrow \mathbf{L}_i'^{\tau_2}$ iff $\tau_2 \subseteq \tau_1$.

It is worth mentioning that when tasks are highly related, the numbers of neurons in $\mathbf{L}_i^{\tau}$ with $1 < |\tau| < K$ can be extremely small (as in our experiment on the LFW dataset in Sec. 4.6.1). Therefore we can simplify Algorithm 5 by fixing $n = 1$ and skip the remaining loops. Every layer in the multitask network merged by the simplified PAM contains only neuron sets $\mathbf{L}_i^t$ with $t \in \upsilon$ and one shared neuron set $\mathbf{L}_i^{\upsilon}$. Shared neurons in $\mathbf{L}_i^{\upsilon}$ are always activated, while non-shared neurons in $\mathbf{L}_i^t$ are activated iff task t is executed.

4.6 Experiments

We compare different network merging schemes on whether lower computation is achieved when performing *any subset* of tasks.

4.6.1 Experiment Settings

Baselines for Network Merging. We compare PAM with two merging schemes.

- **Baseline 1**. It simply skips network merging in the "merge & prune" framework. Therefore, no multitask network is constructed. As mentioned in Sec. 4.1, this scheme optimises the pruning of *single-task* networks.
- **Baseline 2**. Pre-trained single-task networks are merged as a multitask network by MTZ [29], a state-of-the-art network merging scheme. Applying MTZ in "merge & prune" can minimise the computation cost of a multitask network when *all tasks* are executed.

Methods for Network Pruning. Since we aim to compare different network merging schemes in the "merge & prune" framework, we apply the same network pruning method on the neural network(s) constructed by different merging schemes. To show that PAM works with different pruning methods, we choose two state-of-the-art structured network pruning methods: one [10] uses information theory based metrics (denoted as P1), and the other [45] uses sensitivity based metrics (denoted as P2).

The pruning methods are applied to the neural network(s) constructed by different merging schemes as follows. For Baseline 1, each single-task network is pruned independently. For the multitask network constructed with Baseline 2 and PAM, we prune every subgraph for each individual task in an alternating manner (*e.g.*, task $A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow \cdots$) in order to balance between tasks. However, only P2 is originally designed to prune a ResNet. Hence we only experiment ResNets with P2.

Datasets and Single-Task Networks. We define tasks from three datasets: Fashion-MNIST [109], CelebA [41], and LFW [110]. Fashion-MNIST and CelebA each contains *two* tasks. LFW contains *five* tasks. We use LeNet-5 [24] as pre-trained single-task networks for tasks derived from Fashion-MNIST, and VGG-16 [2] for tasks from CelebA and LFW. We also use ResNet-18 and ResNet-34 [108] as pre-trained single-task networks for CelebA. Table 4.1 summarises the inference accuracy and FLOPs of the pre-trained single-task networks.

The Fashion-MNIST dataset[1] contains 8000 training images and 2000 test images with a resolution of 496×124. Each image has four fashion product images randomly selected from Fashion-MNIST [109]. The 10 categories of fashion products is considered as 10 binary classification problem, and we divide them into two groups (5/5) to form task A and B. On each task we train a LeNet-5, a commonly used architecture for Fashion-MNIST.

The CelebA dataset[2] contains over 200 thousand celebrity face images labelled with 40 attributes. The 40 attributes is divided into two groups $(20/20)$ to form

[1] https://github.com/f-rumblefish/Multi-Label-Fashion-MNIST
[2] http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

Table 4.1 Test accuracy and computation cost of pre-trained single-task networks.

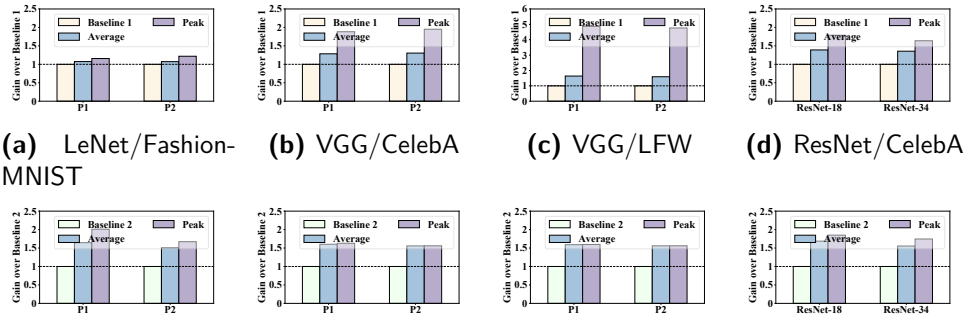
Model/Dataset	Task	Accuracy	FLOPs ($\times 10^6$)
LeNet-5/Fashion-MNIST	A	96.05%	106.42
	B	96.37%	106.42
VGG-16/CelebA	A	90.28%	3112.20
	B	89.03%	3112.20
VGG-16/LFW	A	90.23%	3110.12
	B	84.15%	3110.12
	C	85.03%	3110.12
	D	86.62%	3110.12
	E	87.44%	3110.12
ResNet-18/CelebA	A	90.56%	994.00
	B	88.91%	994.00
ResNet-34/CelebA	A	90.42%	1115.06
	B	88.70%	1115.06

task A and B. The dataset is divided into training and test sets containing 80% and 20% of the samples. The input picture resolution is resized to 72×72. On each task we train slightly modified VGG-16 models, a commonly used single-task network architecture on CelebA. The width of the fully connected layers in VGG-16 is changed to 512. The convolutional layers are initialised with weights pre-trained for imdb-wiki [25], and use the same pre-processing steps.

The Labelled Faces in the Wild (LFW) dataset[3] contains over 13,000 face photographs collected from the web. Each face photo is associated with 73 attributes [111]. We randomly split the 73 labels in the LFW dataset into four groups with 15 labels each and one group with 13 labels. Each group of labels forms a single task. The dataset is divided into training and test sets containing 80% and 20% of the samples. Same as in CelebA, the input picture resolution is resized to 72×72. On each task we train slightly modified VGG-16 models, a commonly used single-task network architecture on LFW. The width of the fully connected layers in VGG-16 is changed to 128. The convolutional layers are initialised with weights pre-trained for imdb-wiki [25], and use the same pre-processing steps.

Evaluation Metrics. For a given set of tasks, we aim to minimise the computation cost of all task combinations. To assess computation cost independent of hardware, we use the number of floating point operations (FLOP) as the metric. For fair comparison, the network(s) constructed by different merging schemes are pruned while preserving almost the same inference accuracy. To quantify the performance advantage of PAM *over baselines* over all task combinations, we adopt the following two single-valued criteria:

[3] http://vis-www.cs.umass.edu/lfw/



(a) LeNet/Fashion-MNIST **(b)** VGG/CelebA **(c)** VGG/LFW **(d)** ResNet/CelebA

(e) LeNet/Fashion-MNIST **(f)** VGG/CelebA **(g)** VGG/LFW **(h)** ResNet/CelebA

Figure 4.5 Average and peak gain of PAM over baselines in different combinations of models, datasets, and pruning methods. The upper row (a)-(d) shows the gain of PAM over baseline 1. The lower row (e)-(h) shows the gain of PAM over baseline 2. Note that the average and peak gain of each baseline is 1 by definition.

■ **Average Gain**. This metric measures the averaged computation cost reduction of "PAM & prune" over "baseline & prune" across *all task combinations*. For example, given two tasks A and B, there are three task combinations: A, B and $A\&B$. When executing these task combinations, the FLOPs of the network after "PAM & prune" are c_A^P, c_B^P and $c_{A,B}^P$, respectively. After "baseline 1 & prune", the FLOPs are c_A^{B1}, c_B^{B1} and $c_{A,B}^{B1}$, respectively. The average gain over baseline 1 is calculated as $\frac{1}{3}(c_A^{B1}/c_A^P + c_B^{B1}/c_B^P + c_{A,B}^{B1}/c_{A,B}^P)$.

■ **Peak Gain**. This metric measures the maximal computation cost reduction across *all task combinations*. Using the same example and notations as above, the peak gain over baseline 1 is calculated as $\max\{c_A^{B1}/c_A^P, c_B^{B1}/c_B^P, c_{A,B}^{B1}/c_{A,B}^P\}$.

All experiments are implemented with TensorFlow and conducted on a workstation with Nvidia RTX 2080 Ti GPU.

4.6.2 Main Experiment Results

Overall Performance Gain. Fig. 4.5 shows the average and peak gains of PAM over the two baselines tested with different models (LeNet-5, VGG-16, ResNet-18, RestNet-34), datasets (Fashion-MNIST, CelebA, LFW), and pruning methods (P1, P2). The detailed FLOPs and inference accuracy on task merging (Fashion-MNIST and CelebA) are listed in Table 4.2, Table 4.3, Table 4.4 and Table 4.5.

Compared with baseline 1, PAM achieves $1.07\times$ to $1.64\times$ average gain and $1.16\times$ to $4.87\times$ peak gain. Compared with baseline 2, PAM achieves $1.51\times$ to $1.69\times$ average gain and $1.56\times$ to $2.01\times$ peak gain. In general, PAM has significant performance advantage over both baselines across datasets and network architectures.

Effectiveness of PAM. From Fig. 4.5, the performance gain of PAM varies

Table 4.2 Test accuracy and computation cost of all tasks combinations with LeNet-5 on Fashion-MNIST pruned by P1/P2.

Pruning	Tasks	Accuracy			FLOPs ($\times 10^6$)		
		B1	B2	PAM	B1	B2	PAM
P1	A	95.42%	95.30%	94.67%	28.34	52.58	28.49
	B	96.30%	96.40%	95.70%	28.34	52.58	26.16
	A&B	95.86%	95.85%	95.19%	56.69	52.58	48.68
P2	A	95.82%	95.73%	95.70%	18.64	31.19	18.65
	B	96.46%	96.72%	96.38%	18.64	31.19	18.65
	A&B	96.14%	96.22%	96.04%	37.27	31.19	26.48

Table 4.3 Test accuracy and computation cost of all tasks combinations with VGG-16 on CelebA pruned by P1/P2.

Pruning	Tasks	Accuracy			FLOPs ($\times 10^6$)		
		B1	B2	PAM	B1	B2	PAM
P1	A	89.45%	89.09%	89.60%	4.52	7.3	4.48
	B	87.81%	87.69%	88.00%	4.32	7.3	4.49
	A&B	88.63%	88.39%	88.80%	8.85	7.3	4.70
P2	A	90.34%	90.27%	90.36%	153.13	243.20	155.82
	B	88.84%	88.74%	88.76%	152.65	243.20	155.84
	A&B	89.59%	89.51%	89.56%	305.78	243.20	156.74

across baselines and datasets. Such variations in average and peak gains are influenced by *how many neurons are shared* and *how many networks are merged*. Fig. 4.6 shows how many neurons (kernels) are shared after "PAM & prune" on LeNet-5 and VGG-16.

- **The more neurons shared, the higher gain PAM has over baseline 1.** "Baseline 1 & prune" can effectively reduce the computation cost when *only one* task is performed. However, when many neurons can be shared (see Fig. 4.6(b), (c), (e), and (f)), baseline 1 is sub-optimal when multiple tasks are executed simultaneously, as it is unable to reduce computation by sharing neurons. This is why PAM outperforms baseline 1 more on CelebA and LFW.

- **The fewer neurons shared, the higher gain PAM has over baseline 2.** "Baseline 2 & prune" can effectively reduce the computation cost via neuron sharing when *all* tasks are performed simultaneously. However, when only few neurons can be shared (see Fig. 4.6(a) and (d)), the multitask network merged by baseline 2 cannot shut down the unnecessary neurons when not all tasks are executed, and hence yields sub-optimal computation cost. This is why PAM outperforms baseline 2 more on Fashion-MNIST.

- **The more networks merged, the higher gain PAM has over both baselines.** As the number of single-task networks (tasks) increases, "PAM & prune" can either share more neurons and yield lower computation than "baseline 1 & prune", or shut down more unnecessary neurons and yield lower computation than "baseline 2 & prune". Therefore the performance gain of PAM over baseline 1 on LFW is such significantly higher than on CelebA. This

Table 4.4 Test accuracy and computation cost of all tasks combinations with VGG-16 on LFW pruned by P1/P2.

Pruning	Tasks	Accuracy			FLOPs ($\times 10^6$)		
		B1	B2	PAM	B1	B2	PAM
P1	A	89.77%	89.49%	89.87%	7.96	12.66	7.94
	B	82.81%	82.82%	82.14%	7.91	12.66	7.95
	C	83.20%	82.68%	83.30%	7.94	12.66	7.94
	D	85.74%	86.45%	86.03%	7.58	12.66	7.93
	E	87.10%	86.52%	86.90%	7.87	12.66	7.93
	A&B	86.29%	86.16%	86.00%	15.87	12.66	7.98
	A&C	86.48%	86.09%	86.59%	15.90	12.66	7.97
	A&D	87.75%	87.97%	87.95%	15.54	12.66	7.97
	A&E	88.44%	88.01%	88.39%	15.84	12.66	7.96
	B&C	83.00%	82.75%	82.72%	15.85	12.66	7.98
	B&D	84.28%	84.64%	84.09%	15.49	12.66	7.97
	B&E	84.95%	84.67%	84.52%	15.79	12.66	7.97
	C&D	84.47%	84.57%	84.66%	15.52	12.66	7.96
	C&E	85.15%	84.60%	85.10%	15.81	12.66	7.96
	D&E	86.42%	86.49%	86.47%	15.45	12.66	7.96
	A&B&C	85.26%	85.00%	85.10%	23.81	12.66	8.01
	A&B&D	86.11%	86.25%	86.01%	23.45	12.66	8.01
	A&B&E	86.56%	86.28%	86.30%	23.75	12.66	8.00
	A&C&D	86.24%	86.21%	86.40%	23.48	12.66	8.00
	A&C&E	86.69%	86.23%	86.69%	23.78	12.66	7.99
	A&D&E	87.54%	87.49%	87.60%	23.42	12.66	7.99
	B&C&D	83.92%	83.98%	83.82%	23.43	12.66	8.01
	B&C&E	84.37%	84.01%	84.11%	23.73	12.66	8.00
	B&D&E	85.22%	85.26%	85.02%	23.37	12.66	8.00
	C&D&E	85.35%	85.22%	85.41%	23.39	12.66	7.99
	A&B&C&D	85.38%	85.36%	85.34%	31.39	12.66	8.04
	A&B&C&E	85.72%	85.38%	85.55%	31.69	12.66	8.03
	A&B&D&E	86.35%	86.32%	86.23%	31.33	12.66	8.03
	A&C&D&E	86.45%	86.29%	86.53%	31.36	12.66	8.02
	B&C&D&E	84.71%	84.62%	84.59%	31.31	12.66	8.03
	A&B&C&D&E	85.72%	85.59%	85.65%	39.27	12.66	8.06
P2	A	89.57%	89.38%	89.24%	22.91	36.33	23.28
	B	81.96%	83.15%	83.39%	23.16	36.33	23.29
	C	82.96%	81.61%	82.10%	22.93	36.33	23.28
	D	85.04%	85.12%	85.29%	21.16	36.33	23.27
	E	86.43%	85.81%	85.57%	21.29	36.33	23.27
	A&B	85.76%	86.27%	86.31%	46.07	36.33	23.32
	A&C	86.26%	85.50%	85.67%	45.84	36.33	23.31
	A&D	87.31%	87.25%	87.27%	44.07	36.33	23.30
	A&E	88.00%	87.60%	87.41%	44.20	36.33	23.30
	B&C	82.46%	82.38%	82.75%	46.08	36.33	23.31
	B&D	83.50%	84.14%	84.34%	44.31	36.33	23.31
	B&E	84.19%	84.48%	84.48%	44.45	36.33	23.31
	C&D	84.00%	83.37%	83.69%	44.09	36.33	23.30
	C&E	84.69%	83.71%	83.83%	44.22	36.33	23.30
	D&E	85.74%	84.47%	85.43%	42.45	36.33	23.29
	A&B&C	84.83%	84.71%	84.91%	68.99	36.33	23.34
	A&B&D	85.52%	85.88%	85.97%	67.22	36.33	23.34
	A&B&E	85.99%	86.11%	86.07%	67.36	36.33	23.34
	A&C&D	85.86%	85.37%	85.54%	67.00	36.33	23.33
	A&C&E	86.32%	85.60%	85.64%	67.13	36.33	23.32
	A&D&E	87.01%	86.77%	86.70%	65.36	36.33	23.32
	B&C&D	83.32%	83.29%	83.59%	67.24	36.33	23.34
	B&C&E	83.78%	83.52%	83.69%	67.37	36.33	23.33
	B&D&E	84.48%	84.69%	84.75%	65.60	36.33	23.33
	C&D&E	84.81%	84.18%	84.32%	65.38	36.33	23.32
	A&B&C&D	84.88%	84.82%	85.00%	90.15	36.33	23.37
	A&B&C&E	85.23%	84.99%	85.07%	90.28	36.33	23.36
	A&B&D&E	85.75%	85.87%	85.87%	88.51	36.33	23.36
	A&C&D&E	86.00%	85.48%	85.55%	88.29	36.33	23.35
	B&C&D&E	84.10%	83.92%	84.09%	88.53	36.33	23.36
	A&B&C&D&E	85.19%	85.01%	85.12%	111.44	36.33	23.39

is also the reason why the performance gain of PAM over baseline 2 on LFW is not much lower than on CelebA, although on LFW we have the highest degree of sharing.

Takeaways. Although the performance of PAM varies across tasks, it achieves consistently solid advantages over both baselines. We may conclude that it

Table 4.5 Test accuracy and computation cost with ResNet-18/ResNet-34 on CelebA pruned by P1.

Model	Tasks	Accuracy			FLOPs ($\times 10^6$)		
		B1	B2	PAM	B1	B2	PAM
ResNet-18	A	89.83%	89.30%	89.93%	5.72	8.84	4.78
	B	88.25%	88.20%	88.36%	5.72	8.84	4.83
	A&B	89.04%	88.75%	89.15%	11.44	8.84	6.40
ResNet-34	A	89.99%	89.70%	90.05%	8.43	12.11	6.94
	B	88.44%	88.98%	88.42%	8.43	12.11	6.94
	A&B	89.22%	89.34%	89.24%	16.86	12.11	10.29

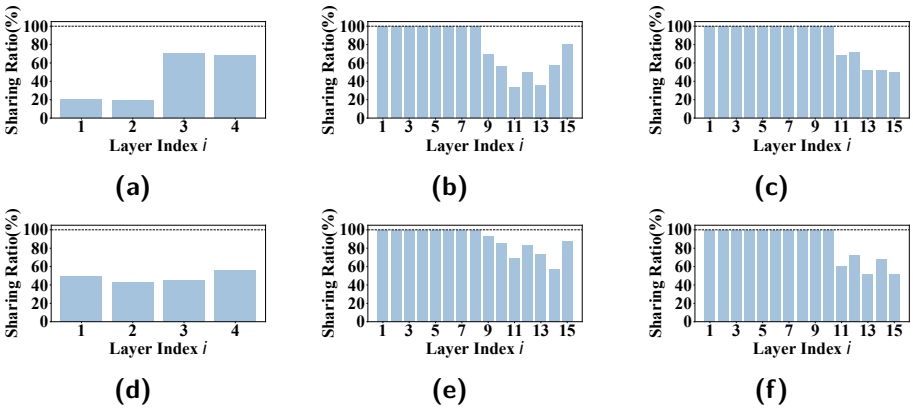


Figure 4.6 Sharing ratio of each layer after "PAM & prune (P1 or P2)" on (a) LeNet/Fashion-MNIST with P1, (b) VGG/CelebA with P1, (c) VGG/LFW with P1, (d) LeNet/Fashion-MNIST with P2, (e) VGG/CelebA with P2, and (f) VGG/LFW with P2. In each layer, the sharing ratio is calculated as the number of shared neurons in $\mathbf{L}_i'^{A,B}$, divided by all neurons in $\mathbf{L}_i^{A,B}$. It ranges from 0% to 100%.

is always preferable to use PAM for efficient multitask inference, regardless of the amount of shareable neurons, of the probability of executing each task combination, of the network architecture, or of the pruning method used after merging.

4.6.3 Ablation Study

Here we present experiments to further understand the effectiveness of PAM.

4.6.3.1 Visualisation of Algorithm 4

Fig. 4.7 illustrates two iterations of Line 19-22 and 24-27 in Algorithm 4 by showing $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ against the number of iterations. Here we use the `f7` layer of VGG-16 trained and merged for CelebA dataset as an example. The tuning parameter α is set to infinitely large in order to show all the possible cases of the iterations. From Fig. 4.7, we can observe three phases:

1. In the first phase, $I(\mathbf{L}_i'^A; \mathbf{Y}^B)$ and $I(\mathbf{L}_i'^B; \mathbf{Y}^A)$ remains small, indicating

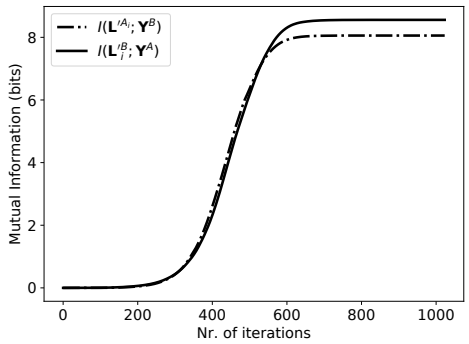


Figure 4.7 Iterations of Line 19-22 and 24-27 in Algorithm 4. The shown example is on the `f7` layer of the VGG-16 networks trained and merged on CelebA.

that the selected $\mathbf{L}'^A_i$ and $\mathbf{L}'^B_i$ provides little information about the other task.

2. In the second phase, $I(\mathbf{L}'^A_i; \mathbf{Y}^B)$ and $I(\mathbf{L}'^B_i; \mathbf{Y}^A)$ start to increase as it is impossible to add more neurons to $\mathbf{L}'^A_i$ and $\mathbf{L}'^B_i$ while keeping $I(\mathbf{L}'^A_i; \mathbf{Y}^B)$ and $I(\mathbf{L}'^B_i; \mathbf{Y}^A)$ close to zero.

3. In the third phase, $I(\mathbf{L}'^A_i; \mathbf{Y}^B)$ and $I(\mathbf{L}'^B_i; \mathbf{Y}^A)$ start to saturate as the newly joined neurons contain mostly information already included in existing $\mathbf{L}'^A_i$ and $\mathbf{L}'^B_i$.

In practice, the parameter α tuned as remains small, and the iterations in Algorithm 4 as well as Algorithm 5 usually stop at the end of the first phase or the beginning of the second phase.

4.6.3.2 Impact of Task Relatedness

This study aims to show the impact of task relatedness on the performance gain PAM can achieve. The number of neurons that can be shared among pre-trained networks is related to the relatedness among tasks. An effective network merging scheme should enforce increasing numbers of shared neurons between tasks with the increase of task relatedness.

Settings. We consider the 73 labels in LFW as 73 binary classification tasks, and measure the relatedness between each task pair by $I(\mathbf{Y}^A; \mathbf{Y}^B)$. We then pick four pairs of tasks with $I(\mathbf{Y}^A; \mathbf{Y}^B) \approx 0$, 0.1, 0.2 and 0.5 bits, train four pairs of single-task VGG-16's on them, and construct four multitask networks using PAM.

Results. Fig. 4.8a plots the number of shared neurons in layer `f7` of these four multitask networks with different tuning threshold α. The multitask networks for tasks pairs with higher correlation always share neurons. Hence, PAM can share an increasing number of neurons between tasks with the increase of task relatedness.

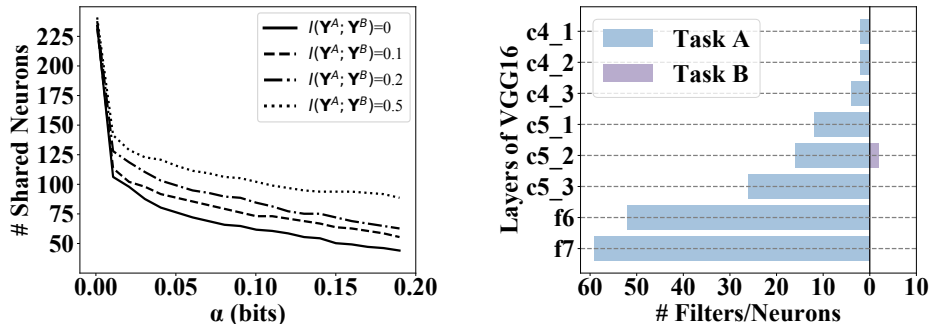


Figure 4.8 Ablation studies: (a) Number of shared neurons in layer `f7` of the four multitask networks constructed with PAM for different task pairs on LFW dataset, with different tuning parameter α. (b) The number of non-shared neurons in $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$ in the last eight layers when task B is a sub-task of task A. The networks are trained and merged on LFW.

4.6.3.3 Case Study: Task Inclusion

This study aims to validate the effectiveness of PAM in an extreme yet common case of task relatedness where task B is a sub-task of task A. Ideally, when the mutual information is precisely estimated and true largest sets of task-exclusive neurons are selected, PAM should effectively pick out only task-A-exclusive neurons.

Settings. We pick 30 labels in LFW as task A and 15 of them as task B. Hence task A includes task B. We train two single-task VGG-16's on these two tasks separately and then merge them by PAM.

Results. Fig. 4.8b shows the number of non-shared neurons in $\mathbf{L}_i'^A$ and $\mathbf{L}_i'^B$ in the last eight layers of the merged network (the previous layers have exclusively shared neurons). Almost no neurons are selected for $\mathbf{L}_i'^B$ by Algorithm 4, validating its effectiveness.

4.7 Conclusion

In this chapter, we investigate network merging schemes for efficient multitask inference. Given a set of single-task networks pre-trained for individual tasks, we aim to construct a multitask network such that applying existing network pruning methods on it can minimise the computation cost when performing any subset of tasks. We theoretically identify the conditions on the multitask network, and design Pruning-Aware Merging (PAM), a heuristic network merging scheme to construct such a multitask network. The merged multitask network can then be effectively pruned by existing network pruning methods. Extensive evaluations show that pruning a multitask network constructed by

PAM achieves low computation costs when performing any subset of tasks in the network.

So far, with the MTZ introduced in Chapter 2, the MTS introduced in Chapter 3, and the PAM introduced in this chapter, we have covered all three types of MMDL systems discussed in Sec. 1.4 and provided comprehensive solutions to the multi-model compression problem. In the next chapter, we move on to investigate another perspective of on-device intelligence: the efficient on-device adaptation discussed in Sec. 1.5.

5

Pruning Meta-Trained Networks for On-Device Adaptation

In previous chapters, with the MTZ introduced in Chapter 2, the MTS introduced in Chapter 3, and the PAM introduced in Chapter 4, we have provided comprehensive solutions to the multi-model compression problem introduced in Sec. 1.4. Now we move on to investigate another perspective of on-device intelligence: the efficient on-device adaptation discussed in Sec. 1.5.

Adapting neural networks to unseen tasks with few training samples on resource-constrained devices benefits various Internet-of-Things applications. Such neural networks should learn the new tasks with limited data and be compact. Meta-learning enables such learning with limited data, yet the meta-trained networks can be over-parameterised. This means that deploying such models on resource-constrained edge devices can be problematic. As introduced in Sec. 1.3, compression techniques like network pruning hold the potential to reduce the number of parameters in over-parameterised models drastically. However, the naive combination of standard compression techniques like network pruning with meta-learning jeopardises the ability to learn with limited data. This chapter proposes adaptation-aware network pruning (ANP), a novel pruning scheme that works with existing meta-learning methods for a compact network capable of learning with limited data. ANP uses a weight importance metric based on the meta-objective's sensitivity rather than the conventional loss function and adopts approximation of derivatives and layer-wise pruning techniques to reduce the overhead of computing the new importance metric. Evaluations on few-shot classification benchmarks show that ANP can prune meta-trained convolutional and residual networks by 85% without affecting their ability to learn with limited data.

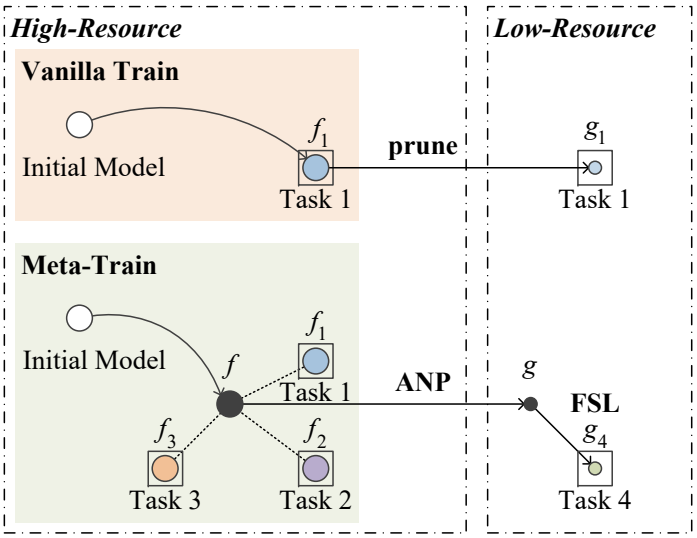


Figure 5.1 Comparison between pruning a vanilla-trained model and a meta-trained model. Existing pruning methods are designed for vanilla-trained networks, where the task is the same before and after pruning. For example, the vanilla-trained model f_1 is pruned into a compact model g_1 to deploy on low-resource platforms, where both f_1 and g_1 are optimised for Task 1. Our focus is to prune a meta-trained network, where the model is optimised on batches of tasks. The pruned meta-trained model should fast adapt to new tasks after deployment. For example, the weights for the initial architecture is meta-trained to f, which is a good initialisation for Task 1, 2, and 3. Then f is pruned into a compact model g, where g is expected to adapt into g_4 after few-shot learning (FSL) and yield high inference accuracy on Task 4.

5.1 Introduction

On-device adaptation refers to learning previously unseen tasks by updating an initial model on-board. This is desired in internet of things (IoT) applications including personal drones, home robots and self-driving vehicles, since uploading newly collected data for model updating can be infeasible due to unstable wireless connections, limited bandwidth or privacy concerns. An initial model for on-device adaptation should allow *fast adaptation* and should be *compact in size* due to the following reasons: *(i)* Only a limited amount of training data is available locally, which requires the model to adapt to new tasks with few samples. This requirement is reasonable because users are often asked to provide supervision with a few private data samples so that the general initial model can be rapidly customized, personalized, or calibrated to new tasks, users or environments without affecting user experiences. *(ii)* IoT applications often run on resource-constrained devices, where a large model easily overwhelms the computation and memory resources. In contrast to the cloud, IoT platforms powered by systems on chips (SoCs) or micro-controllers are particularly limited by the small memory system (KB to MB) to store model parameters [112].

An effective solution to enable fast adaptation is meta-learning, where the

initial model for deployment is meta-trained, and adaptation is implemented and assessed by few-shot learning [33], [34]. Of particular interest is Model-Agnostic Meta-Learning (MAML), a general gradient-based algorithm that learns the weights of an given initial architecture, such that the meta-trained model excels at few-shot learning [33]. Gradient-based algorithms [33], [113], [114] are suited for on-device adaptation since recent research due to the feasibility of gradient-based training on low-resource devices [115], [116]. Despite allowing fast adaptation, MAML fails to generate a compact model as it only optimised the network parameters, but does not alter the initial architecture [117]. The initial architecture, however, has to be over-parameterised for effective meta-training [118]. Consequently, the meta-trained model is also over-parameterised.

An intuitive remedy for compact models is network pruning, which has the potential to radically remove unimportant parameters in a neural network without deterioration in inference accuracy [8]. However, existing pruning methods [11], [12], [29], [73], [119], [120] are incompatible with meta-learning and may jeopardise the ability of fast adaptation, as they are designed to retain the *inference accuracy on a known single task*. In contrast, the goal of meta-learning (in the following called *meta-objective*) is to optimise the *ability to adapt to new tasks following certain distribution*, which differs from the objective of single task pruning. Therefore, to construct a compact network capable of fast adaptation, a new network pruning scheme , which can optimise the topology in accordance with the meta-objective, is required, as illustrated in Fig. 5.1.

In this chapter, we propose Adaptation-aware Network Pruning (ANP), a novel network pruning scheme that works in synergy with meta-learning. While meta-learning optimises the weights, ANP compresses and optimises the topology. Together we are able to construct compact neural networks capable of fast adaptation.

At a high level, ANP extends the analysis of second order derivatives for pruning vanilla-trained networks [12], [121] to meta-learning scenario. That is, ANP calculates weight importance values from the training data for *tasks sampled from a certain distribution* and removes those weights that induces minimal changes on the meta-objective. However, pruning based on such second order derivatives of the meta-objective is computation-intensive, as it requires the global third order derivatives and generalised inverses of Hessian matrices. ANP avoids calculations of third order derivatives via a novel approximation approach. It further reduces the computation overhead by layer-wise pruning such that the generalised inverse Hessian matrices are obtained efficiently and stably.

Evaluations on Mini-ImageNet [34] and Caltech-UCSD Birds-200-2011 (CUB) [122] show that ANP can prune common used initial architectures by 85% with less than 1% accuracy loss in few-shot classification and it works with different gradient-based meta-learning methods (*e.g.*, MAML [33], [113], [114]). In contrast, pruning the initial architecture to the same ratio with existing methods [11], [73] will lead to a loss of 7.01% to 26.70% in few-shot classification

accuracy.

Our main contributions and results are as follows.

- To the best of our knowledge, this is the first investigation of pruning *meta-trained* neural networks for model compression. Due to the inconsistency between the meta-objective and the weight importance metrics in network pruning, naive combination of pruning and meta-learning deteriorates the model adaptability in few-shot learning.
- We design ANP, a novel meta-learning-compatible network pruning scheme. ANP can prune over-parameterized meta-trained networks without sacrificing their ability for fast adaptation. It applies approximation of derivatives and layer-wise pruning to reduce the computation overhead in pruning meta-trained deep models.
- Evaluations on few-shot classification benchmarks show that ANP can prune the initial architectures for meta-learning by 85% while retaining the few-shot classification accuracy.

In the rest of this chapter, we review related work in Sec. 5.2, introduce our ANP method in Sec. 4.5, present its evaluations in Sec. 5.4 and conclude in Sec. 5.5.

5.2 Related Work

Our work is relevant to the following threads of research.

Meta-Learning for Few-Shot Learning. Training a deep neural network upon limited samples *i.e.*, in few-shots, tends to overfit [123]. Meta-learning has been a successful solution to few-shot learning [124], [125], where the meta-trained model is able to learn a new task from a few training samples. In this chapter, we focus on gradient-based meta-learning methods [33], [113], [114] for their applicability in various learning tasks and the potential to enable on-device adaptation. Specifically, we aim to generate an initial model that can fast adapt to new tasks and is compact in size.

MAML-like algorithms [33], [113], [114] only adapt *weights* of the initial architecture without alerting its topology. A few studies [117], [126] propose to integrate MAML with neural architecture search to optimise the initial architecture. Since their primary goal is higher few-shot inference accuracy, the resulting model can even have more parameters than the initial architecture in MAML [117]. Our work also adapts the initial architecture, yet with a complementary objective. Particularly, we sparsify it without sacrificing its ability of fast adaptation, which results in a much smaller model.

Network Pruning. Given an over-parameterised network well-trained for a given task, network pruning eliminates unimportant parameters without major accuracy loss on the inference task [8]. Fine-grained pruning (*e.g.*, weights)

[11], [73] results in a higher compression rate whereas coarse-grained pruning (*e.g.*, filters) [10], [127] is a better fit for acceleration on commodity hardware. Various importance criteria have been proposed, such as magnitude [73], second order derivatives [11], [12], [29], and information bottleneck [10]. However, all existing parameter importance metrics are derived for vanilla-trained networks, *i.e.*, the pruned network targets at the same task as before pruning. In contrast, we propose a new weight importance metric for meta-trained networks, where the pruned network should fast adapt to new tasks unseen before pruning.

A very recent study [128] explored improving the meta-training procedure via pruning. Specifically, in [128], pruning is applied as a model capacity constraint to avoid meta-overfitting, where the pruned parameters are re-activated during the retraining phase. Its final output is still a *large dense* network which is unfit for deployment on resource-constrained devices. As will be shown in our evaluations (see Sec. 5.4.2), directly combining iterative hard thresholding (equivalent to the "Magnitude" baseline in our work) and meta-learning method like Reptile [113] as in [128] leads to significant drop in few-shot classification accuracy at high pruning ratios.

5.3 Method

In this section, we first provides a primer on meta learning (Sec. 5.3.1) and then explain our ANP in detail. Specifically, we introduce our new weight importance metric for pruning meta-trained networks (Sec. 5.3.2), followed by derivative approximations (Sec. 5.3.3) and layer-wise pruning (Sec. 5.3.4) for efficient calculating of the weight importance metric, and finally present the complete algorithm (Sec. 5.3.5).

5.3.1 Primer on MAML

Model-Agnostic Meta-Learning (MAML) [33] learns an initial model f such that given a new task, f can learn it with a few training samples. MAML is a two-tier gradient decent based optimisation process. In each iteration of the optimisation, M tasks with corresponding training datasets $\{\mathcal{D}_i\}, i \in \{1, \cdots, M\}$ are sampled from a certain distribution. We use $\boldsymbol{\theta}$ to represent the current parameters of f. In each iteration, $\boldsymbol{\theta}$ is updated to $\boldsymbol{\theta}''$ as follows.

$$\boldsymbol{\theta}'^i = \boldsymbol{\theta} - \alpha \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}_i\right) \tag{5.1}$$

$$\boldsymbol{\theta}'' = \boldsymbol{\theta} - \beta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}_m \tag{5.2}$$

where

$$\mathcal{L}_m = \frac{1}{M} \sum_{i=1}^{M} \mathcal{L}\left(\boldsymbol{\theta}'^i, \mathcal{D}_i\right) \tag{5.3}$$

is called the meta-objective, $\mathcal{L}(\cdot, \cdot)$ is a loss function, and α and β are learning rates. The inner-loop gradient decent (5.1) updates parameters $\{\boldsymbol{\theta}'^i\}$ for task-

specific objectives. Note that $\boldsymbol{\theta}'^i$ is the vector for parameters trained on task i. The outer-loop gradient descent (5.2) then updates the parameters for the meta-objective. Since the meta-objective $\mathcal{L}_m$ contains first derivatives of $\boldsymbol{\theta}$ in (5.1), the gradient in (5.2) is in effect a second derivative of $\boldsymbol{\theta}$.

5.3.2 Weight Importance in Meta-Training

Pruning eliminates unimportant weights in the network, where the weight importance is assessed by the impact of its removal on the inference accuracy. A classic and effective approach to quantify weight importance is an analysis based on second order derivatives, which measures the change in the objective caused by a weight change [11], [12], [29], [121]. Weight importance of vanilla-trained neural networks is defined using the traditional loss functions as the objective. We now define weight importance for meta-trained neural networks via an analysis based on the second order derivatives of the meta-objective as (5.3).

Defining Weight Importance. To quantify the weight importance of meta-trained networks, the conventional loss function is replaced by the meta-objective. Specifically, the Taylor series of the change in the meta-objective due to a parameter change is

$$\delta\mathcal{L}_m = \left(\frac{\partial\mathcal{L}_m}{\partial\boldsymbol{\theta}}\right)^\top \delta\boldsymbol{\theta} + \frac{1}{2}\delta\boldsymbol{\theta}^\top\mathbf{H}\delta\boldsymbol{\theta} + O(||\delta\boldsymbol{\theta}||^3) \tag{5.4}$$

where $\mathbf{H} = \frac{\partial^2\mathcal{L}_m}{\partial\boldsymbol{\theta}^2}$ is the Hessian matrix of the meta-objective with respect to $\boldsymbol{\theta}$. Similar to the analysis of second derivatives for vanilla-trained networks [12], [121], the first term vanishes for a well meta-trained network, and the higher order derivatives can be ignored. Therefore,

$$\delta\mathcal{L}_m \approx \frac{1}{2}\delta\boldsymbol{\theta}^\top\mathbf{H}\delta\boldsymbol{\theta} \tag{5.5}$$

Then, identifying the q-th weight in parameter $\boldsymbol{\theta}$ that minimizes the impact on the meta-objective can be formulated as the following optimisation problem:

$$\min_q \quad \frac{1}{2}\delta\boldsymbol{\theta}^\top\mathbf{H}\delta\boldsymbol{\theta} \quad \text{s.t.} \quad \mathbf{e}_q^\top\delta\boldsymbol{\theta} + \boldsymbol{\theta}_q = 0 \tag{5.6}$$

where $\mathbf{e}_q$ is the unit vector whose q-th element is 1 and otherwise 0, and $\boldsymbol{\theta}_q$ is the same as $\boldsymbol{\theta}$ except that the q-th element is set to 0. Forming a Lagrange from (5.6) as in [12], we find a closed-form solution for $\delta\boldsymbol{\theta}$

$$\delta\boldsymbol{\theta} = -\frac{\boldsymbol{\theta}_q}{[\mathbf{H}^{-1}]_{qq}}\mathbf{H}^{-1}\mathbf{e}_q \tag{5.7}$$

and the corresponding minimal change in the meta-objective

$$\Delta\mathcal{L}_m = \frac{1}{2}\frac{\boldsymbol{\theta}_q^2}{[\mathbf{H}^{-1}]_{qq}} \tag{5.8}$$

This term is also considered as the importance of the q-th element (weight) in the parameter vector $\boldsymbol{\theta}$.

Challenges to Compute Hessian. From (5.8), it seems that the weight importance metric for pruning vanilla-trained and meta-trained networks share the same form, except that the Hessian matrix is defined on the meta-objective rather than the traditional loss function. We show that naively calculating the Hessian involves third order derivatives with respect to $\boldsymbol{\theta}$, which is computation-intensive for deep neural networks.

Assume $\boldsymbol{\theta} \in \mathbb{R}^d$. Then $\mathbf{H} \in \mathbb{R}^{d \times d}$ and each element $H_{m,n}$ in $\mathbf{H}$ is computed as

$$H_{m,n} = \frac{\partial \mathcal{L}_m}{\partial \theta_m, \theta_n} \tag{5.9}$$

$$= \frac{1}{M} \sum_{i=1}^{M} \frac{\partial \mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial \theta_m, \theta_n} \tag{5.10}$$

$$= \frac{1}{M} \sum_{i=1}^{M} \frac{\partial \mathcal{L}\left(\boldsymbol{\theta} - \alpha \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}_i\right), \mathcal{D}_i\right)}{\partial \theta_m, \theta_n} \tag{5.11}$$

where (5.10) and (5.11) simply substitute $\mathcal{L}_m$ and $\boldsymbol{\theta}'^i$ with (5.2) and (5.1).

From (5.11), each element in $\mathbf{H}$ requires computing third order derivatives with respect to $\boldsymbol{\theta}$.

5.3.3 Approximation of Derivatives

We avoid computing the third order derivatives in (5.11) by approximating them as follows.

Since $\boldsymbol{\theta}'^i$ is a function of θ_m and θ_n, we apply the Faà di Bruno's formula [129] to (5.10):

$$\begin{aligned} H_{m,n} &= \frac{1}{M} \sum_{i=1}^{M} \frac{\partial \mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial \theta_m, \theta_n} \\ &= \frac{1}{M} \sum_{i=1}^{M} \left(\sum_k \frac{\partial \mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial \theta_k'^i} \frac{\partial \theta_k'^i}{\partial \theta_m, \theta_n} + \sum_{k,l} \frac{\partial \mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial \theta_k'^i, \theta_l'^i} \frac{\partial \theta_k'^i}{\partial \theta_m} \frac{\partial \theta_l'^i}{\partial \theta_n} \right) \end{aligned} \tag{5.12}$$

From (5.1), we know

$$\frac{\partial \theta_k'^i}{\partial \theta_m} = \begin{cases} 1 - \alpha \dfrac{\partial \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_i)}{\partial \theta_k, \theta_m}, & \text{if } k = m, \\ -\alpha \dfrac{\partial \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_i)}{\partial \theta_k, \theta_m}, & \text{if } k \neq m \end{cases} \tag{5.13}$$

Omitting the second derivative $\frac{\partial}{\partial \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_i)} / \theta_k, \theta_m$, we obtain the first order approximation for $\frac{\partial}{\partial \theta_k'^i} / \theta_m$

$$\frac{\partial \theta_k'^i}{\partial \theta_m} \approx \begin{cases} 1, & \text{if } k = m, \\ 0, & \text{if } k \neq m \end{cases} \tag{5.14}$$

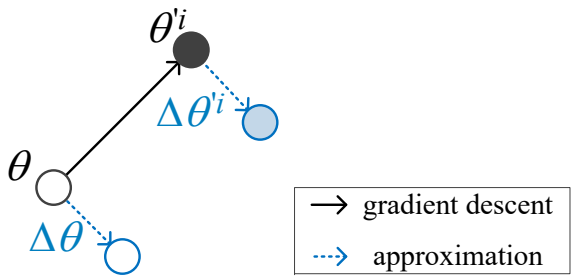


Figure 5.2 Derivative approximation in (5.17). A small change $\Delta\boldsymbol{\theta}$ in $\boldsymbol{\theta}$ induces approximately the same change $\Delta\boldsymbol{\theta}'^i$ as in $\boldsymbol{\theta}'^i$. It especially well for a well-meta-trained network, as $\nabla_{\boldsymbol{\theta}}\mathcal{L}_m$ is small while $\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}_i\right)$ remains large.

And therefore,

$$\frac{\partial\theta'^i_m}{\partial\theta_m, \theta_n} = 0 \tag{5.15}$$

The first term in (5.12) is therefore vanished, and according to (5.14), we can also simplify the second term with

$$\sum_{k,l}\frac{\partial\mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial\theta'^i_k, \theta'^i_l}\frac{\partial\theta'^i_k}{\partial\theta_m}\frac{\partial\theta'^i_l}{\partial\theta_n} = \frac{\partial\mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial\theta'^i_m, \theta'^i_n}\frac{\partial\theta'^i_m}{\partial\theta_m}\frac{\partial\theta'^i_n}{\partial\theta_n} = \frac{\partial\mathcal{L}(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial\theta'^i_m, \theta'^i_n}. \tag{5.16}$$

which finally leads to

$$H_{m,n} \approx \frac{1}{M}\sum_{i=1}^{M}\frac{\partial\mathcal{L}(\boldsymbol{\theta}'^i\mathcal{D}_i)}{\partial\theta'^i_m, \theta'^i_n}. \tag{5.17}$$

In summary, (5.17) approximates the computation of the third order derivatives w.r.t. $\boldsymbol{\theta}$ by calculating the second derivatives w.r.t. $\boldsymbol{\theta}'^i$.

Understanding the Derivatives Approximation. The approximation used for reducing the computation of third derivatives to second derivatives relies on (5.14), which suggests that a small change in the *pre-adaptation* parameters $\boldsymbol{\theta}$ leads to approximately the same small change in the *post-adaptation* parameters $\boldsymbol{\theta}'^i$ (see Fig. 5.2). Such an approximation is reasonable because for a well-meta-trained network, $\nabla_{\boldsymbol{\theta}}\mathcal{L}_m$ is small as the network nearly converged. Meanwhile, $\nabla_{\boldsymbol{\theta}}\mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}_i\right)$ is large, as the tasks can substantially differ from each other. In essence, (5.14) performs a first order approximation by omitting $\frac{\partial}{\partial\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_i)}/\theta_k, \theta_m$. Such an approach is supported by observations in other studies that the second derivatives are usually close to nought [130]. Prior studies [33], [113] have also shown that first order approximations can be as effective as full second derivatives in meta-learning.

5.3.4 Layer-Wise Pruning

To further reduce the computation, we adapt the layer-wise approach for pruning vanilla-trained networks [11] and expand it to the pruning of meta-trained networks.

Layer-Wise Meta-Objective. The pre-activation output vector of the l-th layer is denoted as $\mathbf{y}_l$, and the post-activation output vector is denoted as $\mathbf{z}_l = \sigma(\mathbf{y}_l)$, where $\sigma\cdot$ is the activation function. We use a layer-wise loss function

$$\mathcal{L}^l\left(\boldsymbol{\theta}, \mathcal{D}_i\right) = \frac{1}{K}\sum^{K} ||\hat{\mathbf{y}}_l - \mathbf{y}_l||^2 \tag{5.18}$$

where $\hat{\mathbf{y}}_l$ is the pre-activation output after the pruning, K is the number of training samples, and $||\cdot||$ is the l^2-norm. Note that the summation is over all training samples from the same task. The layer-wise meta-objective is then

$$\mathcal{L}^l_m = \frac{1}{M}\sum_{i=1}^{M}\mathcal{L}^l\left(\boldsymbol{\theta}'^i, \mathcal{D}_i\right) \tag{5.19}$$

Layer-Wise Hessian. As the network is pruned layer by layer, we need only the Hessian of $\mathcal{L}^l_m$ w.r.t. $\boldsymbol{\theta}_l$, which is the equivalent of $\boldsymbol{\theta}$ for the l-th layer. From (5.17), we find

$$\mathbf{H}_l \approx \frac{\partial^2\mathcal{L}^l_m}{\partial(\boldsymbol{\theta}'^i_l)^2} = \frac{1}{M}\sum_{i=1}^{M}\frac{\partial^2\mathcal{L}^l(\boldsymbol{\theta}'^i, \mathcal{D}_i)}{\partial(\boldsymbol{\theta}'^i_l)^2}. \tag{5.20}$$

where $\boldsymbol{\theta}'^i_l$ is the equivalent of $\boldsymbol{\theta}'^i$ (see (5.1)) for the l-th layer.

Similar to [11], $\mathbf{H}_l$ is a block diagonal square matrix with each diagonal blocks being $\mathbf{H}_{l_{kk}} = \frac{\partial^2\mathcal{L}^l_m}{\partial(\boldsymbol{\theta}'^i_{l_k})^2}$, where $\boldsymbol{\theta}'^i_{l_k}$ are the vectorised incoming weights of the k-th neuron in the l-th layer. All blocks $\mathbf{H}_{l_{kk}}$ are identical and can be calculated as

$$\mathbf{H}_{l_{kk}} = \frac{1}{M}\frac{1}{K}\sum_{i=1}^{M}\sum^{K}\mathbf{z}_{l-1}\cdot(\mathbf{z}_{l-1})^\top \tag{5.21}$$

Efficient Computing of Inverse of the Hessian. As shown in (5.7) and (5.8), we need the inverse of the Hessian $\mathbf{H}_l^{-1}$, a block diagonal square matrix with its diagonal blocks being $\mathbf{H}_{l_{kk}}^{-1}$. In ANP, we calculate $\mathbf{H}_l^{-1}$ recursively over the training samples of all M tasks using the Sherman-Morrison-Woodbury formula [131]

$$\begin{aligned}\mathbf{H}^{-1}_{l_{kk},j} &= \mathbf{H}^{-1}_{l_{kk},j-1} - \frac{\mathbf{H}^{-1}_{l_{kk},j-1}\cdot\mathbf{z}_{l-1,j}\cdot\mathbf{z}^\top_{l-1,j}\cdot\mathbf{H}^{-1}_{l_{kk},j-1}}{M\cdot K + \mathbf{z}^\top_{l-1,j}\cdot\mathbf{H}^{-1}_{l_{kk},j-1}\cdot\mathbf{z}_{l-1,j}} \\ \text{with}\quad & \mathbf{H}^{-1}_{l_{kk},0} = \alpha^{-1}\mathbf{I} \text{ and } \mathbf{H}^{-1}_{l_{kk},M\cdot K} = \mathbf{H}^{-1}_{l_{kk}}\end{aligned} \tag{5.22}$$

where $\alpha \in [10^{-8}, 10^{-4}]$ is a small constant to make $\mathbf{H}^{-1}_{l_{kk},0}$ meaningful and to which the method is insensitive [12]. Note that the two summations in (5.21) are integrated in (5.22), as the iteration goes through all $M\cdot K$ samples.

Algorithm 6: Adaptation-aware Network Pruning

Input: $p(\mathcal{T})$: distribution over tasks
α: a small constant ($10^{-8} \leq \alpha \leq 10^{-4}$)
β_l: pruning step size hyper-parameter ($0 < \beta_l < 1$)
γ_{pr}: pruning ratio ($0 < \gamma_{pr} < 1$)
Output: Sparse network

1 randomly initialise weights $\boldsymbol{\theta}$
2 meta-train the network until the meta-objective converges
3 **while** *required* γ_{pr} *not achieved* **do**
4 sample a batch of M tasks $\mathcal{T}_i \sim p(\mathcal{L})$
5 **for** *each task* $\mathcal{T}_i$ **do**
6 sample K data-points from $\mathcal{T}_i$ and form datasets $\mathcal{D}_i$
7 compute post-adaptation parameters $\boldsymbol{\theta}'^i$ with Eq.(5.1)
8 **end**
9 **for** *all layers* **do**
10 calculate $\mathbf{H}^{-1}_{l_{kk}}$ recursively using Eq.(5.22)
11 calculate $\delta\boldsymbol{\theta}$ and $\Delta\mathcal{L}_m$ for each weight using Eq.(5.7) and Eq.(5.8), respectively
12 prune β_l of the weights with the least $\Delta\mathcal{L}_m$, and update the rest with $\delta\boldsymbol{\theta}$
13 **end**
14 meta-train the network again until the meta-objective converges, such that the performance is re-boosted
15 **end**
16 **return** *the sparse network*

5.3.5 Putting It Together

Algorithm 6 outlines the process of ANP for K-shot learning. The pruning first begins after the network is well-meta-trained (Line 2). As in MAML, a batch of M tasks are sampled from a given distribution $p(\mathcal{T})$ (Line 4), and K data-points are sampled from each task (Line 6). Then the post-adaptation $\boldsymbol{\theta}'^i$ is calculated with gradient descent (5.1) for each task. As the training samples and post-adaptation weights are ready, the inverse Hessian can be recursively calculated with (5.22) (Line 9). The pruning is done iteratively (Line 3). We use a tuning parameter β_l to control the proportion of weights to be pruned in each iteration. The importance $\Delta\mathcal{L}_m$ of each weight is assessed with (5.8) (Line 10), and β_l of the least important weights in each layer are removed, while the remaining weights are updated using those $\delta\boldsymbol{\theta}$ calculated with (5.7). Combining derivatives approximation (Sec. 5.3.3) and layer-wise pruning approach (Sec. 5.3.4), ANP is able to prune meta-trained neural network effectively.

Extensions beyond MAML. It is worth mentioning that ANP is not restricted to MAML. Here we briefly explain how to extend Algorithm 6 to two popular variants of MAML: CAVIA [114], an improvement of MAML with context parameters, and Reptile [113], the first order simplification of MAML. On CAVIA, as the number of the context parameters is limited, we can apply Algorithm 6 only to the weights *i.e.*, non-context parameters in the initial architecture. On Reptile, the meta-objective (5.3) is already omitted. We compute an averaged importance of weights in the inner loop, then prune the least important weights as in Algorithm 6.

5.4 Evaluation

This section presents the evaluations of our method.

5.4.1 Experimental Settings

Metrics. Since we aim at pruning meta-trained networks without sacrificing their ability of fast adaptation, we compare different methods with the following metrics:

- *Pruning Ratio (PR)*: the ratio of pruned parameters to the original parameters of the initial architecture.
- *Few-Shot Accuracy*: the few-shot classification accuracy.

In experiments where many testing tasks are available, we conduct multi-run testing by repeatedly selecting 5 random tasks for the few-shot learning test. In Fig. 5.3, Fig. 5.4, Fig. 5.5 and Fig. 5.6, we use error bars to represent the standard deviation over multiple runs.

Datasets. We use two standard few-shot classification benchmarks.

- Mini-ImageNet [34]: it is a dataset for image classification, which contains $60,000$ colour images with 100 classes, each having 600 images of size 84×84. The dataset is split into 64 training classes, 12 validation classes, and 24 test classes as [33], [114]. We use 5-way 1-shot and 5-way 5-shot settings.
- Caltech-UCSD Birds-200-2011 (CUB) [122]: it is a dataset for fine-grained classification, which contains $11,788$ images of 200 bird species, each having about 60 images. The dataset is split into 100 training classes, 50 validation classes, and 50 test classes and the images are resized to 84×84 as [132]. We use the 5-way 1-shot setting.

Initial Architectures. We use two common initial architectures from gradient-based meta-learning literature [33], [113], [114], [133].

- ConvNet-4: it consists of 4 layers with 3×3 convolutions followed by batch normalisation, ReLU, and 2×2 max-pooling. We use 32-filter convolutions for evaluations as in [33], [114], [132].
- ResNet-12: it consists of 4 residual blocks, each containing three 3×3 convolutional layers [132], [133]. In each residual block, the first two convolution layers are followed by batch normalisation and ReLu, and the last convolution layer is followed by batch normalisation and a skip connection. A 2×2 max-pooling is used after each residual block. The number of filters in each residual block is 64, 128, 256, and 512.

Methods for Meta-Training. By default, the weights is meta-trained by MAML [33]. We also compare ANP with baselines using the more advanced CAVIA method [114] and the popular first-order method Reptile [113]. The context parameters in CAVIA follow the default settings, which is a 100

dimensional vector initialized as 0 before each adaptation step and update during the inner training loops. For Reptile, we follow the hyper-parameter settings in [113]. Table 5.1 summarises the hyperparameters to meta-train the initial architecture via MAML, CAVIA and Reptile.

All the experiments use Adam optimizer for meta training and SGD optimizer for inner training loops with default hyperparameters. When optimizing the initial architectures, the pruned weights are set as zero and their gradients are masked by point-wise production with a zero-one matrix.

Baselines. Since ANP is a pruning scheme for meta-trained networks, we compare it with two existing pruning methods.

- Magnitude [73]: a classic pruning method that removes network weights based on their magnitude.
- L-OBS [11]: a representative Hessian-based pruning strategy that removes weights by a layer-wise Hessian-based metric.

In contrast to ANP, Magnitude and L-OBS are originally designed for single-task pruning. Therefore, one *pruning target task* has to be given for pruning, then the algorithms assess and prune the weights based on their importance to this pruning target task. In our experiments, we randomly select a pruning target tasks. During each pruning iteration, about 10% weights in each layer are pruned, and the pruned model is retrained for 40 epochs. We repeat this step-wise pruning iteration until a desired pruning ratio is reached.

Table 5.1 Hyperparameter setup for meta-training.

Method	Backbone	Dataset	Setup	Inner/Outer LR	Meta Epoch	LR Decay	Meta Batch Size	# Update Step
MAML [33]	ConvNet-4	Mini-ImageNet	5-way, 1-shot	$1.0 \times 10^{-2}/1.0 \times 10^{-3}$	$80,000$	-	4	5
			5-way, 5-shot	$1.0 \times 10^{-2}/1.0 \times 10^{-3}$	$80,000$	-	4	5
		CUB	5-way, 1-shot	$1.0 \times 10^{-2}/1.0 \times 10^{-3}$	$80,000$	-	4	5
			5-way, 5-shot	$1.0 \times 10^{-2}/1.0 \times 10^{-3}$	$80,000$	-	4	5
	ResNet-12	Mini-ImageNet	5-way, 1-shot	$1.0 \times 10^{-2}/1.0 \times 10^{-3}$	$100,000$	-	4	5
		CUB	5-way, 1-shot	$1.0 \times 10^{-2}/1.0 \times 10^{-3}$	$100,000$	-	4	5
CAVIA [114]	ConvNet-4	Mini-ImageNet	5-way, 1-shot	$1.0 \times 10^{-2}/1.0$	$200,000$	0.9	16	2
		CUB	5-way, 1-shot	$1.0 \times 10^{-3}/1.0$	$200,000$	0.9	16	2
Reptile [113]	ConvNet-4	Mini-ImageNet	5-way, 1-shot	$1.0 \times 10^{-2}/1.0$	$100,000$	1.0×10^{-5}	5	50
		CUB	5-way, 1-shot	$1.0 \times 10^{-2}/1.0$	$100,000$	1.0×10^{-5}	5	50

5.4.2 Main Experimental Results

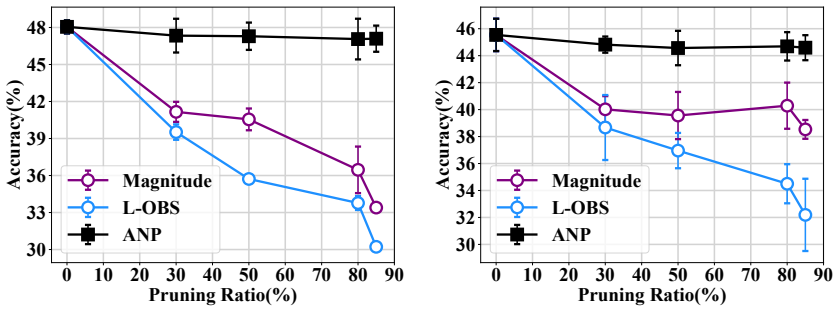
Results Organisation. We have conducted extensive experiments and compared the performance of ANP with the two baselines on varieties of initial architectures (ConvNet-4 and ResNet-12) , meta-training methods (MAML, CAVIA and Reptile), datasets (Mini-ImageNet and CUB) and FSL setups (5-way 1-shot and 5-way 5-shot). In general, we conducted five experiments and their results are organised as follows:

- **Exp.1:** We first use ConvNet-4 as initial architecture, MAML as meta-training methods, and 5-way 1-shot as the FSL setup. The comparison of ANP with the baselines is shown in Fig. 5.3, and the detailed results are listed in Table 5.2 & Table 5.3.
- **Exp.2:** On the basis of Exp.1, we change the few-shot learning setup to 5-way 5-shot. Results are shown in Fig. 5.4 and details in Table 5.4 & Table 5.5.
- **Exp.3:** On the basis of Exp.1, we change the initial architecture to the more complicated ResNet-12. Results are shown in Fig. 5.5 and details in Table 5.6 & Table 5.7.
- **Exp.4:** On the basis of Exp.1, we change the meta-training method to the more advanced CAVIA. Results are shown in Fig. 5.6 and details in Table 5.8 & Table 5.9.
- **Exp.5:** Finally, similar to Exp.4, we change the meta-training method to the popular first-order method Reptile. All other settings are the same as Exp.1. Results are shown in Fig. 5.7 and details in Table 5.10 & Table 5.11.

All the aforementioned experiments are conducted on both the Mini-ImageNet and CUB dataset. The error bars in the figures represent the standard error over different tasks chosen for few-shot learning.

Overall Performance. Inducing a decrease in few-shot accuracy no more than 1%, our ANP achieves a pruning ratio of at least 85% across all the initial architectures, datasets and meta-training methods. In comparison, given a pruning ratio of 85%, Magnitude induces a drop of 7.01% to 19.80% in few-shot accuracy, and L-OBS introduces a drop of 10.05% to 26.70% in few-shot accuracy. In fact, the baselines already induce over 5.16% loss in few-shot accuracy at a pruning ratio of 30% even in the best case (5-way 1-shot on CUB, ResNet-12 as initial architecture, pruned by Magnitude and meta-trained by MAML).

Takeaways. ANP significantly and consistently outperforms both baselines across different scenarios (1-shot or 5-shot), regardless of initial architectures (ConvNet-4 or ResNet-12) , meta-training methods (MAML, CAVIA or Reptile) and datasets (Mini-ImageNet or CUB). It turns out that ANP can find a *topology* that is meta-optimised for potential new tasks and, combined with existing meta-learning methods which find the meta-optimised *weights*, provides in the end a compact network for fast adaptation.



(a) Mini-ImageNet **(b)** CUB

Figure 5.3 **5-way 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **MAML** and pruned by Magnitude, L-OBS, or ANP on (a) Mini-ImageNet and (b) CUB.

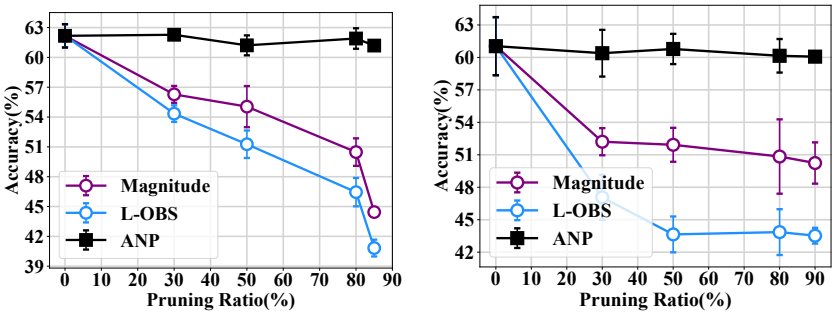
Table 5.2 **5-way, 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **MAML** and pruned on Mini-ImageNet.

PR (%)	Mini-ImageNet Accuracy (%)		
	Magnitude	L-OBS	ANP
0	48.05 ± 0.0053		
30	$41.16\ 6.89\downarrow \pm 0.0081$	$39.52\ 8.53\downarrow \pm 0.0063$	$47.34\ 0.71\downarrow \pm 0.0136$
50	$40.55\ 7.50\downarrow \pm 0.0088$	$35.72\ 12.33\downarrow \pm 0.0027$	$47.29\ 0.76\downarrow \pm 0.0111$
80	$36.46\ 11.59\downarrow \pm 0.0189$	$33.77\ 14.28\downarrow \pm 0.0057$	$47.06\ 0.99\downarrow \pm 0.0165$
85	$33.40\ 14.65\downarrow \pm 0.0036$	$30.22\ 17.83\downarrow \pm 0.0034$	$47.09\ 0.96\downarrow \pm 0.0106$

Table 5.3 **5-way, 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **MAML** and pruned on CUB.

PR (%)	CUB Accuracy (%)		
	Magnitude	L-OBS	ANP
0	45.54 ± 0.0120		
30	$40.02\ 5.52\downarrow \pm 0.0098$	$38.67\ 6.87\downarrow \pm 0.0241$	$44.82\ 0.72\downarrow \pm 0.0060$
50	$39.56\ 5.98\downarrow \pm 0.0175$	$36.96\ 8.58\downarrow \pm 0.0131$	$44.57\ 0.97\downarrow \pm 0.0128$
80	$40.29\ 5.26\downarrow \pm 0.0171$	$34.50\ 11.04\downarrow \pm 0.0145$	$44.69\ 0.85\downarrow \pm 0.0105$
85	$38.53\ 7.01\downarrow \pm 0.0070$	$32.19\ 13.35\downarrow \pm 0.0268$	$44.59\ 0.95\downarrow \pm 0.0093$

Observations and Comments. From the aforementioned experiment results, we made the following observations. *(i)* L-OBS generally performs worse on meta-learning than Magnitude. The reason may be that L-OBS is able to find a topology more specialised for the "target task" than Magnitude, which leads to actually worse FSL performance. This is further discussed in Sec. 5.4.3. *(ii)* When used with Reptile, the first-order approximation of ANP is not required. The significant advantage against the baselines still persists, which is the consequence of the better optimisation target of ANP: finding the topology optimised for all potential new tasks instead of the single "target task". *(iii)* In some cases with low pruning ratio (30% to 50%), ANP is even capable of



(a) Mini-ImageNet **(b)** CUB

Figure 5.4 **5-way 5-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **MAML** and pruned by Magnitude, L-OBS, or ANP on (a) Mini-ImageNet and (b) CUB.

Table 5.4 **5-way, 5-shot** accuracy vs.PR of **ConvNet-4** meta-trained by **MAML** and pruned on Mini-ImageNet.

PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	62.17 ± 0.0117		
30	$56.28_{5.89\downarrow} \pm 0.0085$	$54.35_{7.82\downarrow} \pm 0.0083$	$62.28_{0.11\uparrow} \pm 0.0037$
50	$55.05_{7.12\downarrow} \pm 0.0207$	$51.27_{10.90\downarrow} \pm 0.0139$	$61.21_{0.96\downarrow} \pm 0.0100$
80	$50.48_{11.69\downarrow} \pm 0.0138$	$46.45_{15.72\downarrow} \pm 0.0143$	$61.90_{0.27\downarrow} \pm 0.0103$
85	$44.44_{17.73\downarrow} \pm 0.0045$	$40.82_{21.35\downarrow} \pm 0.0085$	$61.19_{0.98\downarrow} \pm 0.0020$

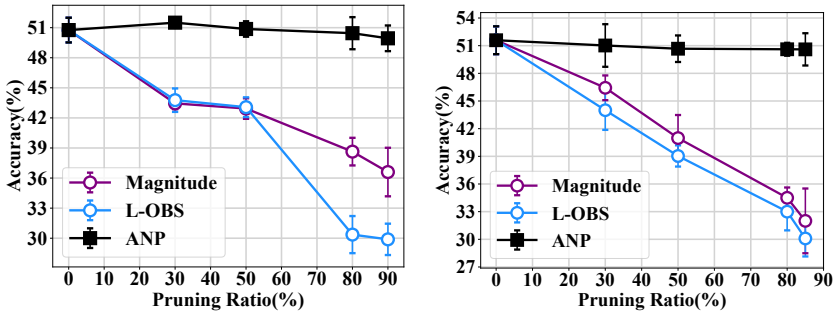
Table 5.5 **5-way, 5-shot** accuracy vs.PR of **ConvNet-4** meta-trained by **MAML** and pruned on CUB.

PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	61.04 ± 0.0268		
30	$52.21_{8.83\downarrow} \pm 0.0208$	$47.07_{13.97\downarrow} \pm 0.0125$	$60.39_{0.65\downarrow} \pm 0.0216$
50	$51.92_{9.12\downarrow} \pm 0.0166$	$43.64_{17.40\downarrow} \pm 0.0157$	$60.78_{0.26\downarrow} \pm 0.0140$
80	$50.84_{10.20\downarrow} \pm 0.0212$	$43.85_{17.19\downarrow} \pm 0.0344$	$60.15_{0.89\downarrow} \pm 0.0155$
90	$50.24_{10.80\downarrow} \pm 0.0073$	$43.52_{17.52\downarrow} \pm 0.0191$	$60.07_{0.97\downarrow} \pm 0.0036$

slightly improving the FSL accuracy. This is due to the generalisation effect known for network pruning.

5.4.3 Ablation Study

Why Baseline Pruning Methods Fail. As mentioned in Sec. 5.4.1, the baseline methods require one pruning target task during the pruning. Although the pruned networks are meta-trained and therefore their weights are considered to be optimised for all the potential new tasks, their topology, constructed via



(a) Mini-ImageNet **(b)** CUB

Figure 5.5 **5-way 1-shot** accuracy vs. PR of **ResNet-12** meta-trained by **MAML** and pruned by Magnitude, L-OBS, or ANP on (a) Mini-ImageNet and (b) CUB.

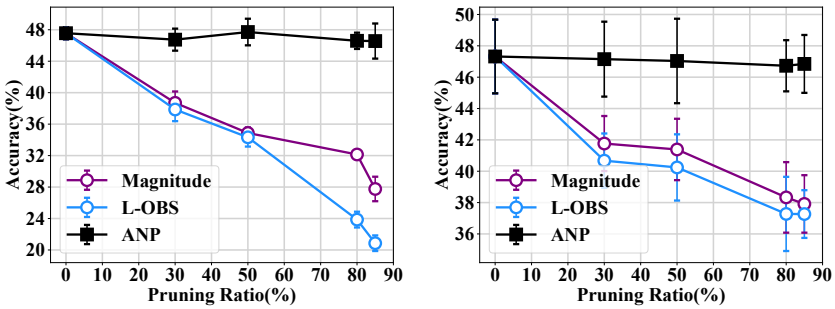
Table 5.6 **5-way, 1-shot** accuracy vs. PR of **ResNet-12** meta-trained by **MAML** and pruned on Mini-ImageNet.

PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	50.76 ± 0.0123		
30	$43.45\ 7.31\downarrow \pm 0.0058$	$43.77\ 6.99\downarrow \pm 0.0117$	$51.50\ 0.74\uparrow \pm 0.0028$
50	$42.93\ 7.83\downarrow \pm 0.0102$	$43.07\ 7.69\downarrow \pm 0.0098$	$50.87\ 0.11\uparrow \pm 0.0077$
80	$38.63\ 12.13\downarrow \pm 0.0138$	$30.36\ 20.40\downarrow \pm 0.0185$	$50.45\ 0.31\downarrow \pm 0.0160$
90	$36.60\ 14.16\downarrow \pm 0.0242$	$29.89\ 20.87\downarrow \pm 0.0156$	$49.94\ 0.82\downarrow \pm 0.0128$

Table 5.7 **5-way, 1-shot** accuracy vs. PR of **ResNet-12** meta-trained by **MAML** and pruned on CUB.

PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	51.59 ± 0.0151		
30	$46.43\ 5.16\downarrow \pm 0.0134$	$44.00\ 7.59\downarrow \pm 0.0213$	$51.02\ 0.57\downarrow \pm 0.0231$
50	$40.98\ 10.61\downarrow \pm 0.0250$	$39.04\ 12.55\downarrow \pm 0.0115$	$50.67\ 0.92\downarrow \pm 0.0144$
80	$34.50\ 17.09\downarrow \pm 0.0112$	$33.00\ 18.59\downarrow \pm 0.0203$	$50.61\ 0.98\downarrow \pm 0.0070$
85	$32.00\ 19.59\downarrow \pm 0.0351$	$30.09\ 21.50\downarrow \pm 0.0195$	$50.60\ 0.99\downarrow \pm 0.0175$

Magnitude or L-OBS, is biased to the pruning target task, which leads to sub-optimal performance of fast adaptation. To illustrate this bias, we randomly selected a 5-way "target task" from the Mini-ImageNet dataset, then train & prune via both baseline methods (Magnitude and L-OBS) ConvNet-4 backbones up to a compression ratio of 85%. These compressed models are then tested for few-shot accuracy on not only the aforementioned "target task", but also the so-called "other tasks", which are tasks again randomly selected from the dataset. Note that none of the "other tasks" has been used during the pruning. Fig. 5.8 shows the training accuracy curves on the "target task" and "other



(a) Mini-ImageNet **(b)** CUB

Figure 5.6 **5-way 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **CAVIA** and pruned by Magnitude, L-OBS, or ANP on (a) Mini-ImageNet and (b) CUB.

Table 5.8 **5-way, 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **CAVIA** and pruned on Mini-ImageNet.

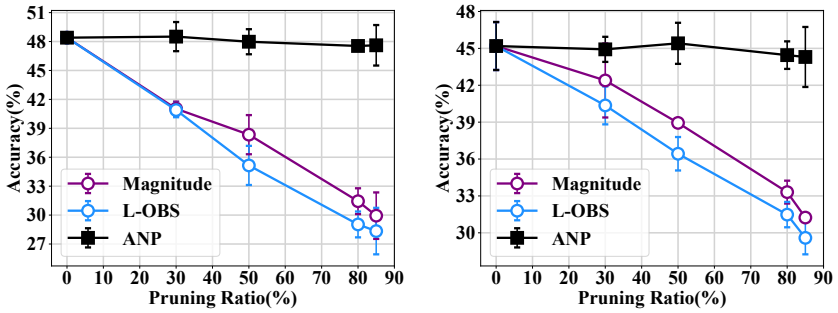
PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	47.56 ± 0.0076		
30	$38.71\ 8.85\downarrow \pm 0.0145$	$37.86\ 9.70\downarrow \pm 0.0148$	$46.74\ 0.82\downarrow \pm 0.0140$
50	$34.87\ 12.69\downarrow \pm 0.0072$	$34.32\ 13.24\downarrow \pm 0.0118$	$47.71\ 0.15\uparrow \pm 0.0169$
80	$32.14\ 15.42\downarrow \pm 0.0060$	$23.86\ 23.70\downarrow \pm 0.0100$	$46.59\ 0.97\downarrow \pm 0.0103$
85	$27.76\ 19.80\downarrow \pm 0.0156$	$20.86\ 26.70\downarrow \pm 0.0099$	$46.57\ 0.99\downarrow \pm 0.0223$

Table 5.9 **5-way, 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **CAVIA** and pruned on CUB.

PR (%)	CUB Accuracy (%)		
	Magnitude	L-OBS	ANP
0	47.32 ± 0.0235		
30	$41.77\ 5.55\downarrow \pm 0.0176$	$40.67\ 6.65\downarrow \pm 0.0173$	$47.15\ 0.17\downarrow \pm 0.0239$
50	$41.39\ 5.93\downarrow \pm 0.0196$	$40.24\ 7.08\downarrow \pm 0.0211$	$47.03\ 0.29\downarrow \pm 0.0269$
80	$38.33\ 8.99\downarrow \pm 0.0225$	$37.28\ 10.04\downarrow \pm 0.0237$	$46.73\ 0.59\downarrow \pm 0.0163$
85	$37.91\ 9.41\downarrow \pm 0.0183$	$37.26\ 10.05\downarrow \pm 0.0152$	$46.85\ 0.47\downarrow \pm 0.0185$

tasks" when performing 5-way, 1-shot learning tests. The few-shot accuracy results on the multiple "other tasks" are aggregated into one line with error bars. As we can see, the accuracy on the pruning "target task" is high, but on "other tasks", which may notably differ from the "target task", the accuracy is low.

Few-Shot Learning: Convergence. Fig. 5.9 plots the few-shot training loss and few-shot accuracy curves of a meta-trained & pruned initial architecture to perform few-shot learning on test tasks. ANP provides not only a better accuracy, but also a faster convergence. This is a crucial benefit for resource-



(a) Mini-ImageNet **(b)** CUB

Figure 5.7 **5-way 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **Reptile** and pruned by Magnitude, L-OBS, or ANP on (a) Mini-ImageNet and (b) CUB.

Table 5.10 **5-way, 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **Reptile** and pruned on Mini-ImageNet.

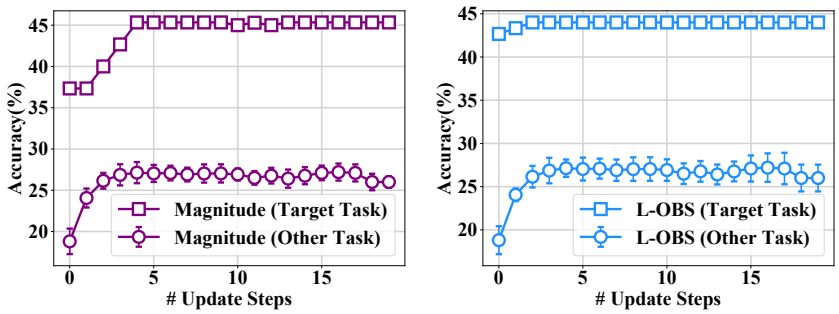
PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	48.40 ± 0.0051		
30	$41.03\ 7.37\downarrow \pm 0.0074$	$40.89\ 7.51\downarrow \pm 0.0213$	$48.51\ 0.11\uparrow \pm 0.0151$
50	$38.34\ 10.06\downarrow \pm 0.0203$	$35.15\ 13.25\downarrow \pm 0.0084$	$47.98\ 0.42\downarrow \pm 0.0130$
80	$31.55\ 16.85\downarrow \pm 0.0134$	$29.03\ 19.37\downarrow \pm 0.0093$	$47.54\ 0.86\downarrow \pm 0.0034$
85	$29.94\ 18.46\downarrow \pm 0.0241$	$28.34\ 20.06\downarrow \pm 0.0124$	$47.61\ 0.79\downarrow \pm 0.0210$

Table 5.11 **5-way, 1-shot** accuracy vs. PR of **ConvNet-4** meta-trained by **Reptile** and pruned on CUB.

PR (%)	Accuracy (%)		
	Magnitude	L-OBS	ANP
0	45.19 ± 0.0195		
30	$42.39\ 2.8\downarrow \pm 0.0301$	$40.36\ 4.83\downarrow \pm 0.0154$	$44.92\ 0.27\downarrow \pm 0.0102$
50	$38.94\ 6.25\downarrow \pm 0.0013$	$36.43\ 8.76\downarrow \pm 0.0136$	$45.41\ 0.22\uparrow \pm 0.0167$
80	$33.31\ 11.88\downarrow \pm 0.0093$	$31.48\ 13.71\downarrow \pm 0.0103$	$44.45\ 0.74\downarrow \pm 0.0112$
85	$31.24\ 13.95\downarrow \pm 0.0035$	$29.59\ 15.60\downarrow \pm 0.0134$	$44.30\ 0.89\downarrow \pm 0.0244$

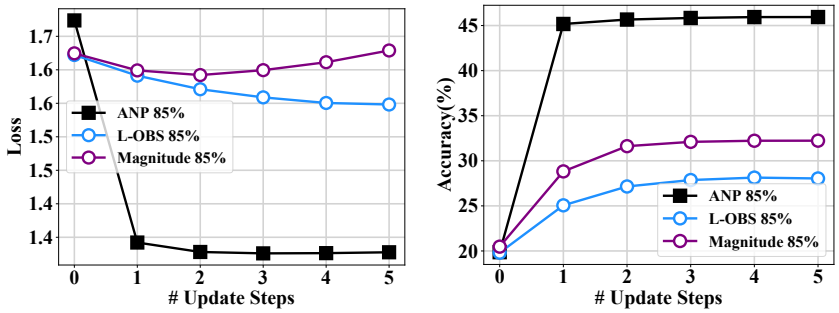
constrained devices, as with fewer training steps the computation cost and power consumption can be reduced during adaptation. Moreover, in some experiments (*e.g.*, the loss curve for Magnitude in Fig. 5.9(a)), the training loss curve of the baseline methods may even not converge. In contrast, ANP provides consistent and stable convergence.

Fig. 5.10, Fig. 5.11 and Fig. 5.12 show the loss and accuracy curves of a meta-trained & pruned ConvNet-4 to perform 5-way, 1-shot learning on Mini-ImageNet. The initial architecture is compressed to a pruning ratio of 30%, 50%, and 80%, respectively. The accuracy advantage of ANP against baselines



(a) Magnitude **(b)** L-OBS

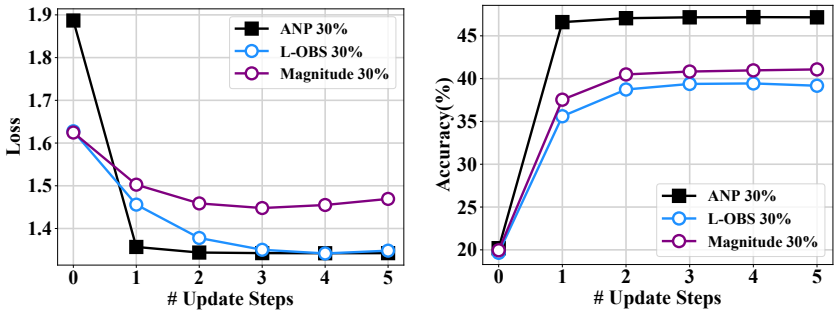
Figure 5.8 5-way 1-shot training accuracy curves on the task selected for pruning and other tasks of ConvNet-4 meta-trained by MAML and pruned with (a) Magnitude and (b) L-OBS on Mini-ImageNet.



(a) Loss Curves **(b)** Accuracy Curves

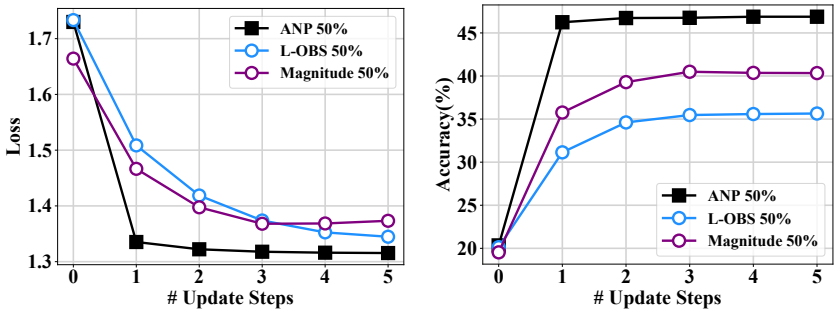
Figure 5.9 5-way 1-shot (a) **training** loss and (b) **testing** accuracy curve of ConvNet-4 meta-trained by MAML and pruned by Magnitude, L-OBS, or ANP on Mini-ImageNet (**PR = 85%**).

increases as the PR increases as expected. Furthermore, the faster and more stable convergence with ANP compared to the baselines, which is observed on PR $= 85\%$, can also be observed with PR $= 30\%$, 50%, and 80%.



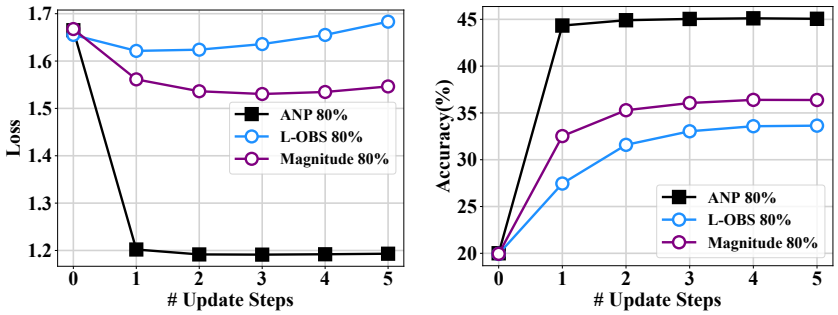
(a) Loss Curves **(b)** Accuracy Curves

Figure 5.10 5-way 1-shot (a) **training** loss and (b) **testing** accuracy curve of ConvNet-4 meta-trained by MAML and pruned by Magnitude, L-OBS, or ANP on Mini-ImageNet (**PR** $= 30\%$).



(a) Loss Curves **(b)** Accuracy Curves

Figure 5.11 5-way 1-shot (a) **training** loss and (b) **testing** accuracy curve of ConvNet-4 meta-trained by MAML and pruned by Magnitude, L-OBS, or ANP on Mini-ImageNet (**PR** $= 50\%$).



(a) Loss Curves **(b)** Accuracy Curves

Figure 5.12 5-way 1-shot (a) **training** loss and (b) **testing** accuracy curve of ConvNet-4 meta-trained by MAML and pruned by Magnitude, L-OBS, or ANP on Mini-ImageNet (**PR** $= 80\%$).

5.5 Conclusion

In this chapter, we investigated the on-device adaptation problem discussed in Sec. 1.5. We explored pruning *meta-trained* deep neural networks for few-shot learning on resource-constrained platforms. Prior pruning methods deteriorate the fast adaptability of meta-trained networks due to inconsistent weight importance metrics with the meta-objective. In response, we propose ANP, the first meta-training-compatible network pruning scheme. ANP defines a weight importance metric for the meta-objective to find a topology meta-optimised for learning new tasks in a few shots. Evaluations on few-shot classification benchmarks show that ANP can prune MAML-like meta-trained convolutional and residual backbones by 85% with a minimal drop in few-shot classification accuracy. We envision our work will offer guidelines to enable fast model adaptation on low-resource platforms.

6

Conclusions and Outlook

The deployment of artificial neural network enabled deep learning models on edge devices holds the potential for bringing machine intelligence to our everyday life. However, modern DL models are often compute-intensive and have significant memory requirements, thus compression and optimisation of these DL models are necessary for on-device deployment. In this dissertation, we have presented advances in this area from two perspectives: the multi-model compression for efficient inference of MMDL systems, and the efficient on-device adaptation.

AI-powered mobile applications increasingly demand multiple deep neural networks for correlated tasks to be performed continuously and concurrently on resource-constrained devices. In this dissertation, we showed that multi-modal compression is the key to utilising task relatedness and consequently compressing the footprint of MMDL systems. We distinguish between three types of MMDL systems and provide solutions for all of them.

On-device adaptation refers to learning previously unseen tasks by updating an initial model onboard, which is desired in many on-device intelligence applications including personal drones, home robots and self-driving vehicles, as uploading newly collected data for model updating can be infeasible due to unstable wireless connections, limited bandwidth or privacy concerns. Previous works provided solutions for adaption with a limited amount of training data, but the provided DL models are still over-parameterised and compute-intensive. In this dissertation, we provided a solution that enables the construction of a compact model capable of fast on-device adaption.

6.1 Contributions

A weight sharing based network merging framework (Chapter 2). We proposed Multi-Task Zipping (MTZ), a framework to automatically merge correlated, pre-trained deep neural networks in the first and second types of MMDL system, discussed in Sec. 1.4.1 and Sec. 1.4.2. Central in MTZ is a layer-wise neuron sharing and incoming weight updating scheme that induces a minimal change in the error function. MTZ inherits information from each model and demands light retraining to re-boost the accuracy of individual tasks. MTZ supports typical network layers (fully-connected, convolutional and residual) and applies to inference tasks with different input domains. Evaluations show that MTZ can fully merge the hidden layers of two VGG-16 networks with a 3.18% increase in the test error averaged on ImageNet for object classification and CelebA for facial attribute classification, or share 39.61% parameters between the two networks with $< 0.5\%$ increase in the test errors. The number of iterations to retrain the combined network is at least $17.8\times$ lower than that of training a single VGG-16 network. Moreover, MTZ can effectively merge nine residual networks for diverse inference tasks and models for different input domains. And with the model merged by MTZ, the latency to switch between these tasks on memory-constrained devices is reduced by $8.71\times$.

A DNN graph rewriter for efficient execution of networks merged via our Multi-Task Zipping framework (Chapter 3). We designed Multi-Task Stitching (MTS), a novel graph rewriter for efficient multitask inference with weight-shared DNNs, such as those merged via our MTZ. MTS adopts a model stitching algorithm which outputs a single computational graph for weight-shared DNNs without duplicating any shared weight. MTS also utilises a model grouping strategy to avoid overwhelming the GPU when co-running tens of DNNs. Extensive experiments show that MTS accelerates multitask inference by up to $6.0\times$ compared to sequentially executing multiple weight-shared DNNs. MTS also yields up to $2.5\times$ lower latency and $3.7\times$ less memory usage compared with NETFUSE, a state-of-the-art multi-DNN graph rewriter.

A neuron merging based network merging scheme (Chapter 4). With the help of information theory, we formally defined the redundancy within the third type of MMDL system introduced in Sec. 1.4.3 and identify the optimal topology for merging. We also theoretically identified the conditions such that the merged network can be effectively pruned via existing pruning schemes and the computation of all task combinations can be minimised, which is often demanded by modern mobile applications. On this basis, we proposed Pruning-Aware Merging (PAM), a heuristic network merging scheme to construct a multitask network that approximates these conditions. The merged network is then ready to be further pruned via existing network pruning methods. Evaluations with different pruning schemes, datasets, and network architectures show that PAM achieves up to $4.87\times$ less computation against the baseline

without network merging and up to $2.01\times$ less computation against the baseline with a state-of-the-art network merging scheme.

A novel pruning scheme that works with existing meta-learning methods for on-device adaptation (Chapter 5). For on-device adaptation, we proposed Adaptation-aware Network Pruning (ANP), a novel pruning scheme that works with existing meta-learning methods for a compact network capable of fast adaptation. ANP uses a weight importance metric that is based on the sensitivity of the meta-objective rather than the conventional loss function and adopts approximation of derivatives and layer-wise pruning techniques to reduce the overhead of computing the new importance metric. Evaluations on few-shot classification benchmarks show that ANP can prune meta-trained convolutional and residual networks by 85% without affecting their fast adaptation.

6.2 Future Developments

Towards Artificial General Intelligence. In Sec. 1.4, we introduced three types of MMDL system. On this basis, we can, in practice, construct another type of MMDL system by connecting the output of a second type (Sec. 1.4.2) to the input of a third type (Sec. 1.4.3). Such an MMDL system takes inputs from different data sources, combines the information, and uses them effectively for different tasks. Many believe this kind of unified DL system is the key towards *Artificial General Intelligence (AGI)*, an advance of AI approaching the level of intelligence shown by humans. Currently, no research work has investigated such AGI systems' on-device deployment to the best of our knowledge.

Common toolset for the on-device deployment of DL systems. As far as we know, most state-of-the-art frameworks for on-device deployment, including TVM [18] and TensorFlow Lite [77], do not support automatically applying model-compression techniques like network pruning, let alone our multi-model compression methods. Therefore, a comprehensive toolset providing an automatic end-to-end solution for the deployment of DL models would be interesting for both research and commercial purposes.

More efficient on-device learning. In Chapter 5, we provided a meta-pruning method for reducing the memory and computation during on-device adaptation. Besides using model compression techniques like network pruning, there are also other potential solutions to further improve the efficiency of on-device learning. On the one hand, the stochastic gradient descent (SGD) method is the primary cause of the large memory consumption for neural network training. Zero-order optimisation is a potential alternative to the SGD method for NN training, which may improve training efficiency under certain circumstances. On the other hand, distributed learning paradigms like federated learning [134] can mitigate the lack of data during local training.

7

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

X. He, X. Wang, Z. Zhou, J. Wu, Z.Yang, L. Thiele. **On-Device Deep Multi-Task Inference via Multi-Task Zipping** *IEEE Transactions on Mobile Computing (TMC)* (Chapter 2)

X. He, Z. Zhou, L. Thiele. **Multi-Task Zipping via Layer-wise Neuron Sharing** *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)* (Chapter 2)

Z.Wang, X. He, Z. Zhou, X. Wang, Q. Ma, X. Miao, L. Thiele, Z. Yang. **Stitching Weight-Shared Deep Neural Networks for Efficient Multitask Inference on GPU** *In Submission* (Chapter 3)

X. He, D. Gao, Z. Zhou, Y. Tong, L. Thiele. **Pruning-Aware Merging for Efficient Multitask Inference** *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Chapter 4)

D. Gao*, X. He*, Z. Zhou, Y. Tong, L. Thiele. **Pruning Meta-Trained Networks for On-Device Adaptation** *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM)* (Chapter 5)

The following list includes publications that were written during the PhD studies, yet are not part of this thesis.

Y. Cheng, X. He, Z. Zhou, L. Thiele. **ICT: In-field calibration transfer for air quality sensor deployments.** *In Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT), 3(1), 1-19.*

Y. Cheng, X. He, Z. Zhou, L. Thiele. **MapTransfer: Urban air quality map generation for downscaled sensor deployments.** *In 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI) (pp. 14-26).*

D. Gao, X. He, Z. Zhou, Y. Tong, K.Xu, L. Thiele. **Rethinking Pruning for Accelerating Deep Inference At the Edge** *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*

W. Duan*, X. He*, Z. Zhou, H. Rao, L. Thiele **Injecting Descriptive Meta-information into Pre-trained Language Models with Hypernetworks** *Proceedings of Interspeech 2021*

S. Liu, K. Koch, Z. Zhou, S. Föll, X. He, T. Menke, E. Fleisch **The EmpatheticCar: Exploring Emotion Inference via Driver Behaviour and Traffic Context** *In Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, vol.5, no.3, pp.117:1-117:34, 2021.*

S. Liu, K. Koch, Z. Zhou, M. Maritsch, X. He, E. Fleisch, F. Wortmann **Towards Non-Intrusive Camera-Based Heart Rate Variability Estimation in the Car under Naturalistic Condition** *IEEE Internet of Things Journal (IoTJ)*

Bibliography

[1] S. Pouyanfar, S. Sadiq, Y. Yan, *et al.*, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018, Publisher: ACM New York, NY, USA.

[2] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, _eprint: 1409.1556, 2014.

[3] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, event-place: Boston, MA, USA: USENIX Association, 2010, p. 21.

[4] R. Prabhavalkar, K. Rao, T. N. Sainath, B. Li, L. Johnson, and N. Jaitly, "A comparison of sequence-to-sequence models for speech recognition," in *Proc. Interspeech 2017*, 2017, pp. 939–943. DOI: 10.21437/Interspeech.2017-233.

[5] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17, event-place: Niagara Falls, New York, USA, New York, NY, USA: Association for Computing Machinery, 2017, pp. 68–81, ISBN: 978-1-4503-4928-4. DOI: 10.1145/3081333.3081359. [Online]. Available: https://doi.org/10.1145/3081333.3081359.

[6] S. Lee and S. Nirjon, "Fast and scalable in-memory deep multitask learning via neural weight virtualization," in *Proceedings of ACM Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA: ACM, 2020, pp. 175–190.

[7] N. A. Sulieman, L. Ricciardi Celsi, W. Li, A. Zomaya, and M. Villari, "Edge-oriented computing: A survey on research and use cases," *Energies*, vol. 15, no. 2, 2022, ISSN: 1996-1073. DOI: 10.3390/en15020452. [Online]. Available: https://www.mdpi.com/1996-1073/15/2/452.

[8] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020, Publisher: IEEE.

[9] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=rJl-b3RcF7.

[10] B. Dai, C. Zhu, B. Guo, and D. Wipf, "Compressing neural networks using the variational information bottleneck," in *Proceedings of International Conference on Machine Learning*, New York, NY, USA: ACM, 2018, pp. 1135–1144.

[11] X. Dong, S. Chen, and S. J. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, event-place: Long Beach, California, USA, Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 4860–4874, ISBN: 978-1-5108-6096-4.

[12] B. Hassibi and D. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles, Eds., vol. 5, Morgan-Kaufmann, 1992. [Online]. Available: https://proceedings.neurips.cc/paper/1992/file/303ed4c69846ab36c2904d3ba8573050-Paper.pdf.

[13] S. Liu, B. Guo, K. Ma, Z. Yu, and J. Du, "AdaSpring: Context-adaptive and runtime-evolutionary deep model compression for mobile applications," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 5, no. 1, 24:1–24:22, 2021.

[14] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, Jun. 1, 2021, ISSN: 1573-1405. DOI: 10.1007/s11263-021-01453-z. [Online]. Available: https://doi.org/10.1007/s11263-021-01453-z.

[15] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[16] Z. Qu, Z. Zhou, Y. Cheng, and L. Thiele, "Adaptive loss-aware quantization for multi-bit networks," in *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 7988–7997.

[17] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, *A survey of quantization methods for efficient neural network inference*, 2021. DOI: 10.48550/ARXIV.2103.13630. [Online]. Available: https://arxiv.org/abs/2103.13630.

[18] T. Chen, T. Moreau, Z. Jiang, *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA: USENIX, 2018, pp. 578–594.

[19] P. Georgiev, S. Bhattacharya, N. D. Lane, and C. Mascolo, "Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, 50:1–50:19, 2017.

[20] Y.-M. Chou, Y.-M. Chan, J.-H. Lee, C.-Y. Chiu, and C.-S. Chen, "Unifying and merging well-trained deep neural networks for inference stage," in *Proceedings of International Joint Conference on Artificial Intelligence*, Burlington, MA, USA: Morgan Kaufmann, 2018, pp. 2049–2056.

[21] B. Fang, X. Zeng, and M. Zhang, "NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of ACM Annual International Conference on Mobile Computing and Networking*, New York, NY, USA: ACM, 2018, pp. 115–127.

[22] C.-E. Wu, J.-H. Lee, T. S. Wan, Y.-M. Chan, and C.-S. Chen, "Merging well-trained deep cnn models for efficient inference," in *Proceedings of IEEE Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1594–1600.

[23] S.-A. Rebuffi, H. Bilen, and A. Vedaldi, "Learning multiple visual domains with residual adapters," in *Advances in Neural Information Processing Systems*, 2017, pp. 506–516.

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] R. Rothe, R. Timofte, and L. Van Gool, "Deep expectation of real and apparent age from a single image without facial landmarks," *International Journal of Computer Vision*, vol. 126, no. 2, pp. 144–157, 2018, Publisher: Springer.

[26] Y. Lu, A. Kumar, S. Zhai, Y. Cheng, T. Javidi, and R. Feris, "Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification," in *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5334–5343.

[27] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" In *Advances in Neural Information Processing Systems*, 2014, pp. 3320–3328.

[28] S. Zhang, S. Zhang, T. Huang, and W. Gao, "Multimodal deep convolutional neural network for audio-visual emotion recognition," in *Proceedings of ACM on International Conference on Multimedia Retrieval*, New York, NY, USA: ACM, 2016, pp. 281–284.

[29] X. He, Z. Zhou, and L. Thiele, "Multi-task zipping via layer-wise neuron sharing," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, event-place: Montreal, Canada, Red Hook, NY, USA: Curran Associates Inc., 2018, pp. 6019–6029.

[30] Y. Wang and L. Guan, "Recognizing human emotional state from audiovisual signals," *IEEE Transactions on Multimedia*, vol. 10, no. 5, pp. 936–946, 2008.

[31] J. Zhao, X. Xie, X. Xu, and S. Sun, "Multi-view learning overview: Recent progress and new challenges," *Information Fusion*, vol. 38, pp. 43–54, 2017, ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2017.02.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253516302032.

[32] J. S. Jeong, S. Kim, G.-I. Yu, Y. Lee, and B.-G. Chun, "Accelerating multi-model inference by merging DNNs of different weights," *arXiv preprint arXiv:2009.13062*, 2020.

[33] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proceedings of International Conference on Machine Learning*, New York, NY, USA: ACM, 2017, pp. 1126–1135.

[34] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," in *Proceedings of International Conference on Learning Representations*, 2017.

[35] S. Liu, J. Du, K. Nan, *et al.*, "AdaDeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2020.

[36] Y. Zhang and Q. Yang, "An overview of multi-task learning," *National Science Review*, vol. 5, no. 1, pp. 30–43, 2018, Publisher: Oxford University Press.

[37] S. Ruder, *An overview of multi-task learning in deep neural networks*, _eprint: 1706.05098, 2017.

[38] Y. Yang and T. Hospedales, "Deep multi-task representation learning: A tensor factorisation approach," in *Proceedings of International Conference on Learning Representations*, 2016.

[39] X. Sun, R. Panda, R. Feris, and K. Saenko, "Adashare: Learning what to share for efficient deep multi-task learning," in *Advances in Neural Information Processing Systems*, 2020, pp. 8728–8740.

[40] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015, Publisher: Springer.

[41] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of IEEE International Conference on Computer Vision*, 2015, pp. 3730–3738.

[42] H. Bilen and A. Vedaldi, "Integrated perception with recurrent multi-task neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 235–243.

[43] T. Standley, A. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese, "Which tasks should be learned together in multi-task learning?" In *Proceedings of ACM International Conference on Machine Learning*, 2020, pp. 9120–9132.

[44] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert, "Cross-stitch networks for multi-task learning," in *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016, pp. 3994–4003.

[45] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 264–11 272.

[46] N. D. Lane, S. Bhattacharya, P. Georgiev, *et al.*, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of ACM/IEEE International Conference on Information Processing in Sensor Networks*, New York, NY, USA: ACM, 2016, pp. 1–12.

[47] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[48] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *Proceedings of IEEE International Conference on Computer Communications*, 2018, pp. 1421–1429.

[49] Y. Kang, J. Hauswald, C. Gao, *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: ACM, 2017, pp. 615–629.

[50] X. Wang, Z. Yang, J. Wu, Y. Zhao, and Z. Zhou, "Edgeduet: Tiling small object detection for edge assisted autonomous mobile vision," in *Proceedings of IEEE International Conference on Computer Communications*, 2021, pp. 1–1.

[51] J. Yi, S. Choi, and Y. Lee, "EagleEye: Wearable camera-based person identification in crowded urban spaces," in *Proceedings of ACM Annual International Conference on Mobile Computing and Networking*, New York, NY, USA: ACM, 2020, pp. 1–14.

[52] P. K. Deb, S. Misra, T. Sarkar, and A. Mukherjee, "Magnum: A distributed framework for enabling transfer learning in b5g-enabled industrial IoT," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 10, pp. 7133–7140, 2021. DOI: 10.1109/TII.2020.3047206.

[53] M. Chao, R. Stoleru, L. Jin, S. Yao, M. Maurice, and R. Blalock, "AMVP: Adaptive CNN-based multitask video processing on mobile stream processing platforms," in *Proceedings of IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 96–109.

[54] Z. Ouyang, J. Niu, Y. Liu, and M. Guizani, "Deep CNN-based real-time traffic light detector for self-driving vehicles," *IEEE Transactions on Mobile Computing*, vol. 19, no. 2, pp. 300–313, 2020, Publisher: IEEE.

[55] C. Wang, Y. Xiao, X. Gao, L. Li, and J. Wang, "A framework for behavioral biometric authentication using deep metric learning on mobile devices," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[56] Y. Zhang, T. Gu, and X. Zhang, "MDLdroidLite: A release-and-inhibit control approach to resource-efficient deep neural networks on mobile devices," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[57] NVIDIA, *Jetson nano developer kit*, 2020. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[58] S. Zagoruyko and N. Komodakis, *Wide residual networks*, _eprint: 1605.07146, 2016.

[59] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, vol. 32, pp. 323–332, 2012, Publisher: Elsevier.

[60] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, Issue: 2, vol. 2011, 2011, p. 5.

[61] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, 2009.

[62] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[63] K. Soomro, A. R. Zamir, and M. Shah, *UCF101: A dataset of 101 human actions classes from videos in the wild*, _eprint: 1212.0402, 2012.

[64] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Proccesings of IEEE Indian Conference on Computer Vision, Graphics & Image Processing*, 2008, pp. 722–729.

[65] S. Munder and D. M. Gavrila, "An experimental study on pedestrian classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1863–1868, 2006.

[66] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, and A. Vedaldi, "Describing textures in the wild," in *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2014, pp. 3606–3613.

[67] S. Maji, E. Rahtu, J. Kannala, M. Blaschko, and A. Vedaldi, *Fine-grained visual classification of aircraft*, _eprint: 1306.5151, 2013.

[68] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *Proceedings of IEEE Real-Time Systems Symposium*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 392–405.

[69] J. Yi and Y. Lee, "Heimdall: Mobile GPU coordination platform for augmented reality applications," in *ACM Annual International Conference on Mobile Computing and Networking*, New York, NY, USA: ACM, 2020, pp. 1–14.

[70] F. Yu, S. Bray, D. Wang, *et al.*, "Automated runtime-aware scheduling for multi-tenant dnn inference on gpu," in *Proceedings of IEEE/ACM International Conference On Computer Aided Design*, Piscataway, NJ, USA: IEEE Press, 2021.

[71] B. Cox, J. Galjaard, A. Ghiassi, R. Birke, and L. Y. Chen, "Masa: Responsive multi-dnn inference on the edge," in *Proceedings of IEEE International Conference on Pervasive Computing and Communications*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1–10.

[72] X. He, X. Wang, Z. Zhou, J. Wu, Z. Yang, and L. Thiele, "On-device deep multi-task inference via multi-task zipping," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021. DOI: 10.1109/TMC.2021.3124306.

[73] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proceedings of International Conference on Learning Representations*, 2016.

[74] NVIDIA, *NVIDIA TensorRT*. [Online]. Available: https://developer.nvidia.com/tensorrt.

[75] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: ACM, 2013, pp. 395–406.

[76] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proceedings of IEEE Real-Time Systems Symposium*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 104–115.

[77] M. Abadi, P. Barham, J. Chen, *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA: USENIX, 2016, pp. 265–283.

[78] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2019, pp. 8026–8037.

[79] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of ACM Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA: ACM, 2018, pp. 389–400.

[80] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proceedings of ACM Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA: ACM, 2017, pp. 82–95.

[81] S. Zhang, Y. Li, X. Liu, *et al.*, "Towards real-time cooperative deep inference over the cloud and edge end devices," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 4, no. 2, 69:1–69:24, 2020.

[82] T. S. T. Wan, J.-H. Lee, Y.-M. Chan, and C.-S. Chen, "Co-compressing and unifying deep CNN models for efficient human face and speaker recognition," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2019, pp. 461–468. DOI: 10.1109/CVPRW.2019.00060.

[83] NVIDIA, *NVIDIA CUDA streams*. [Online]. Available: https://developer.download.nvidia.com/CUDA%20/training/StreamsAndConcurrencyWebinar.pdf.

[84] ——, *NVIDIA multi-process service*. [Online]. Available: https://docs.nvidia.com/deploy/pdf%20/CUDA_Multi_Process_Service_Overview.pdf.

[85] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia, "Accelerating deep learning workloads through efficient multi-model execution," in *NeurIPS Workshop on Systems for Machine Learning*, 2018.

[86] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel gpu task scheduling for deep learning," in *Advances in Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2020.

[87] J. Lee, Y. Liu, and Y. Lee, "ParallelFusion: Towards maximum utilization of mobile gpu for dnn inference," in *Proceedings of the International Workshop on Embedded and Mobile Deep Learning*, New York, NY, USA: ACM, 2021, pp. 25–30.

[88] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *Proceedings of Conference on Machine Learning and Systems*, 2019, pp. 1–13.

[89] P. Jain, X. Mo, A. Jain, *et al.*, "Dynamic space-time scheduling for gpu inference," *arXiv preprint arXiv:1901.00041*, 2018.

[90] S. Jiang, L. Ran, T. Cao, Y. Xu, and Y. Liu, "Profiling and optimizing deep learning inference on mobile GPUs," in *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems*, New York, NY, USA: ACM, 2020, pp. 75–81.

[91] Z. Ji, "ILP-m conv: Optimize convolution algorithm for single-image convolution neural network inference on mobile GPUs," *arXiv preprint arXiv:1909.02765*, 2019.

[92] L. L. Zhang, S. Han, J. Wei, *et al.*, "Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 81–93.

[93] NVIDIA, *NVIDIA nsight systems*. [Online]. Available: https://developer.nvidia.com/nsight-systems.

[94] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in *Proceedings of ACM International Conference on Machine Learning*, New York, NY, USA: ACM, 2017, pp. 527–536.

[95] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable neural networks," in *Proceedings of International Conference on Learning Representations*, 2019.

[96] X. He, D. Gao, Z. Zhou, Y. Tong, and L. Thiele, "Pruning-aware merging for efficient multitask inference," in *Proceedings of ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, New York, NY, USA: ACM, 2021, pp. 585–595.

[97] M. Denil, B. Shakibi, L. Dinh, N. De Freitas, *et al.*, "Predicting parameters in deep learning," in *Proceedings of Advances In Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2013, pp. 2148–2156.

[98] D. Gao, X. He, Z. Zhou, Y. Tong, K. Xu, and L. Thiele, "Rethinking pruning for accelerating deep inference at the edge," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, event-place: Virtual Event, CA, USA, New York, NY, USA: Association for Computing Machinery, 2020, pp. 155–164, ISBN: 978-1-4503-7998-4. DOI: 10.1145/3394486.3403058. [Online]. Available: https://doi.org/10.1145/3394486.3403058.

[99] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[100] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances in Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2016, pp. 3104–3112.

[101] S. Han, X. Liu, H. Mao, *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, event-place: Seoul, Republic of Korea, IEEE Press, 2016, pp. 243–254, ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.30. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30.

[102] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *Proceedings of International Conference on Learning Representations*, 2017.

[103] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2016, pp. 2082–2090.

[104] A. M. Saxe, Y. Bansal, J. Dapello, *et al.*, "On the information bottleneck theory of deep learning," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=ry_WPG-A-.

[105] N. Tishby and N. Zaslavsky, "Deep learning and the information bottleneck principle," in *2015 IEEE Information Theory Workshop (ITW)*, 2015, pp. 1–5. DOI: 10.1109/ITW.2015.7133169.

[106] A. J. Bell, "The co-information lattice," in *Proceedings of the Fifth International Workshop on Independent Component Analysis and Blind Signal Separation: ICA*, vol. 2003, Citeseer, 2003.

[107] A. Kolchinsky and B. D. Tracey, "Estimating mixture entropy with pairwise distances," *Entropy*, vol. 19, no. 7, 2017, ISSN: 1099-4300. DOI: 10.3390/e19070361. [Online]. Available: https://www.mdpi.com/1099-4300/19/7/361.

[108] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[109] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms*, 2017. DOI: 10.48550/ARXIV.1708.07747. [Online]. Available: https://arxiv.org/abs/1708.07747.

[110] G. Huang, M. Mattar, H. Lee, and E. Learned-miller, "Learning to align from scratch," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/d81f9c1be2e08964bf9f24b15f0e4900-Paper.pdf.

[111] N. Kumar, A. C. Berg, P. N. Belhumeur, and S. K. Nayar, "Attribute and simile classifiers for face verification," in *2009 IEEE 12th International Conference on Computer Vision*, 2009, pp. 365–372. DOI: 10.1109/ICCV.2009.5459250.

[112] C. Banbury, C. Zhou, I. Fedorov, *et al.*, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[113] A. Nichol, J. Achiam, and J. Schulman, *On first-order meta-learning algorithms*, _eprint: 1803.02999, 2018.

[114] L. Zintgraf, K. Shiarli, V. Kurin, K. Hofmann, and S. Whiteson, "Fast context adaptation via meta-learning," in *Proceedings of International Conference on Machine Learning*, New York, NY, USA: ACM, 2019, pp. 7693–7702.

[115] M. Gooneratne, K. C. Sim, P. Zadrazil, A. Kabel, F. Beaufays, and G. Motta, "Low-rank gradient approximation for memory-efficient on-device training of deep neural network," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 3017–3021.

[116] J. Liu, J. Liu, W. Du, and D. Li, "Performance analysis and characterization of training deep learning models on mobile device," in *Proceedings of International Conference on Parallel and Distributed Systems*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 506–515.

[117] T. Elsken, B. Staffler, J. H. Metzen, and F. Hutter, "Meta-learning of neural architectures for few-shot learning," in *Proceedings of Conference on Computer Vision and Pattern Recognition*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 12 365–12 375.

[118] S. Arnold, S. Iqbal, and F. Sha, "When MAML can adapt fast and how to assist when it cannot," in *Proceedings of International Conference on Artificial Intelligence and Statistics*, PMLR, 2021, pp. 244–252.

[119] T. Lin, S. U. Stich, L. Barba, D. Dmitriev, and M. Jaggi, "Dynamic model pruning with feedback," in *Proceedings of International Conference on Learning Representations*, 2020.

[120] M. Zhu and S. Gupta, *To prune, or not to prune: Exploring the efficacy of pruning for model compression*, _eprint: 1710.01878, 2017.

[121] Y. Le Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 1990, pp. 598–605.

[122] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The caltech-ucsd birds-200-2011 dataset," California Institute of Technology, 2011.

[123] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, 2. MIT press Cambridge, 2016, vol. 1.

[124] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–34, 2020.

[125] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, *Meta-learning in neural networks: A survey*, _eprint: 2004.05439, 2020.

[126] D. Lian, Y. Zheng, Y. Xu, *et al.*, "Towards fast adaptation of neural architectures with meta learning," in *Proceedings of International Conference on Learning Representations*, 2020.

[127] Z. Liu, H. Mu, X. Zhang, *et al.*, "Metapruning: Meta learning for automatic neural network channel pruning," in *Proceedings of International Conference on Computer Vision*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 3296–3305.

[128] H. Tian, B. Liu, X.-T. Yuan, and Q. Liu, "Meta-learning with network pruning," in *Proceedings of European Conference on Computer Vision*, Berlin, Germany: Springer, 2020, pp. 675–700.

[129] M. Hardy, "Combinatorics of partial derivatives," *The Electronic Journal of Combinatorics*, vol. 13, p. 1, R1 2006.

[130] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, _eprint: 1412.6572, 2014.

[131] T. Kailath, *Linear Systems*. Prentice-Hall Englewood Cliffs, NJ, 1980, vol. 156.

[132] W.-Y. Chen, Y.-C. Liu, Z. Kira, Y.-C. F. Wang, and J.-B. Huang, "A closer look at few-shot classification," in *Proceedings of International Conference on Learning Representations*, 2019.

[133] L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil, "Bilevel programming for hyperparameter optimization and meta-learning," in *Proceedings of International Conference on Machine Learning*, New York, NY, USA: ACM, 2018, pp. 1568–1577.

[134] J. Konecny, H. B. McMahan, F. X. Yu, P. Richtarik, A. T. Suresh, and D. Bacon, *Federated learning: Strategies for improving communication efficiency*, 2016. DOI: 10.48550/ARXIV.1610.05492. [Online]. Available: https://arxiv.org/abs/1610.05492.