



Universidade Estadual de Campinas
Instituto de Computação



Vanderson Martins do Rosario

Accelerating capsule networks with lanes
Acelerando redes neurais de cápsulas com Lanes

CAMPINAS
2021

Vanderson Martins do Rosario

**Accelerating capsule networks with lanes
Acelerando redes neurais de cápsulas com Lanes**

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Edson Borin

Co-supervisor/Coorientador: Prof. Dr. Mauricio Breternitz Junior

Este exemplar corresponde à versão final da Tese defendida por Vanderson Martins do Rosario e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

R712a Rosario, Vanderson Martins do, 1993-
Accelerating capsule networks with lanes / Vanderson Martins do Rosario.
– Campinas, SP : [s.n.], 2021.

Orientador: Edson Borin.

Coorientador: Mauricio Breternitz Junior.

Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Visão por computador. 2. Aprendizado de máquina. 3. Redes neurais (Computação). I. Borin, Edson, 1979-. II. Breternitz Junior, Mauricio. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Acelerando redes neurais de cápsulas com Lanes

Palavras-chave em inglês:

Computer vision

Machine learning

Neural networks (Computer science)

Área de concentração: Ciência da Computação

Titulação: Doutor em Ciência da Computação

Banca examinadora:

Edson Borin [Orientador]

Felipe Maia Galvão França

Jefersson Alex dos Santos

André Carlos Ponce de Leon Ferreira de Carvalho

Rodolfo Jardim de Azevedo

Data de defesa: 17-12-2021

Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)

- ORCID do autor: <https://orcid.org/0000-0002-8737-0252>

- Currículo Lattes do autor: <http://lattes.cnpq.br/4833643221474007>



Universidade Estadual de Campinas
Instituto de Computação



Vanderson Martins do Rosario

Accelerating capsule networks with lanes
Acelerando redes neurais de cápsulas com Lanes

Banca Examinadora:

- Prof. Dr. Edson Borin
IC/UNICAMP
- Prof. Dr. Felipe Maia Galvão França
COPPE/UFRJ
- Prof. Dr. Jefersson Alex dos Santos
DCC/UFGM
- Prof. Dr. André Carlos Ponce de Leon Ferreira de Carvalho
ICMC/USP
- Prof. Dr. Rodolfo Jardim de Azevedo
IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 17 de dezembro de 2021

*Somewhere,
something incredible
is waiting to be known*
(Carl Sagan)

Acknowledgements

Since I was a child, I have had joy in discovering. Reality is stranger than the strangest thing that I could imagine—being part of the cosmos and being at least one of the ways the cosmos has to understand itself fills me with joy.

However, that will was only possible because it was lit and supported by many people. Thus, I dedicate this work to these people, and I will try to acknowledge some, even knowing that the list is too big to fit here.

Thank you: Carl Sagan, Carlos Alberto Panek, and Luciano Panek for starting my passion for science.

Thank you: all my teachers, in special: Anderson Faustino Silva, Edson Borin, and Mauricio Breternitz Jr., for holding the lamp and lighting my path.

Thank you: Maria Rosa Panek (my mon), Vagner Martins do Rosario (my father), Marisa Panek, and Caroline Aparecida Fazio. You all gave me strength, always were at my side, and made me believe that it was all possible.

Thank you: all my friends, all scientists, and all who share with me the passion for discovering.

Since the begging of my Ph.D., my country and the world seemed to be going on a dark path; but I met, during my Ph.D., enough good people to know that it will always be light to face the darkness.

Isn't it a beautiful thing that we are all part of the same universe?

Thank you all!

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. I would also like to thank Samsung and Petrobras for my Ph.D fellowships and Google Cloud for the grants.

Resumo

Redes de Capsulas, do inglês, Capsule Networks, ou apenas CapsNet, surgem como uma alternativa às redes convolucionais típicas, evitando camadas de pooling e melhorando a representação da orientação dos objetos na imagem. No entanto, sem as camadas tradicionais de pooling, essas redes não reduzem seu número de parâmetros em sua profundidade; além disso, o algoritmo de roteamento proposto para as CapsNet ainda não possui implementação eficiente, tudo levando a um alto tempo de execução no treinamento quando comparado a redes convolucionais tradicionais. Abordando esse problema, propomos a Multi-lane Capsule Network (MLC) que utiliza de diferentes caminhos sem dependência de dados entre si, ou lanes, para calcular as dimensões da cápsula final da rede. A MLCN alcança resultados similares à CapsNet, enquanto mantém sua capacidade de representação da orientação dos objetos, mas com execução até 130% mais rápida em uma única GPU. Além disso, nós mostramos que ao organizar a CapsNet em Lanes, abrimos oportunidade para um fácil paralelismo, o qual exploramos neste trabalho trazendo soluções para a paralelização, alocação de recurso e geração automática de redes em cenários com lanes e hardwares heterogêneos. Nós atingimos melhorias de desempenho que chegam a 7.18x em cenários com 8 GPUs, melhorias de 2x quando utilizando do nosso mecanismo de alocação de recursos, e redes 18.6% melhores com nosso sistema de geração automática de redes.

Abstract

Capsule Networks (CapsNet) surges as an alternative to typical convolution networks, avoiding pooling and improving pose representation to face challenges such as the Picasso problem. However, without pooling, the network does not reduce its number of parameters as it goes deep, and the routing algorithm does not have efficient implementations, thus leading to poor training performance. To face that, we propose the Multi-lane Capsule Network (MLCN), a reorganization of the original CapsNet, that has data-independent lanes to calculate the final capsule dimensions. MLCN achieves similar results as CapsNet, still maintaining its pose representation property but achieving up to 130% faster-training speed in single GPUs. Moreover, we show that the lane’s organization opens the opportunity to easily model parallelism, which we explore and bring solutions to heterogeneous lane and hardware scenario scheduling, parallelization, and automatic network construction with computational resources aware Neural Architecture Search (NAS). We achieve speedups as high as $7.18\times$ in 8 GPUs scenarios, $2\times$ training with our scheduling solution for heterogeneous lanes, and find 18.6% better networks with our NAS approach.

List of Figures

2.1	Illustration of a single neuron that compose Neural Networks.	20
2.2	A Deep Neural Network illustration with multiple layers of neurons.	20
2.3	Illustration of a typical Convolutional Neural Network architecture. From Wikimedia Commons.	22
2.4	Examples of data flow diagrams of Neural Networks inference.	23
2.5	Example of a data flow diagram for DL models' training.	26
2.6	Memory usage stacked during a DL model training.	27
2.7	Neural Networks Training Data Flow with the rematerialization of Layer 2.	28
2.8	Diagram with an example of the Bagging Mechanism.	30
2.9	Diagram with an example of the Stacking Mechanism.	31
2.10	Diagram with an example of the Boosting Mechanism.	32
2.11	Two images showing a face, but with the second one, transformed by a 180 degree rotation and zoomed. CNNs trained to recognize the first image can have difficult to recognize the second one as a face.	33
2.12	Example of a 3-dimension tensor with 32-values being pooled to a 2-values tensor using max pool.	34
2.13	Capsule vs Artificial Neuron mathematical equations. From the original Capsule Network paper [1].	35
2.14	Example of vectors agreeing on face features positions.	36
2.15	Visual representation of the dynamic routing agreement algorithm in a Capsule Network. The red highlighted capsule would be weighted by their distance to the clusters, so the order of weights would be: Middle, Top and Bottom.	37
2.16	CapsNet Encoder Architecture. From the original Capsule Network paper [1].	39
2.17	CapsNet Decoder Architecture. From the original Capsule Network paper [1].	40
4.1	Diagram of the Multi-Lane Capsule Network architecture.	50
4.2	Reconstruction of the original Fashion-MNIST inputs using the capsule output by the MLCN network after trained. The reconstruction is done using a fully-connected network.	52
4.3	Impact on the reconstruction of the input when synthetically varying each of the lanes output. It shows that different lanes are learning different features of the image, such as the color in Lane3.	53
4.4	Accuracy on the Fashion-MNIST and the CIFAR-10 datasets by MLCN (1 and 2) networks with different number of lanes.	54
4.5	Accuracy on the Fashion-MNIST and CIFAR-10 dataset by MLCN net- works with 8 lanes but different lanes width.	55

4.6	Impact of lane pruning on the accuracy and execution speedup of the network. Starting from the lane with least impact to the one with the largest impact.	56
4.7	Speedup of the three parallelization approaches: original CapsNet with data parallelism (base), MLCN with data parallelism (mlcn-data), and MLCN with model parallelism (mlcn-model). All speedups are relative to the original CapsNet on one Tesla K80 GPU.	58
4.8	MLCN and original CapsNet speedup scalability for 1, 2, 4, and 8 NVIDIA K80 GPUs using a Google Cloud VM with 24 vCPUs and 90GB of RAM. .	59
4.9	Accuracy impact with the increase of training batch size for the original CapsNet and MLCN in the CIFAR-10 and Fashion-MNIST datasets. Both are equally sensible with the increase of it.	60
4.10	Impact of different MLCN's lanes characteristics in the speedup scalability of the network model.	61
5.1	Diagram showing how multiple MLCN's lanes can be trained in parallel using multiple computing devices even in heterogeneous scenarios.	63
5.2	Average execution time of heterogeneous lanes running on four NVIDIA K80 GPUs with random and the greedy lanes' scheduling approaches. All lanes varying on width and depth.	66
5.3	Average execution time of heterogeneous lanes running on one K80, one P100, one V100, and one M40 NVIDIA GPU in multiple machines communicating using MPI with a random and a greedy scheduling of lanes execution distribution. All lanes varying on width and depth.	67
6.1	Diagram of NAS engine mechanism used to generated random MLCN models with heterogeneous lanes that naturally fit a given set of devices.	70
6.2	Average and best model's accuracy for 30 random generated models for each max FLOPS limit: 1x, 2x, 4x, and 8x. 1x starts with a previously given baseline size. All experiments were executed allowing the use of up to 12GB of memory and 8 buckets (GPUs).	74
6.3	Average and best model's accuracy for 30 random generated models for each memory limit: 2GB, 4GB, 8GB, and 12GB. All experiments were executed allowing the use of up to 8x FLOPS baseline and 8 buckets (GPUs). .	75
6.4	Average and best model's accuracy for 30 random generated models using 4, 8, 16, and 32 K80 GPUs. All experiments were executed allowing the use of up to 8x FLOPS baseline and 12GB of memory.	75
6.5	Best model's accuracy found by Random Search throughout its iterations for 4x FLOPS experiment.	76

List of Tables

- 3.1 Summary of applications of CapsNet in the literature discussed in this section. 43
- 3.2 Summary of enhancements proposed to CapsNet found in the literature and discussed in this section. 46
- 3.3 Comparison between original CapsNet and MLCN performance on CIFAR-10 and Fashion-MNIST datasets. 47
- 3.4 Summary of execution time improvements to CapsNet found in the literature and discussed in this section. 48
- 4.1 Comparison between the performance on the CIFAR-10 and Fashion-MNIST dataset of the original CapsNet and the two MLCN architecture organization proposed (Mlc1 and Mlc2). 57

List of Algorithms

1	Neural Network Inference Algorithm.	22
2	Dynamic Routing Algorithm.	38
3	Greedy Scheduling Algorithm.	65
4	Device Memory Bucket and FLOPS Bucket Representation.	72
5	Generator of random MLCNs that fit devices.	73
6	MLCN Random Network Architecture Search.	73

List of Abbreviations

ANN - Artificial Neural Network

CNN - Convolutional Neural Network

CapsNet - Capsule Network

DL - Deep Learn

DNN - Deep Neural Network

FLOPS - FLoating-point Operations Per Second

FPGA - Field-Programmable Gate Array

GPU - Graphics Processing Unit

HPO - Hyper Parameter Optimization

HW - Hardware

MLCN - Multi-Lane Capsule Network

ML - Machine Learning

NAS - Neural Architectural Search

NN - Neural Network

SGD - Stochastic gradient descent

VM - Virtual Machine

Contents

1	Introduction	16
2	Fundamental Concepts	19
2.1	Inference	22
2.2	Training	24
2.3	Ensemble Learning	29
2.3.1	Bagging	29
2.3.2	Stacking	30
2.3.3	Boosting	31
2.4	Capsule Network	32
3	Related Work	41
3.1	Applications	41
3.2	Architectural Enhancement	44
3.2.1	Execution Time Improvement on CapsNet	46
4	Accelerating Capsule Networks with Lanes	49
4.1	CapsNet Parallelization	50
4.2	Experimental Setup	51
4.2.1	Dropout and Regularization in MLCN	52
4.3	Experimental Results	52
4.3.1	Number of Lanes vs. Model Accuracy	53
4.3.2	Lane's Width vs. Model Accuracy	55
4.3.3	Lane Dropout Trade-off	55
4.3.4	MLCN Training and Inference Time	56
4.3.5	MLCN Scalability and Performance Study	57
4.4	Summary and Considerations	61
5	Optimizing MLCNs on Heterogeneous Scenarios	63
5.1	Execution Cost Model for MLCN Lanes	64
5.2	Scheduling Heuristic	64
5.3	Experimental Setup	65
5.4	Experimental Results	66
5.4.1	Heterogeneous Lanes with Heterogeneous GPU	67
5.5	Summary and Considerations	67
6	Optimizing MLCN to Computing Devices with NAS	69
6.1	Memory Cost Model for MLCN Lanes	70
6.2	Lane Configuration Search	71

6.3	Experimental Setup	73
6.4	Experimental Results	74
6.5	Summary and Considerations	76
7	Conclusions	77
7.1	Challenges	78
7.2	Future Work	79
	Bibliography	80
	A Publications	86
	B Source Code and Code Usage	89

Chapter 1

Introduction

Deep Learning has become a widely used machine learning technique to solve many different problems, from image processing to language translation to audio transcription, amongst many others. In 2014 after the publication of the AlexNet architecture (stacking multiple layers of convolutions and max-pooling) [2], deep learning became the state-of-art in image classification with the use of Convolutional Neural Networks (CNNs). One of the main mechanisms in these traditional CNNs is the Pooling operations that, although achieving outstanding results, add transitional invariance and loss of information. That is why Sabour *et al.* [1] proposed a novel approach to routing data in the network (dynamic routing algorithm) without using the traditional pooling mechanisms and demonstrated it with a Neural Network called CapsNet. To this, it uses vectors to encode features of the image instead of scalars as in a usual CNN, and a dynamic routing algorithm is used to guarantee the global relationship between all vectors. The traditional CNNs can easily miss global relationships and, for instance, miss-classify an image as a face for having a mouth, eyes, and nose, independently of the order or relative position of these features.

Despite promising preliminary results, CapsNets are still a young and not much-explored models. For example, one of the challenges that have been described using and testing CapsNet is that they have far larger training times than CNNs. Therefore, in this work, we explore the CapsNet architecture, proposing a new organization for it. We split the original CapsNet into multiple lanes that are data-independent and responsible for learning different dimensions of the final vectors of the CapsNet. This organization outperforms the original CapsNet in training and inference time by adding more data parallelism and reducing trainable parameters. We also show that this organization helps with the explainability of the network. All this, without losing learnability and generability performance. We show that we can easily construct a faster CapsNet divided into lanes that outperform the accuracy of the original one for the Fashion-MNIST [3] and Cifar10 [4] datasets. We call this new CapsNet architecture a Multi-Lane Capsule Network (MLCN).

These data-independent lanes also open opportunities for more straightforward multiple hardware parallelism. Several approaches to the distributed model parallelization of Deep Neural Networks (DNN) have concentrated in their depth dimension [5, 6, 7], but DNNs can also be organized in a way to be parallelized along their width dimension [8]. The MLCN creates separable and resource-efficient data-independent paths in the

network that can learn different features or add resilience to the network. As these lanes are data-independent, they can be (1) processed in parallel and (2) specialized for distinct computational targets (CPUs, GPUs, FPGAs, and cloud), as well as resource-constrained mobile and IoT targets, leading to opportunities and challenges. In this work, we also present a comprehensive study of the scalability and efficiency of MLCN for multi-GPU systems.

This dissertation also explores how MLCN with heterogeneous lanes (differences in size, type, width, and other characteristics) can lead to better performance accuracy and how these MLCNs are harder to schedule when using a multi-GPU system. To face that, we show that the hardware’s lane schedule can be seen as an optimization problem and we present and experiment with a greedy approach to the problem. We also show that a simple greedy heuristic can generate a schedule that runs almost 50% faster than a random naïve approach.

Finally, we explore a Neural Architecture Search strategy to automatically generate custom MLCN models with good lanes configurations that best fit a given hardware setup. Generating such models for 64 iterations allowed us to find models with more than 18% better accuracy for CIFAR-10. All this while maintaining faster performance without extra effort as the generated models are already loaded balanced.

Thus, the main contributions of this work can be summarized as:

- We present a new organization to Capsule Networks, the MLCN, that increases its execution speed and opens opportunities for parallelism;
- We present a first comprehensive analysis of the efficiency and scalability of MLCN, showing its advantages over the data-parallelism-limited approach of the original CapsNet;
- We define the load balancing problem of distributing heterogeneous MLCN lanes in heterogeneous hardware;
- We present a greedy heuristic to solve the lane-hardware match problem showing that it is superior to a naïve approach; and
- We show how to use NAS to create MLCN models that achieve balanced execution in a set of devices with better accuracies.

Outcomes from this dissertation were published in the IEEE Signal processing letters (2019) presenting for the first time the Multi-lane Capsule Network architecture [9]. Later, an analysis of the MLCN efficiency and scalability was presented on the 31st International Symposium on Computer Architecture and High Performance Computing (2020) [10]. Finally, an extension of the symposium paper was published on the Journal of Parallel and Distributed Computing (2021) bringing new insights on the parallelization and automatic exploration of MLCN configurations [11].

This dissertation is organized as follows: Chapter 2 describes the basics concepts related to Neural Networks, Neural Networks execution performance, and Capsule Networks; Chapter 3 presents the relevant works done with Capsule Networks after its original

publication and the works that studied and used MLCN; Section 4 shows the Multi-Lane Capsule Network and its performance analysis when compared to the original Capsule Network. Chapter 5 presents the problem and analyzes a greedy solution to the parallelization of MLCN into heterogeneous hardware and when the MLCN has a heterogeneous set of lanes; finally, Chapter 6 shows a NAS approach to create MLCN architectures that fit a given set of hardware substrates with the best accuracy possible and Chapter 7 presents our conclusions, and future work.

Chapter 2

Fundamental Concepts

This chapter introduces basic concepts related to Deep Neural Networks (DNNs), DNNs Code Generation, and parallelization.

A set of connected artificial neurons forms a Neural Network. This network can be seen as a direct graph where nodes are neurons, and edges are neuron connections. Each node, or neuron, has inputs, represented by entering edges, and produces an output propagated by leaving edges. Every edge has a weight that is updated during the learning process of the Neural Network.

Input edges of an artificial neuron can be both the output of others neurons or the network's input. A neuron sums the values of all its inputs values (x_i) weighted by their edges weights (w_i). It then adds to the sum a bias value (b) and, finally, uses this scalar value as the argument of a non-linear activation function (σ). See equation 2.1.

$$y = \sigma\left(\sum_i^{\#inputs} w_i x_i + b\right) \quad (2.1)$$

The weights and biases are learned and can change during the training process. The value resulting from the activation function is called activation (y), and it is the neuron's output. The activation will be used as the input of other neurons or be used as the Neural Network's final output. Visually a neuron can be seen as represented in Figure 2.1.

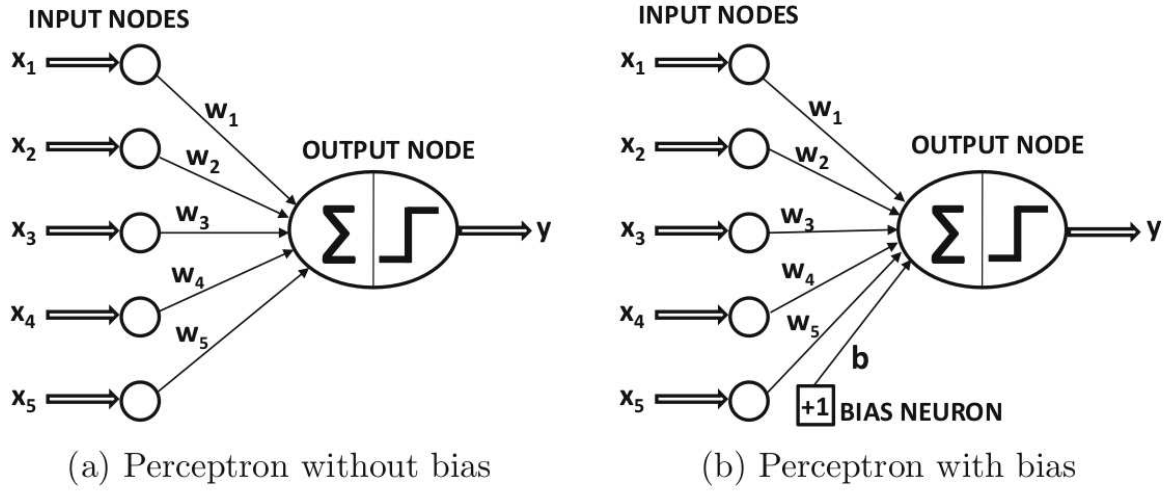


Figure 2.1: Illustration of a single neuron that compose Neural Networks.
[12]

A Neural Network then is the composition of a set of non-linear functions creating a complex function that can fit complex data patterns. Neurons can be organized in layers (a set of neurons that has input neurons from the layer before, and their outputs are inputted in the layers after). Neurons in a layer are not connected with themselves. We can see this layer organization illustrated in the diagram in Figure 2.2.

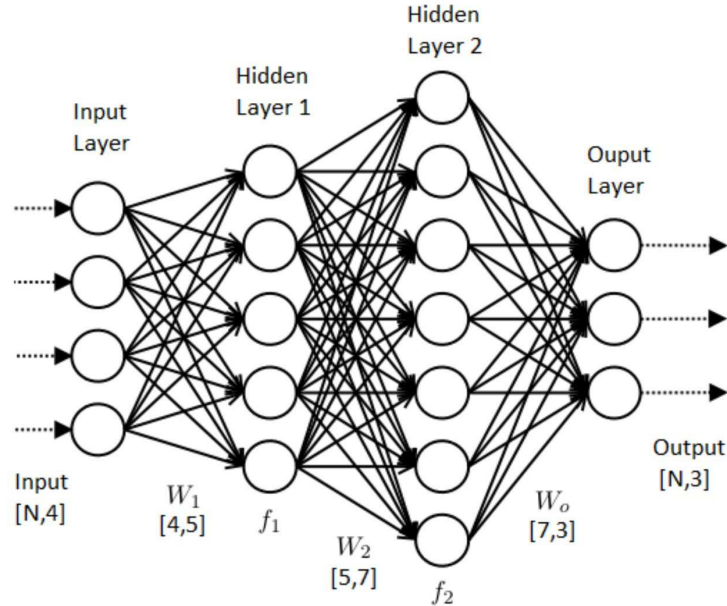


Figure 2.2: A Deep Neural Network illustration with multiple layers of neurons.

In the figure, we have one layer connected to the input, two layers in the middle (input and output are other layers), called hidden layers, and finally, the output layer. In a fully connected network, each neuron in one layer is connected to every other neuron in the layer after. If a layer with N neurons is fully connected to another layer with M neurons, we need $N \times M$ weights, one for each connection.

We can describe the weights, inputs, activations, and outputs of each layer as multi-dimensional vectors. In the field, these vectors are referred to as tensors. It allows us to write all layers neuron calculations as single vector operations such as in Equation 2.2, where W is a tensor with all weights in the layer, X is a tensor with the input of the layer (can be the input of the network or the output of the previous layer), B is a tensor with the bias for the layer and Y is the result of the tensorial operation applied to the activation function σ .

$$Y = \sigma(W \times X + B) \quad (2.2)$$

In the matter of describing complex non-linear functions, it is known that deeper networks (with more layers) are exponentially more efficient than wider layers. In other words, to describe the same function with fewer layers, we would need layers exponentially larger. That is why Deeper Neural Networks have become popular, surging with the Deep Neural Networks (DNNs).

However, increasing the deepness of a network does also have its limitations and challenges. In the backpropagation algorithm, deeper networks suffer from the Vanishing gradient problem [13]. The gradient update tends to zero with deep compositions of the chain rule.

With time, more specific layers were introduced in DNNs. For example, convolution layers and pooling layers as commonly used for image data. The first applies a traditional kernel convolution operation in the activation tensor of the layer before [13]. However, the weights of the kernels being applied are not predefined as in traditional image processing, but learned. The outputs of the convolutions are also tensors, so that they can be naturally connected to the next layer. The second, pooling [13], is a layer that subsamples its input tensor, reducing its size. Different approaches can be used to pool, such as just selecting parts of the input tensor's values or taking the average of a group of tensor values to represent the whole group as just one number. This technique can sound counter-productive as we are losing information, but it helps reduce the redundant information and, more importantly, adding some position invariance to the network.

If you present a cat picture to a network to teach it to detect cats, the cat will be in a specific position in the image and a specific orientation. It would be useless if the network could not, after training, recognize the same cat but in a different position or orientation. For that, the pooling layer is helpful to normalize minor variances. Rotating this part or moving it a little bit, causes negligible impact on the average as, although the positions of the pixels have changed, its values are the same.

Notice that any layer that input a tensor and output a tensor can be coupled to a network. Of course, as we describe in the next section, the layers must be derivable so that gradient descent algorithms can be used to adjust their weights, a process called model training.

Networks that use convolutional layers are called Convolution Neural Networks (CNNs) and are state of the art in many image-related tasks. The diagram of Figure 2.3 shows a typical CNN architecture (organization) for the task of image classification.

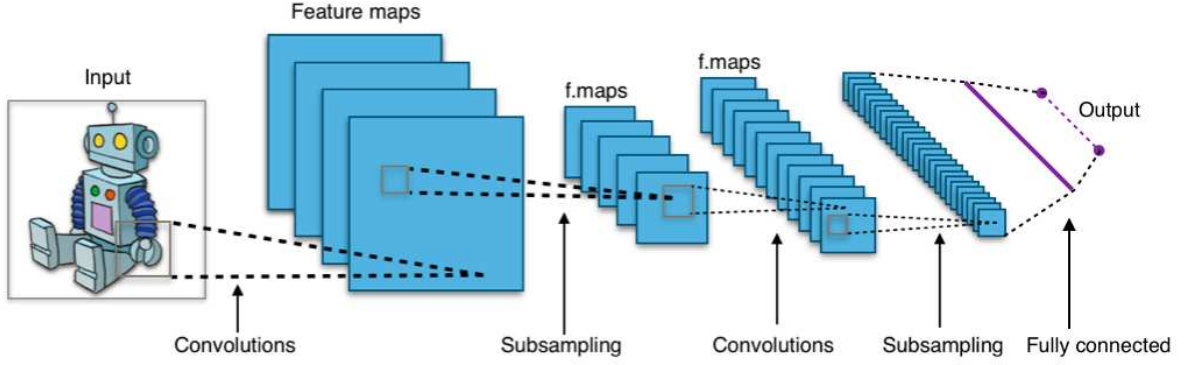


Figure 2.3: Illustration of a typical Convolutional Neural Network architecture. From Wikimedia Commons.

2.1 Inference

Equation 2.2 shows that we can calculate all activations Y from a layer by using tensor operations. For a network with K layers (considering the input and output values as layers too for simplification), the inference process calculates the output value (layer K) from the input values (layer 0). We can denote the weight, bias and activation function of the i^{th} layer as W^i, B^i, A^i . The inference process is described by Algorithm 1. This, for a fully connected network, for other kinds of layers, the expression $A^i(W^i \times X + B^i)$ will vary following the layer specification. However, the logic will be the same: the layer's input is the activation of the layer before.

Algorithm 1 Neural Network Inference Algorithm.

```

1 # Given the weights  $W$ , bias  $B$  and activation functions  $A$  of all layers
2  $Y = []$ 
3  $X = \text{Inputs}$ 
4 for  $i$  in  $\text{range}(0, K)$ :
5      $Y = A^i(W^i \times X + B^i)$ 
6      $X = Y$ 
7 return  $Y$ 
```

We can see from Algorithm 1 that the data flow from one layer to another linearly. However, that is not always the case. Some networks can have layers with multiple layers as input and with multiple layers using their outputs. One such example is the skip connections, used to reduce the vanishing problem in deep networks, such as in the ResNet [14]. All these cases are creating more complex data flow structures.

Independently of its complexity, NNs are naturally represented as dataflow graphs. In these graphs, nodes represent tensor operations (layers), and edges represent data flow between these operations. These dataflow graphs are not only used to illustrate the networks, but they are also the way that frameworks, interpreters, and compilers represent these NNs in memory. Three examples of data flows that can be found in the inference calculation of Neural Networks are illustrated in Figure 2.4. Figure (a) with a simple forward flow, (b) with skip connections, and (c) with multiple layer input/output blocks.

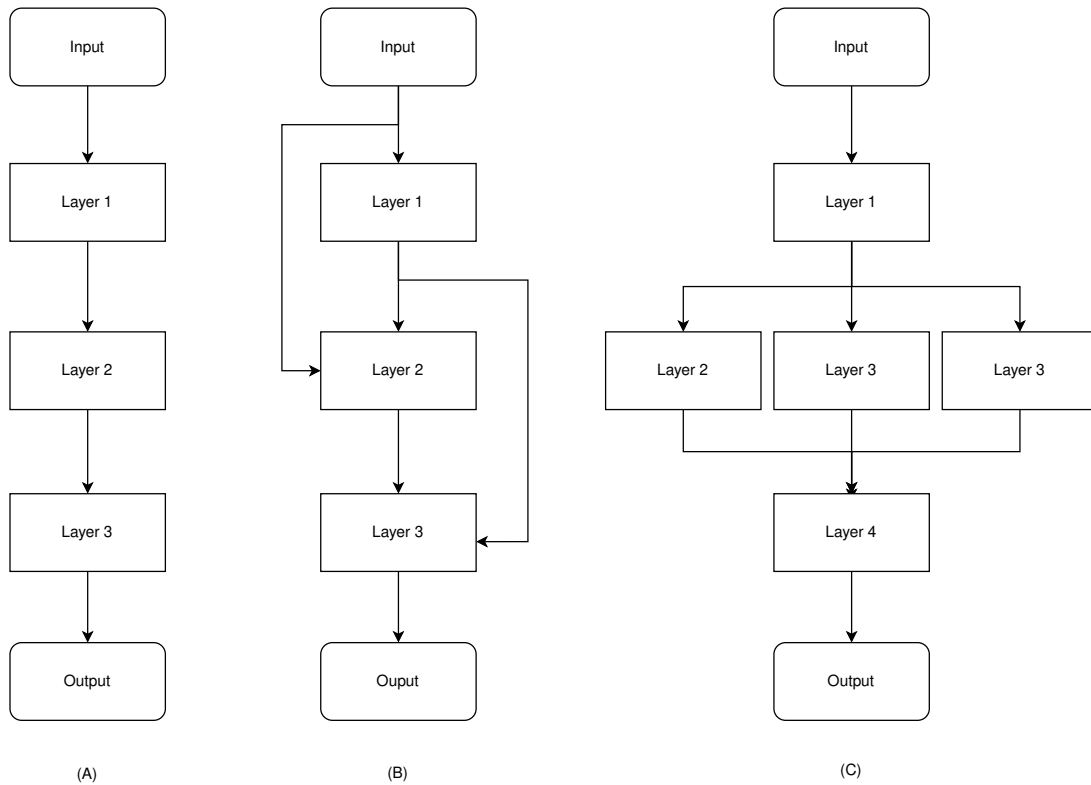


Figure 2.4: Examples of data flow diagrams of Neural Networks inference.

Notice that for less straightforward networks, running the inference requires scheduling to decide which layer to execute first. There is optimal execution scheduling for any network and hardware, and that is usually a compiler optimization task.

The scheduling of a Deep Learning Model (DL Model) affects not only its performance but also its memory usage. Notice that we can release the activations from a layer for a data flow such as in (A) just after the next layer has been calculated. Thus, never having more than two layers in memory at any time. However, in (B), the output of a layer can be used not only by the next layer but by other layers located further down on the model. Thus, layer activations may need to be stored in memory until the last layer that uses it is executed. The order that the layers are executed then affects how long activations will be stored in memory.

The execution of such dataflows can be done by an interpreter that walks through the graph executing layer per layer, by a JIT compiler that compiles a group of layers and then jumps to the compiled code, or by a compiler that generates the binary code for the whole network that can then be executed.

Neural Networks use a small set of common layers; it is common to have the optimized implementation of these layers that are just called either by the interpreter, JIT, or compiler. These optimized implementations are called kernels. Sometimes multiple kernels can be used to execute the logic of one or a set of layers, thus being an optimization problem to the interpreter or compiler to find the best kernel.

Another critical aspect of the inference execution is that the multidimensional tensors flowing through the graph can be represented in memory in different ways. The memory is linear, so one needs to map the multiple tensor dimensions into one memory dimension.

Because of memory locality aspects, the way that this mapping is done can affect the execution performance. Moreover, as the dimensionality of the data changes over the network flow, one optimal performance representation for one stage may not be optimal for another stage.

Notice that to transform from one data representation to another have a cost, so the interpreter/compiler must decide if the gain of changing the representation is more significant than the cost of transforming the data in memory. This can be represented as a graph where we have a copy of the dataflow graph for each data representation, and we add edges from all nodes from one graph to the same node to another graph with a transformation node in between. With the cost of executing each layer, including the transformation from one graph to another, we can use a shortest-path finder algorithm to solve it. This optimization is called layout search [15].

With the data flow scheduled, the data layout used for each layer, and the best kernel to execute each layer, the inference becomes a sequence of calls to these kernels.

2.2 Training

The goal of a machine learning algorithm is to learn from data and generalize this learned knowledge. In practice, the algorithms learn complex non-linear functions that fit the training data but do not overfit it to be used for new data.

In a DL model, we saw in the last section that this complex function comes from the composition of multiple non-linear functions. Each of these functions has a set of weights and biases that needs to be set to define the shape of that function. The goal of the training process thus is to find the weight and bias that creates a function that fits the data. These values that are changed during training are called training parameters.

One important aspect of the training process is that we need a way to measure our function, or how well it fits our data. For that, we define a loss function that represents in a real number how far the results the network is getting are from the ones that are expected. Examples of a loss function the Mean Squared Error (MSE) and the Binary Cross Entropy.

As mentioned, the training goal is to minimize a give loss function by changing the training parameters. The weights and bias may be started with random values (the way that they are randomized matter [16]), and their values are updated in other to try to minimize the loss function.

One possible approach is to randomize the training parameters a few times and pick the ones that achieve the best value in the loss function. However, that can take a long time as there is no guarantee of convergence. So, a more common approach uses the derivate of the layers functions to update its weights and bias. If all layers are derivable, we can use the chain rule to calculate the training parameters' change that minimizes the loss function for all layers. This training algorithm is called Gradient Descent.

Gradient Descent is an optimization method used to find a value in the domain of a function f such that the image value of that function is minimized. In the context of deep learning, the domain can be represented as the training parameters of the DL Model and

f by the loss function. The parameter values that minimize the function are denoted as x^* , like in the Equation 2.3.

$$x^* = \operatorname{argmin} f(x) \quad (2.3)$$

To find x^* , the algorithm uses the gradient of f : a vector containing every partial derivate of the function ($\frac{\delta}{\delta x_i} f(\mathbf{x})$), as following,

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = (\frac{\delta}{\delta x_1} f(\mathbf{x}), \frac{\delta}{\delta x_2} f(\mathbf{x}) \dots \frac{\delta}{\delta x_n} f(\mathbf{x})) \quad (2.4)$$

Based on the derivate concept, $\nabla_{\mathbf{x}} f(\mathbf{x})$ can be used to update the parameters of a function in the direction in which $f(\mathbf{x})$ is minimized. This update process can be written as follow,

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (2.5)$$

in which ϵ is an update factor commonly called a learning rate ranging from 0 to 1, x' is the updated set of parameters, and x is the original set of parameters. If such update is done for enough amount of iterations and f is a convex function, then x^* should be found or at least close approximated.

In the specific case of deep learning, we need to calculate the gradient for all training inputs ($\mathbf{x}_{(i)}$). We can represent the mean loss of all training inputs as $\mathbf{J}(\boldsymbol{\theta})$:

$$\mathbf{J}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}) \quad (2.6)$$

where θ represents the network parameters, L is the loss function, $\mathbf{x}^{(i)}$ e $\mathbf{y}^{(i)}$ are the input and the expected output for the i^{th} training sample. For that, we can obtain the gradient of J as:

$$\nabla_{\boldsymbol{\theta}} \mathbf{J}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}, \boldsymbol{\theta}) \quad (2.7)$$

So, for each input sample i we need to calculate the value of $\nabla_{\boldsymbol{\theta}} L(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}, \boldsymbol{\theta})$.

We need the gradient for each layer so we can update their weights and bias. We can use the chain rule to calculate the gradient per layer. So, to calculate the gradient for the i^{th} layer, we need the activations of that layer and the gradient of the $(i + 1)^{th}$ layer.

Therefore, for each input in the training set, we have to execute the inference (forward pass), compare the inferred value with the expected output, calculate the loss, and, finally, calculate the gradient layer by layer from the output to the input layer using the chain rule (backward). Finally, use all input gradients average to update the weights and bias.

Notice that such an algorithm can become too expensive as the number of input examples increases [13]. To work around this, typically, Stochastic Gradient Descent (SGD) is used, instead. It proposes the calculation of $\nabla_{\boldsymbol{\theta}} \mathbf{J}$ by using random input samples of the training set. These random samples are named mini-batch. For each step of the algorithm, a new random mini-batch is generated, and the estimated gradient is calculated

from it [13]. If a mini-batch has size m' and the whole training set has size m , after m/m' steps, we say that one epoch was executed.

The number of epochs necessary for the SGD algorithm to converge can vary. Commonly, a stop criteria is used, such as a fixed number of epochs or a minimal loss or accuracy improvement threshold.

With mini-batches, we can store all inputs that are going to be used in Equation 2.7 in memory. This means that we do not need the summation iteration. We can use the mini-batch as one of the tensor's dimensions. This also means that we can calculate the gradient using this tensor and just after update its weights.

The training algorithm can also be expressed as a data flow graph, as illustrated in Figure 2.5. In training, a new path need to be account for the gradient calculation.

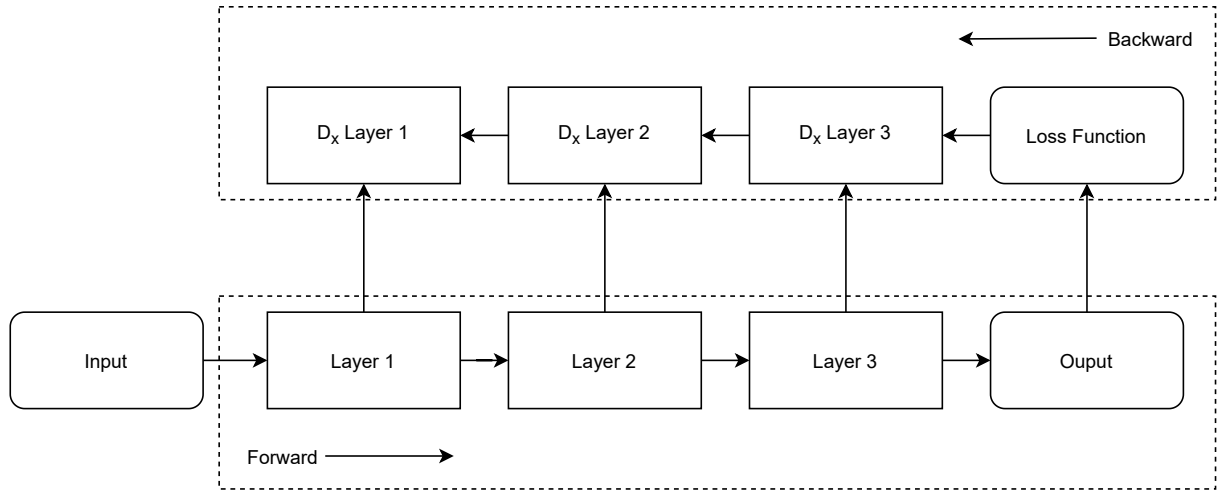


Figure 2.5: Example of a data flow diagram for DL models' training.

Notice that we now have two paths in the data flow, one forward to calculate the inference output and one backward to calculate the gradients and update the weights. Furthermore, we have a data dependence between the forward layer and its backward gradient calculating, implying more complex data dependence graphs.

All the matters mentioned in the inference section about interpreting or compiling the network are still valid and applicable to the training data flow. However, now we have a more considerable amount of calculation and data updates which means a more extensive execution time. This brings attention to the parallelization of the training to increase its performance.

There are three main ways to parallelize DL models' training. However, nothing prevents the use of the three parallelization techniques together.

The first is by parallelizing the kernel's layout executions. The layers are mainly tensor multiplication algorithms, and parallel algorithms can be used to perform that task.

The second is by cutting the mini-batch into smaller batches, each being executed in a different machine or accelerator. That technique, commonly referred to as data parallelization, has a pretty straightforward implementation, but it replicates the model in every execution engine being used. This leads to non-optimal usage of the devices' memory (although the memory will be reduced as the batch dimension size of the tensors

is reduced). Further, splitting the batch and using partial batch gradients affects the network loss performance [17].

The third way is by partitioning the dataflow graph and executing different model parts on different devices, which has the drawback of potentially having to communicate information from the edges that cross the partitions.

Finding the best way to parallelize a network is also an optimization problem. For example, finding the partitioning in the network dataflow graph minimizes device communication but still leads to a good load balance is challenging.

Another essential training execution aspect is memory consumption. Activations from intermediate layers in the forward pass need to be stored until their respective gradient calculation layer be executed. They are leading to a peak in memory just after the end of the forward pass and beginning of the backward pass, where all activations are in memory as seen in Figure 2.6.

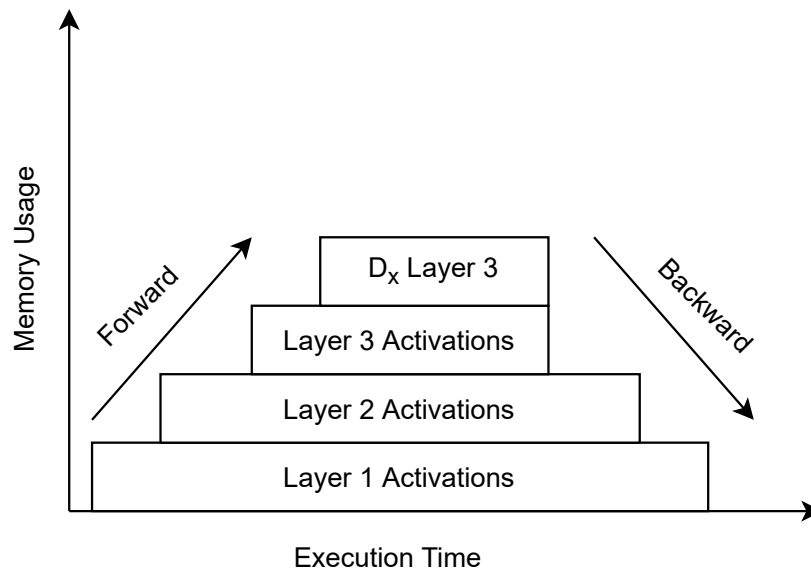


Figure 2.6: Memory usage stacked during a DL model training.

The interpreter or the compiler can throw away activations stored and recompute them when needed to reduce the memory peak. A technique that exchanges memory usage with computational overhead is called rematerialization¹. Finding a set of nodes in the data flow that reduces the memory usage with the least overhead is an optimization problem. Mixed-integer programming approaches were proposed to such problems and also heuristics. Usually, the approaches need to consider both the scheduling and rematerialization problems because one affects the other.

The rematerialization (recomputation) of a node in the data flow can be seen as a copy of such node. In Figure 2.7, we can see the recomputation of layer two by the creation of a new node. Notice that to recalculate the value of layer 2, we still need the activations of the previous node layer 1. Since it is already in memory at that point, it does not

¹During my internship at Microsoft Research, I implemented a rematerialization heuristic on the Tensorflow XLA compiler. The implementation is open source at Microsoft GitHub: <https://github.com/microsoft/tensorflow-rematerialization>

affect memory usage. However, in layer three and Dx Layer 3 nodes, the value of layer two activations is not needed, thus reducing the memory peak.

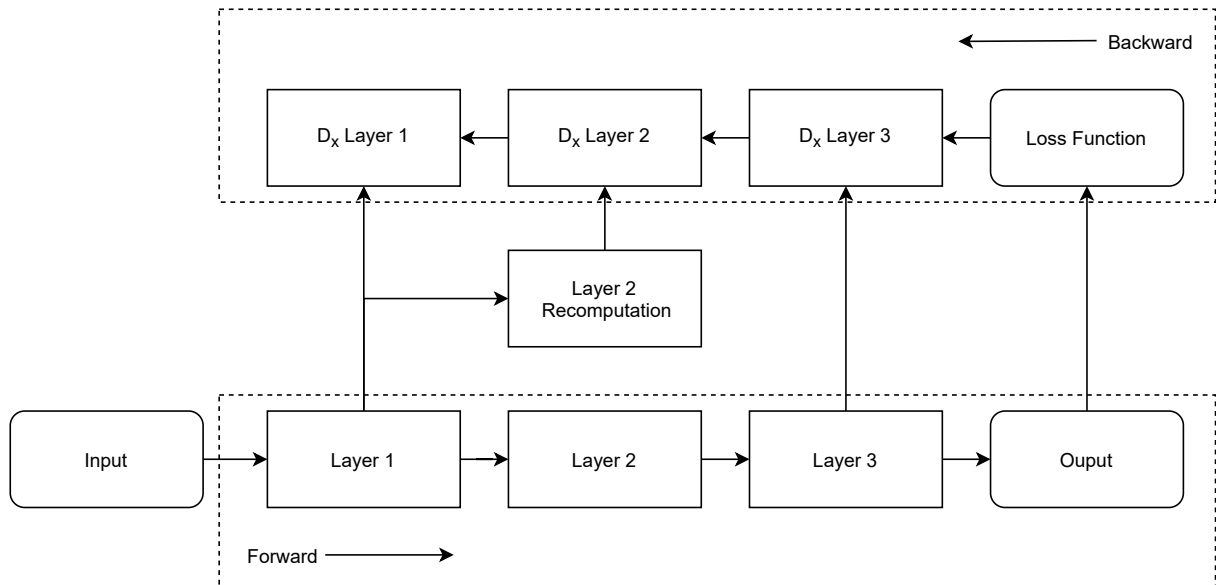


Figure 2.7: Neural Networks Training Data Flow with the rematerialization of Layer 2.

Another technique used to reduce memory pressure, mainly in accelerations, is to offload the activations to the main memory and just maintain a subset of the ones that are going to be used next on the GPU memory. That means transferring activations toward and backward the main memory, which also implies computation overhead. Finding the set of activations that should be in the accelerator memory at any given time is also an optimization problem.

Finally, we can also reduce the memory usage by reducing the number representation precision of the weights, bias, or/and activations (quantization), or by compressing sparse tensors, summing up with the following techniques to address memory usage limits: rematerialization, memory offload, model parallelism, model compression, and quantization.

Nothing prevents using more than one of these techniques together. However, notice that quantization can impact the network loss/accuracy performance [18], and techniques such as rematerialization and memory offload affect the execution performance.

In the context of deep learning, the training algorithms are called optimizers. SGD, for instance, is an optimizer, but there are others for the same purpose. Different optimizers can be more or less efficient for different applications. It is a hyper-parameter of the network.

Hyper-parameters are configurable parameters of the network or the network execution that changes the training behavior. The training algorithm does not update or search their values, like the training parameters (weights and bias). Examples of hyper-parameters are the learning rate, network regularization L1 and L2, number of layers, type of layers, optimizer algorithm, among many others.

Yu and Zhu [19] divide hyper-parameters into two groups: those used in training and those used in the modeling of the Neural Network. The task of automatically figuring out the best hyper-parameters from the first group is generally referred to as Hyper-

Parameter Optimization (HPO). When the second group of hyper-parameters (network modeling, such as the number of layers) is also an optimization object, the task is called Neural Architectural Search (NAS).

2.3 Ensemble Learning

Ensemble learning is a technique in machine learning that combines predictions from two or more models [20]. Many ensemble-based classifiers have been proposed over the last decades, but most of them are just variations of a few established algorithms whose capabilities have been extensively tested and reported [21]. There are three classes of ensemble techniques that encompass the majority of all ensembles used in practice: Bagging, Stacking, and Boosting. Their success comes from their easy implementation together with success on a wide number of problems.

There is an algorithm for each one of these three techniques, but given their success, many extensions and related techniques were proposed and appeared. However, for the sake of understanding Ensemble learning, understanding the base of each technique is enough and also more useful. It is useful to summarize and contrast each approach. We do this in the next subsections. It is also important to remember that these three methods alone do not define the extent of ensemble learning.

2.3.1 Bagging

Bagging names come from Bootstrap AGGREGatING. The two key parts of the Bagging techniques are bootstrap and aggregation. Typically, it involves using a single machine learning algorithm, almost always a decision tree, and training the model on a different sample of the training dataset. The predictions made for each model are combined using vote or averaging.

The bootstrap step guarantees that each model will be trained with different sample data creating a diversity of trained models, this is also achieved by using simple classifiers whose decisions vary with respect to relatively small changes in the training data [22]. A key point for this method is the way that each sample of the dataset is prepared, so each model gets its own unique dataset sample.

One possibility is to randomly draw from the dataset with replacement. Replacement means that if an element is selected, it is returned to the training dataset for potential re-selection in the same training dataset. Meaning that an element may be selected multiple times. This is called a bootstrap sample. It is a technique often used in statistics with small datasets to estimate the statistical value of a data sample. By preparing multiple different bootstrap samples and estimating a statistical quantity and calculating the mean of the estimates, a better overall estimate of the desired quantity can be achieved than simply estimating from the dataset directly.

In the same manner, multiple different training datasets can be prepared, used to estimate a predictive model, and make predictions. Averaging the predictions across the models typically results in better predictions than a single model fit on the training dataset directly.

It is a general approach and easily extended. For example, more changes to the training dataset can be introduced, the algorithm fit on the training data can be replaced, and the mechanism used to combine predictions can be modified. A visual representation of such a bagging mechanism can be seen in Figure 2.8.

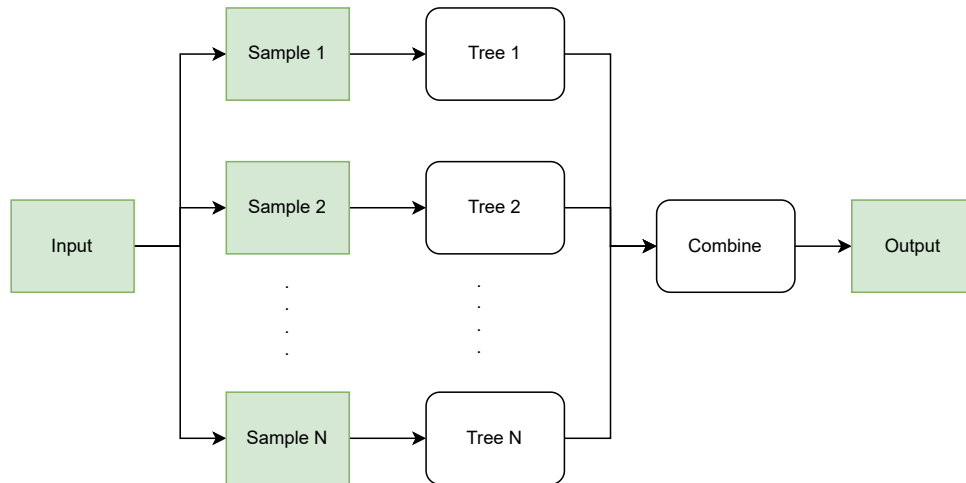


Figure 2.8: Diagram with an example of the Bagging Mechanism.

2.3.2 Stacking

Stacking is a generic technique in ensemble learning where a learner is trained to combine the result of individual learners. So, there are a first-level of learners whose output is the input of a second-level learner also called meta-learner.

A two-level stacking is the most common case, but one may find models with more layers. For instance, we could have three levels organized such as a neural network but with the neurons being machine models themselves.

Stacking is the most used ensemble technique used giving its simplicity and power. The meta-learner is trying to learn which classifiers are reliable and which are not for each class of data. Any machine learning model can be used to implement the meta-learner, but it is common to use a linear model, such as linear regression for regression and logistic regression for binary classification. This encourages the complexity of the model to reside at the lower-level learners and simple models to learn how to harness the variety of predictions made.

Different than bagging that the diversity is focusing on the different samples of data, in stacking the diversity comes from the different machine learning models used as ensemble members. As such, it is desirable to use a suite of models that are learned or constructed in very different ways, ensuring that they make different assumptions and, in turn, have less correlated prediction errors. A visual representation of such a stacking mechanism can be seen in Figure 2.9.

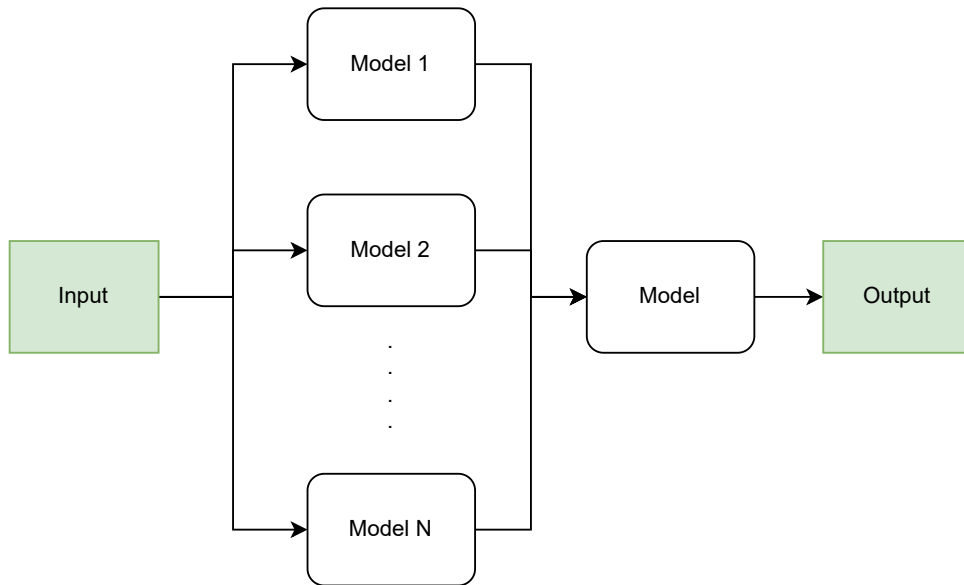


Figure 2.9: Diagram with an example of the Stacking Mechanism.

2.3.3 Boosting

Another generic ensemble technique vastly used is boosting. Its focus is changing the training data to focus attention on examples from the training dataset that previous fit models got wrong. In boosting, the training dataset for each subsequent classifier increasingly focuses on misclassified instances from previously classifiers.

The models added to the ensemble sequentially such that the second model attempts to correct the predictions of the first model, the third corrects the second model, and so on. The key property of boosting ensembles is the idea of correcting prediction errors. Typically, the training dataset is left unchanged, and instead, the learning models are modified to pay more or less attention to specific examples based on whether they have been predicted correctly or incorrectly by previously added ensemble members. For example, the data can be weighed to indicate the amount of focus a learning algorithm must give while training the model.

The idea of combining weaker models to create a strong one using boost was first proposed theoretically and it was difficult to make it work in practice. It was not until the Adaptive Boosting (AdaBoost) algorithm was developed that boosting was demonstrated as an effective ensemble method [23]. After it, stochastic gradient boosting [24] was presented and it is among the most effective techniques for classification and regression on structured data.

A visual representation of such a boosting mechanism can be seen in Figure 2.10.

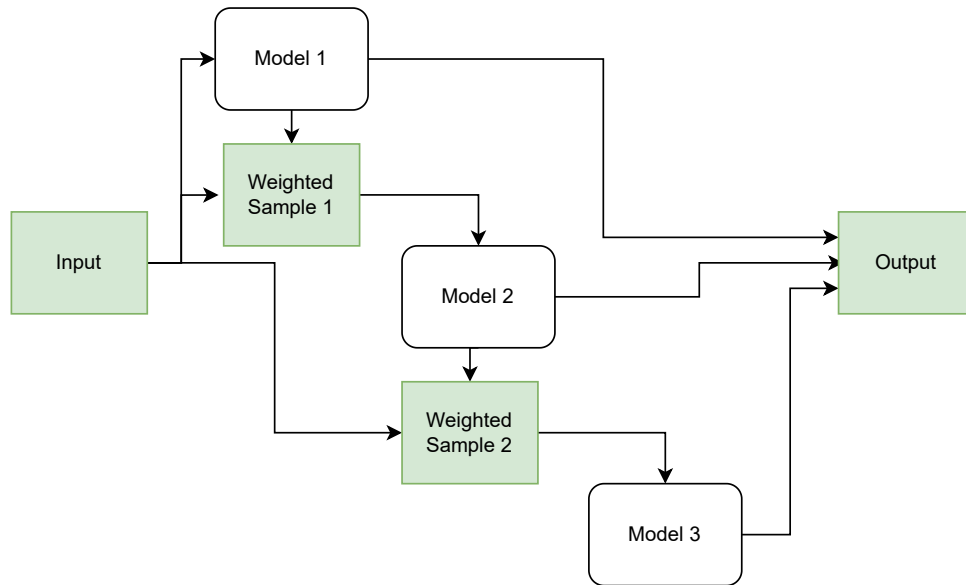


Figure 2.10: Diagram with an example of the Boosting Mechanism.

2.4 Capsule Network

Convolutional Neural Networks (CNNs) are one of the most popular deep learning architectures widely used for computer vision, and they are state-of-art in many computer vision problems. From image classification and object detection to image segmentation, CNNs define the current state of the art. However, these networks still have their drawbacks and open challenges.

Yann LeCun first proposed the CNN in the year 1998. He was then able to detect handwritten digits with a simple five-layer CNN trained on the MNIST dataset. The idea was simple: train the network, identify the features in the images, and classify them. In 2019, EfficientNet-B7 achieved the state of the art performance in classifying images on the ImageNet Dataset. The network can identify the label of a particular picture from over 1.2 million images with 84.4% of accuracy. Looking at these results and progress, we can infer that convolutional approaches make learning many sophisticated features with simple computations. However, they are not infallible. CNNs are challenged when presented with images of different sizes and orientations.

If we rotate a face upside down (see Figure 2.11) and then feed it to a CNN trained with humans face, it could not be able to identify features like the eyes, nose, or mouth. Similarly, if you reconstruct specific regions of the face (*i.e.*, switch the nose and eyes), the network will still recognize the face—even though it is not exactly a face anymore. In short, CNNs can learn the patterns of the images statistically, but not what the actual image looks like in different orientations nor the relative position between these objects.



Figure 2.11: Two images showing a face, but with the second one, transformed by a 180 degree rotation and zoomed. CNNs trained to recognize the first image can have difficulty to recognize the second one as a face.

One standard operation standard in CNNs that helps it to detect images with position transformation is the pooling layers.

The building block of CNNs is the convolutional layer. These layers are responsible for identifying the features in a given image, like the curves, edges, sharpness, and color. In the end, the fully connected layers of the network will combine very high-level features and produce classification predictions. Max/average pooling operations are used together with successive convolutional layers throughout the network. The pooling operation reduces unnecessary information. Using this design, we can reduce the spatial size of the data flowing through the network and thus increase the field of view of the neurons in higher layers, allowing them to detect higher-order features in a broader region of the input image. With a broader field of view, pooling also helps the network to become less sensitive to positional changes. This is how max-pooling operations in CNNs help achieve state-of-the-art performance. CNNs work better than any model before them, but max-pooling is nevertheless losing valuable information. It averages or gets the maximum of regions of the image or feature vector, losing information during this process. Geoffrey Hinton stated that:

"The pooling operation used in Convolutional Neural Networks is a big mistake, and the fact that it works so well is a disaster!"

Figure 2.12 shows how a tensor can be pooled into a much smaller tensor using max pooling.

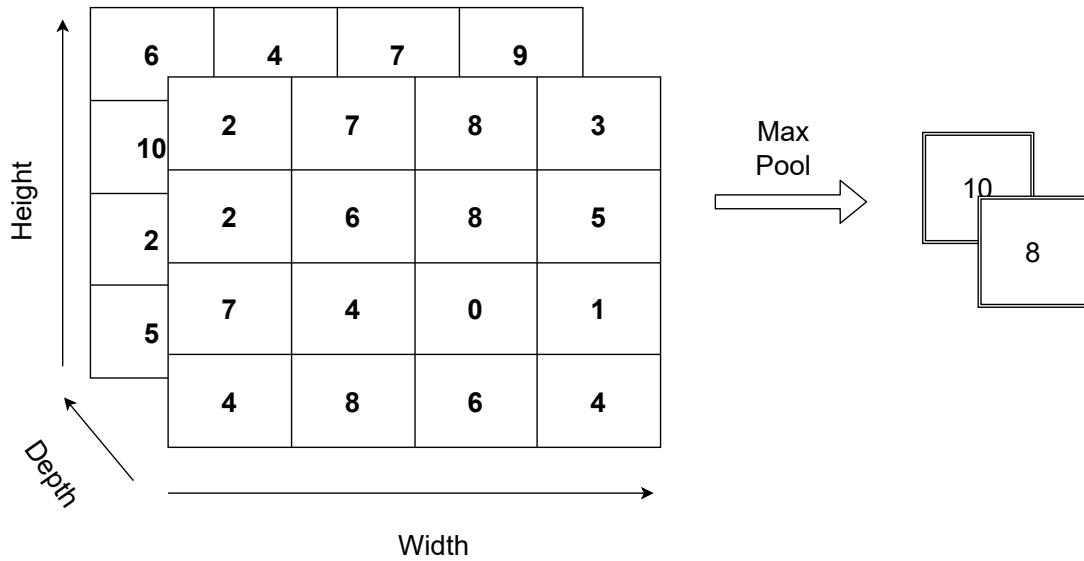


Figure 2.12: Example of a 3-dimension tensor with 32-values being pooled to a 2-values tensor using max pool.

Keeping in mind the problem with poolings, Hinton looked to a new approach that would not depend on lost information. To overcome the problem involving the rotational relationships in images, Sabour and Hinton drew inspiration from neuroscience. They explain that the brain is organized into modules, which can be considered capsules. With this in mind, they proposed capsule networks that incorporate dynamic routing algorithms to estimate features of objects like pose (position, size, orientation, deformation, velocity, albedo, hue, texture, and so on). This research was published in 2017 in their paper titled Dynamic Routing Between Capsules [1].

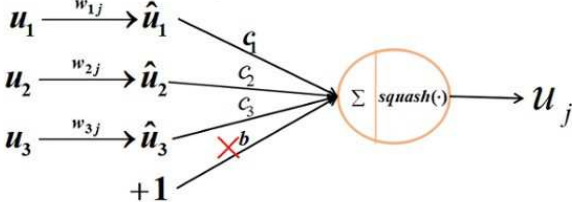
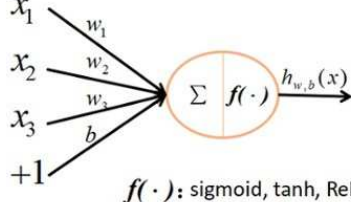
Unlike normal neurons, capsules perform their computations on their inputs and then "encapsulate" the results into a small vector of highly informative outputs. A capsule could be considered a replacement or substitute for your average artificial neuron; whereas artificial neuron deals with scalars, a capsule deals with vectors. This difference is illustrated in the table at Figure 2.13, where the equations of the capsule and traditional neuron are compared. For instance, we could summarize the steps taken by an artificial neuron as follows:

1. Multiply the input scalars with the weighted connections between the neurons.
2. Compute the weighted sum of the input scalars.
3. Apply an activation function (scalar nonlinearity) to get produce output.

On the other hand, a capsule goes through several steps in addition to those listed above to achieve the affine transformation (preserving co-linearity and ratio of distances) of the input. Here, the process is as follows:

1. Multiply the input vectors by the weight matrices (which encode spatial relationships between low-level and high-level features) (matrix multiplication).
2. Multiply the result by the weights.

3. Compute the weighted sum of the input vectors.
4. Apply an activation function (vector nonlinearity) to get produce output.

		capsule	vs.	traditional neuron
Input from low-level neuron/capsule		vector(u_i)		scalar(x_i)
Operation	Affine Transformation	$\hat{u}_{ji} = W_{ij} u_i$ (Eq. 2)		—
	Weighting	$s_j = \sum_i c_{ij} \hat{u}_{ji}$ (Eq. 2)		$a_j = \sum_{i=1}^3 W_i x_i + b$
	Sum			
	Non-linearity activation fun	$v_j = \frac{\ s_j\ ^2}{1 + \ s_j\ ^2} \frac{s_j}{\ s_j\ }$ (Eq. 1)		$h_{w,b}(x) = f(a_j)$
output		vector(v_i)		scalar(h)
				

Capsule = New Version Neuron!
vector in, vector out VS. scalar in, scalar out

Figure 2.13: Capsule vs Artificial Neuron mathematical equations. From the original Capsule Network paper [1].

1. Multiply the input vectors by the weight matrices

The input vectors represent either the initial input or input provided by a previous layer in the network. These vectors are first multiplied by the weight matrices. The weight matrix, as described previously, captures the spatial relationships. Say that one object is centered around another, and they are equally proportioned in size. The product of the input vector and the weight matrix will signify the high-level feature. For example, if the low-level features are nose, mouth, left eye, and right eye, then if the predictions of the four low-level features point to the same orientation and state of a face, a face will be what is predicted (as shown in Figure 2.14). This is what the "high-level" feature is.

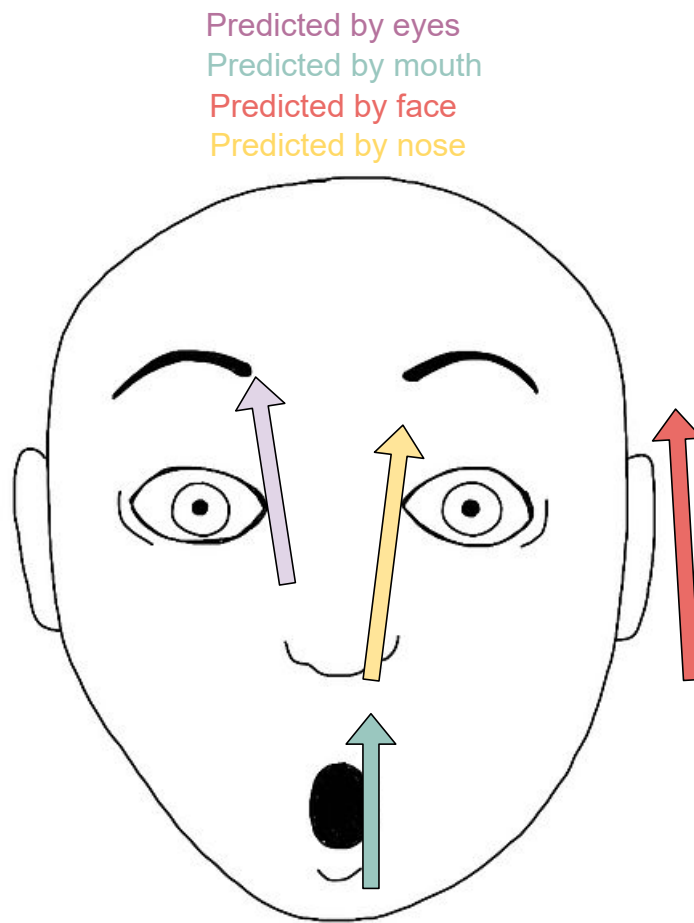


Figure 2.14: Example of vectors agreeing on face features positions.

2. Multiply the result by the weights

In this step, the outputs obtained from the previous step are multiplied by the network's weights. What can the weights be? In a usual Artificial Neural Network (ANN), the weights are adjusted based on the error rate, followed by backpropagation. However, this mechanism is not applied in a Capsule Network. Dynamic routing is what determines the modification of weights in a network. This defines the strategy for assigning weights to the neurons' connections.

A capsule network adjusts the weights such that a low-level capsule is strongly associated with high-level capsules in its proximity. The proximity measure is determined by the affine transformation step we discussed previously (Step 1). The distance between the outputs obtained from the affine transformation step and the dense clusters of the predictions of low-level capsules is computed (the dense clusters could be formed if the predictions made by the low-level capsules are similar, thus lying near each other). The high-level capsule that has the minimum distance between the cluster of already made predictions and the newly predicted one will have a higher weight, and the remaining capsules would be assigned lower weights, based on the distance metric. Figure 2.15 il-

illustrates this concept, where three examples of capsules are used to form a higher level capsule and they are weight based on as more cauterized with nearby capsule they are.

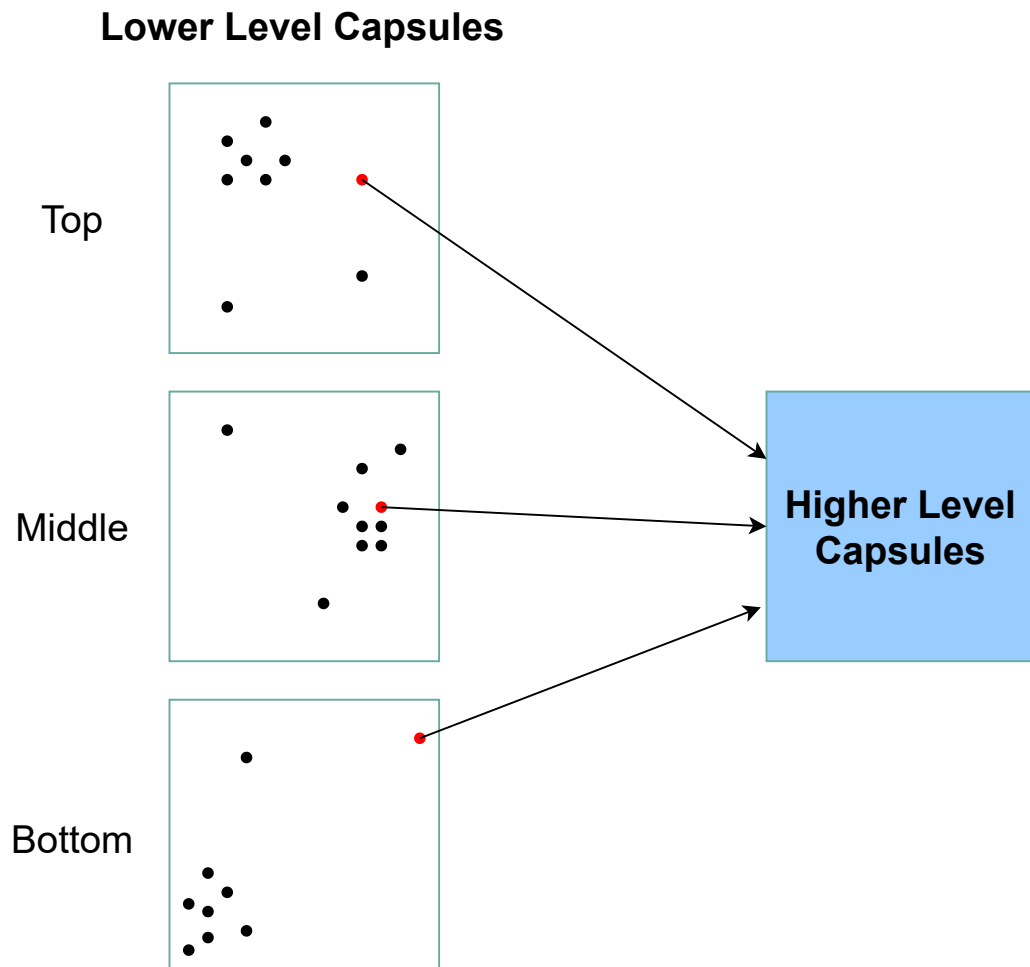


Figure 2.15: Visual representation of the dynamic routing agreement algorithm in a Capsule Network. The red highlighted capsule would be weighted by their distance to the clusters, so the order of weights would be: Middle, Top and Bottom.

In a nutshell, the essence of the dynamic routing algorithm could be seen as this: the lower level capsule will send its input to the higher level capsule that "agrees" with its input.

3. Compute the weighted sum of the input vectors

This sums up all the outputs obtained from the previous step.

4. Apply an activation function (vector nonlinearity) to produce the output

In a capsule network, the vector nonlinearity is obtained by "squashing" (*i.e.*, via an activation function) the output vector for it to have a length of 1 and a constant direction. The non-linearity function is given by Equation 2.8.

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \quad (2.8)$$

Where s_j is the output obtained from the previous step, and v_j is the output obtained after applying the nonlinearity. The left side of the equation performs additional squashing, while the right side performs unit scaling of the output vector.

On the whole, the dynamic routing algorithm is summarized in Algorithm 2.:

Algorithm 2 Dynamic Routing Algorithm.

```

1 def Routing( $u_{j|i}, r, l$ ):
2   for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l+1)$ :  $b_{ij} \leftarrow 0$ .
3   for  $it$  in range( $0, r$ ):
4     for all capsule  $i$  in layer  $l$ :  $c_i \leftarrow \text{softmax}(b_i)$ 
5     for all capsule  $j$  in layer  $(l+1)$ :  $s_j \leftarrow \sum_i c_{ij} u_{j|i}$ 
6     for all capsule  $j$  in layer  $(l+1)$ :  $v_j \leftarrow \text{squash}(s_j)$ 
7     for all capsules  $i, j$  in layers  $l, (l+1)$ :  $b_{ij} \leftarrow b_{ij} + u_{j|i} \times v_j$ 
8   return  $v_j$ 

```

- Line 1: This line defines the procedure of ROUTING, which takes affine transformed input (u), the number of routing iterations (r), and the layer number (l) as inputs.
- Line 2: b_{ij} is a temporary value that is used to initialize c_i in the end.
- Line 3: The for loop iterates r times.
- Line 4: The softmax function applied to b_i makes sure to output a non-negative c_i , where all the outputs sum to 1.
- Line 5: For every capsule in the next layer, the weighted sum is computed.
- Line 6: For every capsule in the next layer, the weighted sum is squashed.
- Line 7: The weights b_{ij} are updated here. $u_{j|i}$ denotes the input to the capsule from low-level capsule i , and v_j denotes the output of high-level capsule j .

The CapsNet architecture consists of an encoder and a decoder, where each has a set of three layers. An encoder has a convolutional layer, a PrimaryCaps layer, and a DigitCaps layer; the decoder has three fully connected layers. In the context of the MNIST dataset, for example, the architecture looks like the diagram in Figure 2.16.

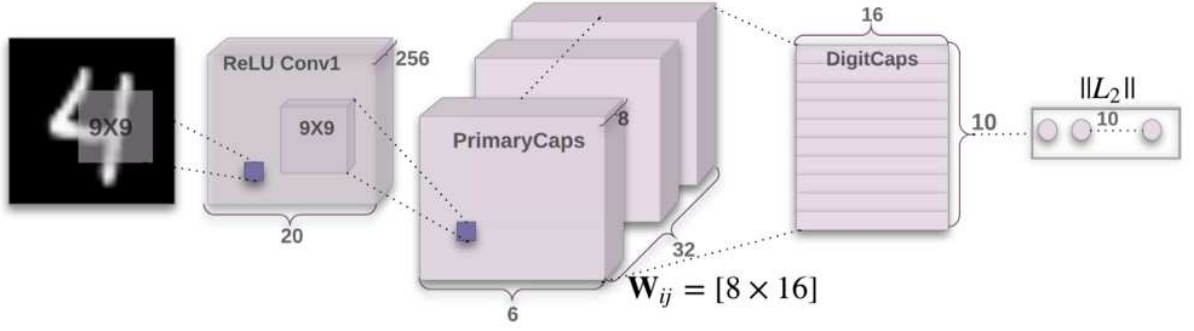


Figure 2.16: CapsNet Encoder Architecture. From the original Capsule Network paper [1].

An encoder has two convolutional layers and one fully connected layer. The first convolutional layer, Conv1, has 256 9×9 convolutional kernels with a stride of 1 and a ReLU activation function. This layer is responsible for converting the pixel intensities to the activities of local feature detectors, which are then fed to the PrimaryCaps layer. The PrimaryCaps layer is a convolutional layer that has 32 channels of convolutional 8-D capsules (each capsule has eight convolutional units with a 9×9 kernel and a stride of 2). Primary capsules perform inverse graphics, meaning they reverse-engineer the process of the actual image generation. The capsule applies eight $9 \times 9 \times 256$ kernels onto the $20 \times 20 \times 256$ input volume, which gives a $6 \times 6 \times 8$ output tensor. As there are 32 8-D capsules, the output would thus be of size $6 \times 6 \times 8 \times 32$. The DigitCaps layer has 16-D capsules per class, where each capsule receives input from the low-level capsule.

The 8×16 W_{ij} is the weight matrix used for affine transformation against each 8-D capsule. The routing mechanism discussed previously always exists between two capsule layers (PrimaryCaps and DigitCaps).

In the end, a reconstruction loss is used to encode the instantiation parameters. The loss is calculated for each training example against all output classes. The total loss is the sum of losses of all the digit capsules. The loss equation is given by Equation 2.9.

$$L_k = T_k \max(0, m^+ - ||v_k||) + \lambda(1 - T_k) \max(0, ||v_k - m^-||)^2 \quad (2.9)$$

Where:

- $T_k = 1$ if a digit of class k is present
- $m^+ = 0.9$
- $m^- = 0.9$
- v_k = vector obtained from DigitCaps layer

The first term of the equation represents the loss for a correct DigitCaps, and the second term represents the loss for an incorrect DigitCaps.

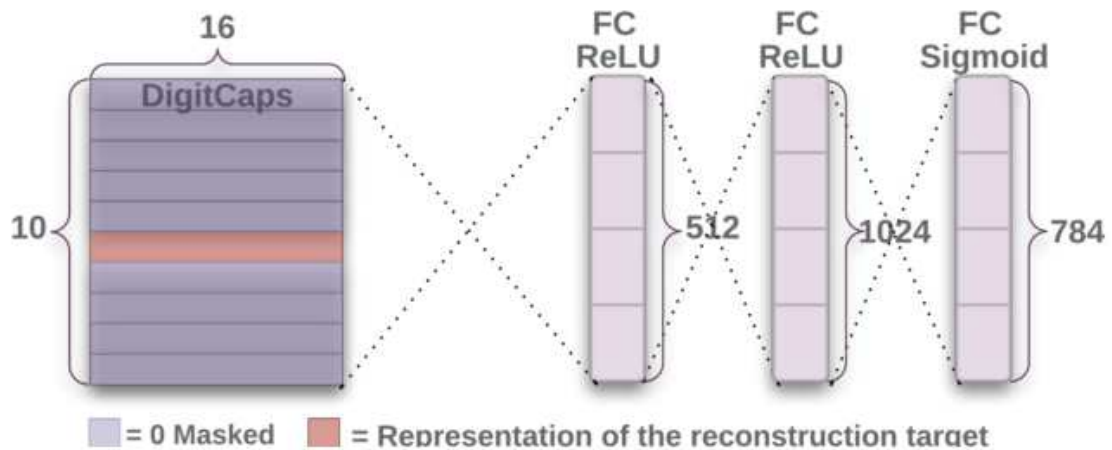


Figure 2.17: CapsNet Decoder Architecture. From the original Capsule Network paper [1].

A decoder takes the correct 16-D digit capsule and decodes it into an image. None of the incorrect digit capsules are taken into consideration. The loss is calculated by finding the Euclidean distance between the input image and the reconstructed image.

A decoder has three fully connected layers. The first layer has 512 neurons, the second has 1024 neurons, and the third has 784 neurons (giving 28×28 reconstructed images for MNIST).

A Capsule Network could be considered a “real-imitation” of the human brain. Unlike Convolutional Neural Networks, which do not evaluate the spatial relationships in the given data, capsule networks consider the orientation of parts in an image as a critical part of data analysis. They examine these hierarchical relationships to identify images better. The inverse graphics in biological face processing (that allows us to recognize faces independent of their rotation) that our brains use is imitated here to build a hierarchical representation of an image and match it with what the network has learned. Though it is not yet computationally efficient, there does seem to be an accuracy boost beneficial in tackling real-world scenarios. The Dynamic Routing of Capsules is what makes all of this possible. It employs an unusual strategy of updating the weights in a network, thus avoiding the pooling operation.

Chapter 3

Related Work

Convolutional Neural Network (CNN) is a DNN [12] commonly used when working with images. CNNs have already achieved state-of-art results in image and video recognition, image classification, and medical image analysis. However, these networks have difficulties with location invariance and loss of location information, *e.g.*, one CNN which can recognize faces could also mistakenly recognize an image with eyes, mouth, and nose at random positions as a face, not understanding that there is an essential spatial relationship between the composing elements. To address this, many different new DNN approaches were proposed, including the notion of capsules proposed by Hinton, Krizhevsky, and Wang in 2011 [25].

Capsule Networks, also known as CapsNets, do not work by representing neurons as simple scalars (as in regular CNNs) but as vectors to encode spatial relationships. Later in 2017, an efficient and realistic training algorithm for such networks was proposed [1]. The algorithm, named Dynamic Routing, dynamically chooses activation paths between capsules from one layer to another, calculating the vectors from the next layer based on a mean from dynamically selected vectors from all previous layers. CapsNet [1] produces a set of Primary Capsules (PCs) by applying two convolutional steps to the original image and splitting it into vectors. Each of these PCs (vectors), identified as u_i , is multiplied by a weight matrix W_i and finally, a final set of capsules, the digit capsules, is created using the dynamic routing algorithm. Each of these digit capsule vectors represents one of the classes in the classification problem, and the vector's length encodes the probability of the class. The digit capsule can also be used to reconstruct the image like an auto-encoder.

Since its proposal, in 2017, CapsNets has been tested in several datasets, different applications, and scenarios. This chapter presets a compilation of papers which uses CapsNets in for different applications (Section 3.1), which explored enhancements to the original CapsNet architecture (Section 3.2), and which explored improving the CapsNet execution time (Section 3.2.1).

3.1 Applications

Already in the first year, many applications appeared such as natural language processing [26, 27], breast cancer histology classification [28], detection of aggression and toxic

comments in social networks [29], classification of hyperspectral imaging [30], among many others. Some of the first papers in the field were categorized by Patrick *et al.* [31] survey.

Shahroudjeh, Mohammadi, and Plataniotis [32] presented an analysis of the explainability of CapsNet, showing that it has properties to help understand and explain its behavior. Jaiswal *et al.* [33] used the CapsNet in a Generative Adversarial Network (GAN) and showed that it could achieve lower error rates than the simple CNN. Ren and Lu [27] showed that CapsNet could be used for text classification and showed how to adapt the compositional coding mechanism to the CapsNet architecture. Jimenez-Sanchez, Albarqouni, and Mateus [34] tested CapsNet for Medical Imaging Data Challenges showing that it can achieve good performance even when having less trainable parameters than the tested counterpart CNNs. Mobiny and Nguyen [35] tested the performance of CapsNet for lung cancer screening and showed that it could outperform CNNs mainly when the training set was small. A similar result was achieved by Kim *et al.* in traffic speed prediction [36] with CapsNet outperforming traditional CNNs approaches. Mukhometzianov and Carrillo [37] also tested CapsNet with multiple image datasets comparing with CNNs. All of them argue that CapsNet still requires higher training times than other CNNs, showing why it is important to investigate ways to improve its parallelization and scalability.

Nguyen, Yamagishi, and Echizen [38] evaluated the use of CapsNet in the detection of fake images and videos, showing that it can achieve similar results as state-of-art CNNs with the same amount of parameters. The authors also showed how capsules could be used to understand the decisions. Li *et al.* [39] used CapsNet to recognize rice images by crewless aerial vehicles and described that it was a convenient choice in terms of understatement of the network decisions. Singh *et al.* [40] proposed the use of CapsNet in Very Low Resolution (VLR) image datasets showing that it can achieve state-of-art results. To overcome the challenges of limited information content in VLR images, the authors used an HR-anchor loss and a proposed targeted reconstruction loss. The proposed losses use high-resolution images as auxiliary data during training to direct feature learning. Zhu *et al.* [41] applied CapsNet to the bearing fault diagnosis problem by using capsules in features extracted from convolution layers, a network they named convolutional capsule network. Chen and Qian [42] showed that CapsNet could be used to extract aspect-level sentiment classification of sentences from document-level labeled data, transferring knowledge from one level to another. Li *et al.* [43] applied a CapsNet architecture to seven real-word datasets of users' preferences and achieved state-of-art results with a higher level of explainability of what the user likes or dislikes. Zhang *et al.* [44] describes that capsules are better than usual CNNs in the extraction of highly overlapped features in NLP relation extraction.

Xiang *et al.* [45] presented a matrix capsule that outperformed their previous vector capsules and baselines in image classification. Kwabena, Weyori, and Migly [46] modified hyperparameters of the original CapsNet architecture to obtain better results in plant disease datasets where a set of complex and varied backgrounds are present. The authors proposed the use of local binary pattern in oppose to CNN layers, the use of sigmoid in oppose to SoftMax, and the use of k-means in oppose to the original dynamic routing. Afshar *et al.* [47] used CapsNet with x-ray image dataset of COVID-19 patients to diagnosis the disease. CapsNet showed useful as it can learn with smaller datasets than traditional

CNNs, thus fitting the emergency scenario of the COVID outbreak. Afshar, Mohammadi, and Plataniotis [48] proposed the use of a Bayesian approach to improve accuracy and give better interpretable results for brain tumor classification using CapsNet. Shamsolmoali *et al.* [49] showed that CapsNet could be used as generative adversarial networks to augment imbalanced image datasets.

These CapsNet applications are listed and summarized in Table 3.1.

Table 3.1: Summary of applications of CapsNet in the literature discussed in this section.

Paper Title	Pub. Year	Used Dataset	Result
Hyperspectral image classification with capsule network using limited training samples [30]	2018	PaviaU (PU) and SalinasA	Higher accuracy than CNN
Capsule networks against medical imaging data challenges [34]	2018	MNIST, Fashion-MNIST and medical (histological and retina images)	Same accuracy as CNN but needing less data
CapsuleGAN: Generative adversarial capsule network [33]	2018	MNIST and CIFAR-10	Better than CNN in Generative Metric
Compositional coding capsule network with k-means routing for text classification [27]	2018	Eight challenging text classification datasets	Same accuracy as state of art but with less parameters
Convolutional capsule network for classification of breast cancer histology images [28]	2018	Images of breast tissue biopsy	Same accuracy as CNN
Identifying aggression and toxicity in comments using capsule network [29]	2018	Kaggle-toxic	Higher accuracy than CNN
A Capsule Network for Traffic Speed Prediction in Complex Road Networks [36]	2018	Real traffic speed data measured in the Santander city of Spain	Higher accuracy than CNN
Fast capsnet for lung cancer screening [35]	2018	lung nodules from computed tomography (CT) scans	Higher accuracy than CNN
CapsNet comparative performance evaluation for image classification [37]	2018	Images of faces, traffic signs, and everyday objects	Worst accuracy than CNN and much Slower
Use of a capsule network to detect fake images and videos [38]	2019	FaceForensics++	Higher accuracy than CNN
The recognition of rice images by UAV based on capsule network [39]	2019	Rice images captured by UAV	Better explainability
A capsule network for recommendation and explaining what you like and dislike [43]	2019	Yelp16-17, Beer, and other 5	Higher accuracy than CNN
Multi-labeled relation extraction with attentive capsule network [44]	2019	NYT-10 and SemEval-2010	Better extraction of overlapping features
Transfer capsule network for aspect level sentiment classification [42]	2019	SemEval	Better accuracy
BayesCap: A Bayesian Approach to Brain Tumor Classification Using Capsule Networks [48]	2020	Brain cancer dataset	Same accuracy
Covid-caps: A capsule network-based framework for identification of covid-19 cases from x-ray images [47]	2020	COVID-19 X-ray dataset	Better accuracy
Exploring the performance of LBP-capsule networks with K-Means routing on complex images [46]	2020	MNIST, fashion-MNIST, CIFAR-10, tomato, maize, and citrus datasets	Same accuracy as CNN with less parameters
Matrix Capsule Convolutional Projection for Deep Feature Learning [45]	2020	Cifar10, Cifar100 and SVHN	Higher accuracy than CNN
Imbalanced data learning by minority class augmentation using capsule adversarial networks [49]	2021	Cifar10, CelebA, Fashion-MNIST, GTSRB and MNIST	Same accuracy as CNN

3.2 Architectural Enhancement

Capsule network with the dynamic routing algorithm was shown to have advantages such as needing smaller training sets and location invariance. However, it also has drawbacks compared to traditional CNNs, such as slower execution and lower accuracy for complex datasets. Since its initial publication, multiple improvements and studies have been proposed to mitigate or solve these drawbacks, and the CapsNet architectural concept has been evolving.

Xiang *et al.* [50] proposed the use of a multi-scale feature extraction that is used to encode the hierarchy of features on the multi-dimensional primary capsule. Moreover, they showed that dropout regularization could be used in capsules to improve the network robustness.

As the original CapsNet architecture is not deep, some authors proposed solutions to increase its deepness, maintaining its properties [51, 52, 53]. All of them show that increasing the deepness of a CapsNet while increasing or maintaining its equivariance property is necessary to change and improve the original network and the routing algorithm.

Jeong, Lee, and Kim [54] introduced a new pruning layer to remove irrelevant capsules before the dynamic routing algorithm. Moreover, to maintain the part-whole spatial relationships, they also introduced a new layer named ladder. Unlike the capsule network adopting the routing-by-agreement, the ladder capsule network uses backpropagation from a loss function to reconstruct the lower-level capsule outputs from higher-level capsules. The ladder capsule network learns an equivariant representation and improves the capability to extrapolate or generalize to pose variations.

Punjabi, Schmid, and Katsaggelos [55] did an evaluation showing the features and differences between CapsNets and regular CNNs, showing that for many features, including invariance, CapsNet has advantages over CNNs. However, these advantages only appear when the network is trained using the image reconstruction mechanism (auto-encoder). Chen and Liu [53] reduced the number of training parameters considerably when deepening the network. Regarding architectural improvements, Jia and Huang [56] proposed the use of a hierarchical architecture that uses residual convolutional layers and the position-wise dot product to improve CapsNet over complex data with complex background. Also, Yang *et al.* [57] proposed the RS-CapsNet, which improves the overall properties of the original CapsNet while reducing to 65% the number of parameters in the model. Edraiki, Rahnavard, and Shah [58] presented a Subspace Capsule Network. Instead of simply grouping neurons to create capsules. A capsule is created by projecting an input feature vector from a lower layer onto the capsule subspace using a learnable transformation. This transformation finds the degree of alignment of the input with the properties modeled by the capsule subspace. Chen and Liu [53] combined the U-Net architecture features to the CapsNet network and proposed a mask to the dynamic routing algorithm to allow such combination. They showed that their new approach reduced trainable parameters while preserving advantages of image reconstruction and equivariance mechanism. Jia and Huang [56] proposed a hierarchical architecture that uses residual convolutional layers and the position-wise dot product to build diverse enhanced primary capsules with various scales of images for complex data. They combined the Diverse Capsule Network

structure with the Spatial Group-wise Enhance (SGE) mechanism achieving better results with fewer parameters.

Jampour, Abbaasi, and Javidi [59] proposed a new regularization term for CapsNet and also showed that it can be used together with ResNet [14] to achieve better results. Sun *et al.* [60] improved the original CapsNet by using multistage separable convolutions and a dense capsule architecture. They were able to drastically reduce the number of parameters while improving the network accuracy.

Concerning the Dynamic Routing algorithm, many improvements were also proposed. Ren [61] shows that the initialization of the algorithm is essential to its final performance and describes how to select better initial values, and Ding *et al.* [61] proposed a new supervised algorithm to be used instead of the dynamic algorithm showing improvements over real-complex data. Li *et al.* [43] describes a bi-Agreement routing algorithm mechanism where an intra and an inter-capsule agreement coefficient is calculated. The geometric mean of both is then used as the capsule agreement in the dynamic routing. Mandal *et al.* [62] introduced a routing algorithm with two phases that computes agreements between neurons at various layers for micro and macro-level features, following a hierarchical learning paradigm—in this way, improving the representation capability of the network.

These CapsNet enhancements are listed and summarized in Table 3.2.

Table 3.2: Summary of enhancements proposed to CapsNet found in the literature and discussed in this section.

Title	Pub. Year	Enhancement	Result
MS-CapsNet: A Novel Multi-Scale Capsule Network [50]	2018	Multi-scale Feature Extraction	Improves accuracy
Deep Tensor Capsule Network [51]	2018	New routing algorithm	Allows deeper networks
Building Deep, Equivariant Capsule Networks [52]	2019	Equivariant routing mechanism based on degree-centrality	Increases transformation-robustness
Grouping Capsules Based Different Types [61]	2019	Improves 'c' constants initialization in dynamic routing	Improves accuracy
A capsule network for recommendation and explaining what you like and dislike [43]	2019	New routing algorithm with Bi-Agreement mechanism	Improves accuracy
Ladder capsule network [54]	2019	Adds a new layer between capsules: the ladder layer	Improves the capability to extrapolate or generalize to pose variations
DE-CapsNet: A Diverse Enhanced Capsule Network with Disperse Dynamic Routing [56]	2020	Adds diverse enhanced primary capsules	Reduces the number of parameters needed
RS-CapsNet: An Advanced Capsule Network [57]	2020	Adds Res2Net block and linear combination between capsules	Improves accuracy and pose representation
Subspace capsule network [58]	2020	Projects the input vector from a lower layer onto the capsule subspace using a learnable transformation	Improves accuracy
CapsNet Regularization and its Conjugation with ResNet for Signature Identification [59]	2021	Adds regularization mechanisms	Reduces needed training set
Two-phase Dynamic Routing for Micro and Macro-level Equivariance in Multi-Column Capsule Networks [62]	2021	Uses a two-phase routing agreement, one from micro and other from macro-level feature layers	Improves accuracy

3.2.1 Execution Time Improvement on CapsNet

One of the CapsNet main drawbacks is its higher execution time compared to CNNs. Approaches such as the ones from Shiri, Sharifi, and Baniasadi try to reduce these bottlenecks by reducing the number of training parameters of the network [63].

Another approach that was followed by our research is to remove data dependence in CapsNet data flow, improving its parallelism. That culminated in the proposal of the Multi-Lane CapsNet, which is presented and explored in this dissertation.

Several approaches to the distributed model parallelization of Deep Neural Networks (DNN) have concentrated in their depth dimension [5, 6, 7], but DNNs can also be organized in a way to be parallelized along their width dimension [8]. The DNN architecture may be organized into distinct Neural Network lanes [9]. This creates separable and resource-efficient data-independent paths in the network that can learn different features or add resilience to the network. Examples of Neural Networks with lanes are the Google Inception [64, 65] and the Multi-Lane Capsule Network (MLCN) [9], proposed in this work. As these lanes are data-independent, they can be (1) processed in parallel and (2)

specialized for distinct computational targets (CPUs, GPUs, FPGAs, and cloud), as well as resource-constrained mobile and IoT targets, leading to opportunities and challenges. Our recent research focused on Multi-Lane Capsule Networks (MLCN), a separable and resource-efficient organization of Capsule Networks (CapsNet) that allows parallel processing while achieving high accuracy at a reduced cost. Table 3.3 shows results from MLCN in comparison with the original CapsNet. MLCN achieves similar accuracy but with a significant speedup provided by the lane organization with a similar number of parameters. Initial experiments were performed in single GPU environments but with highly parallel lanes exploring how MLCN scales with more GPUs is interesting. Here we present a first comprehensive study of the scalability and efficiency of MLCN for multi-GPU systems.

Table 3.3: Comparison between original CapsNet and MLCN performance on CIFAR-10 and Fashion-MNIST datasets.

Network/set	# of lanes	lane's Width	Params.	Train Time (sec./epoch)	Accuracy
CIFAR-10:					
CapsNet	-	-	11k	240	66.36%
MlcN2	4	4	5k	53	69.05%
MlcN2	32	2	14k	204	75.18%
Fashion-MNIST:					
CapsNet	-	-	8k	220	91.30%
MlcN2	2	4	3.6k	20	91.01%
MlcN2	8	4	10.6k	92	92.63%

Moreover, the lanes do not necessarily need to have the exact sizes or shapes and may even learn different features of the given task. This implies that each distinct lane may be better suitable for a distinct hardware substrate. Further, each lane may tolerate different impacts from various optimizations (such as quantization). Thus, given a set of lanes, L , and a set of hardware, H , there is an optimal pair (l, h) for $l \in L$ and $h \in H$ and an optimal sequence of lane optimizations for each pair (l, d) of lane and hardware.

There are two key advantages of this organization over the original CapsNet architecture. First, it allows parallelism of the execution, as each set of Primary Capsules (PCs) is constructed independently, improving performance and allowing training and deployment on distributed environments. Second, it improves the explainability of the network by associating different features of the image to each lane.

Later, Chang and Liu proposed an improvement over MLCN, named MLSCN [66]. This new model improves accuracy while maintaining the parallelism and scalability characteristics of MLCN. As MLSCN work was published concurrently with the development

of this work, we did not use MLSCN in your experiments, although given its characteristics, we believe the performance results would be similar or possibly the same. Similarly, Canqun *et al.* [50] proposed the Multi-Scale CapsNet (MS-CapsNet). They introduced a fixed division of the CapsNet network limited to three “lanes ” (they did not name or explore the division concept), each with a different number of convolutions. The Path Capsule Networks by Amer and Maul [67] (Path-Capsnet) explore the parallelism of CapsNets by splitting the network such that each path or lane is responsible for computing each digitcaps or a primary capsule entirely, unlike the computation of different dimensions/features in MLCN.

Since the proposal of the MLCN, it has been used by others for different applications, such as complex image classification [68] and in X-ray images collected from patients tested for COVID-19 [69]. Finally, Table 3.4 shows the three main performance improvement researches found in the literature.

Table 3.4: Summary of execution time improvements to CapsNet found in the literature and discussed in this section.

Title	Pub. Year	Architectural enhancement?	Improves Parallelism?	Improves Exec. Time?
High performance training of deep neural networks using pipelined hardware acceleration and distributed memory [6]	2018	No	No	Yes - by using pipeline parallelism
MS-CapsNet: A Novel Multi-Scale Capsule Network [50]	2018	Yes	Possibly - but it is not explored	Yes
The Multi-Lane Capsule Network [9]	2019	Yes	Yes	Yes

Chapter 4

Accelerating Capsule Networks with Lanes

The original version of CapsNet [1] produces a set of N Primary Capsules (PCs) by applying two convolutional steps to the original image. Each of these PCs, identified as u_i , is multiplied by a weight matrix W_i and finally, a final set of capsules, the digit capsules, is created using the dynamic routing algorithm. Each of these digit capsule vectors represents one of the classes in the classification problem, and the vector's length encodes the probability of the class being the one in the input image. However, beyond just encoding the probability of a class, each vector also contains information to reconstruct the original image, with distinct dimensions of the vector representing different features of the image. With this in mind, we propose to split the original CapsNet architecture¹ (Figure 4.1), dividing the PCs into independent sets that we call lanes. Each of these sets of PCs, a lane, is responsible for one of the dimensions in the final digit capsules.

The number of PCs per lane may vary, as well as the way they are computed. In the original CapsNet, two 2D convolutions are applied to the input image and then reshaped to produce the PCs. More convolutions may be applied, what we call the *depth* of a lane, or more filters can be used per convolution generating more capsules, what we call the *width* of a lane. Further, distinct dimensions of a final digit capsule can be generated by lanes with different configurations (and thus distinct computational requirements).

There are two key advantages of this organization over the original CapsNet architecture. First, it allows parallelism of the execution, as each set of PCs is constructed independently, improving performance and allowing training and deployment on distributed environments. Second, it improves the explainability of the network by associating different features of the image to each lane.

¹source code in <https://github.com/vandersonmr/lanes-capsnet>

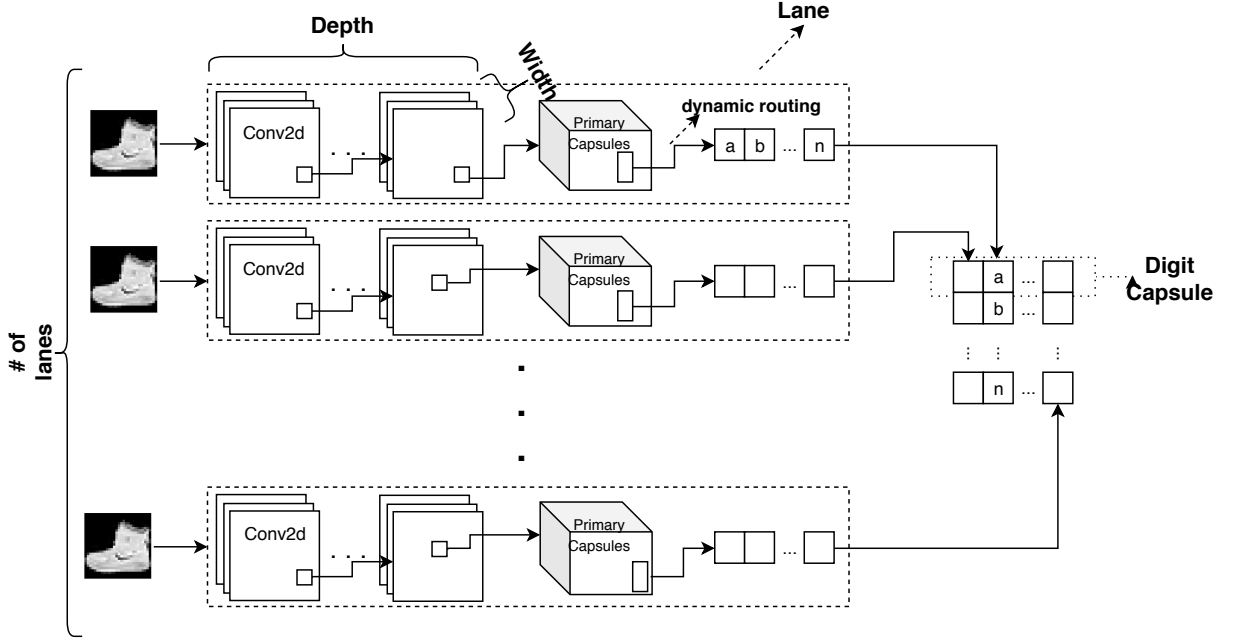


Figure 4.1: Diagram of the Multi-Lane Capsule Network architecture.

Such organization of the capsules also resembles the stacking ensemble learning mechanism (see Section 2.3.2). Each lane here acts as an independent learner that is put together with the routing algorithm to construct the final output, the digit capsule.

4.1 CapsNet Parallelization

A DNN can be parallelized in different ways, and normally finding the best way for a given network is a complex and hard task [70]. Three of the most common techniques used are data parallelism, model parallelism, and pipelining.

The first, data parallelism, splits the data which is going to be computed. It divides the input batch into smaller batches for each compute unit, synchronizing at the end of the batch. Although simple and straightforward, it can only scale by increasing the batch size because dividing too much can result in small computation for each compute unit and too frequent synchronization. Nonetheless, varying the batch size impacts the accuracy, causing a trade-off between accuracy and speedup.

Another possibility is partitioning the computational network graph itself. However, it is not always trivial to find partitions that offer good performance. Normally, if two data-dependent operations are scheduled into two distinct computation units, it results in increased communication overhead. Moreover, the implementation details of this kind of model partitioning and communication in currently available frameworks are not trivial. This approach is known as model parallelism.

Lastly, but not less important, pipelining splits the network into levels that can compute different data at the same time in a pipeline approach. It is normally the approach used in high-performance scenarios.

These are not the only techniques to distribute the training and inference of DNNs, and they are not mutually exclusive and can be used together [8]. Related to MLCN, we

tested its capability of allowing easy model parallelism and compared it to the common approach that is data parallelism. Of course, for huge MLCN networks, the pipeline could also be used, but we focus on showing how being able to facilitate the use of model parallelism can bring many advantages over the use of data parallelism alone. This same advantage can be easily extended by adding pipelining per lane, which remains to be explored in future work.

4.2 Experimental Setup

We tested our approach with the Fashion-MNIST and CIFAR-10 datasets. These are the dataset also used in the original CapsNet paper. Each experiment is performed ten times, and we present the average of the results. There was no significant variance in the results; therefore, we only present the means. Our experimental baseline refers to the original CapsNet configuration tested with the MNIST dataset [1]. The baseline with MNIST creates 1152 PCs with dimension 8 and 10 digit capsules with dimension 16. We also use PCs with eight dimensions and vary the number of PCs. We call a 1-size lane a lane that creates 72 PCs (same proportion of PCs per dimension as in the baseline) and a k -size lane a lane with $k \times 72$ PCs. The experiment also varies the number of lanes (dimensions of the digit capsule), testing networks with 2, 4, 8, 16, and 32 lanes. All reported experiments in this section were performed on a P100 NVIDIA GPU with 16GB of RAM, and the training was performed with 20 epochs.

We tested two variations of lane configurations: the first, **Mlcn1**, has the same configurations as the baseline CapsNet, and the second, **Mlcn2**, includes differences that were found to increase the performance of our architecture significantly. Details of both are described below. In both, *width* is a parameter that can vary (explored in Section V), and it is the width described in Figure 1.

- **Mlcn1:** the image is first processed by one convolutional layer with $16 \times width$ kernels, each one with kernel size 9×9 and stride 1. The output from the first convolution is used as input to another convolutional layer with $16 \times width$ kernels, each kernel with size 9×9 and stride 2. The output of the second convolution layer is then reshaped into $72 \times width$ PCs with eight dimensions. Finally, using all PCs, Dynamic Routing produces one vector with a dimension equal to the number of classification classes in the problem.
- **Mlcn2:** the image is first processed by one 1×1 separable convolutional layer with $4 \times width$ kernels, followed by one convolutional layer with kernel size 9×9 and stride one and $8 \times width$ kernels. The output of the last convolution is used as input to one last convolution with $8 \times width$ kernels, each kernel with kernel size equal to 9×9 and strides of 2. The output of the last convolution is then reshaped as in Mlcn1, outputting $72 \times width$ PCs with eight dimensions. Finally, using all PCs, Dynamic Routing produces one vector with a dimension equal to the number of classification classes in the problem.

4.2.1 Dropout and Regularization in MLCN

In both configurations, we notice that, usually, a subset of lanes would actually be enough for the reconstruction and for the classification. At a certain point, additional lanes stopped providing useful information, and new lanes just would produce similar results, adding no new information to the solution or even over-fitting. To mitigate this effect, we investigated a dropout approach. During the training process, we would discard 10% of the lanes randomly and only use the result of the other 90% to calculate the final classification (for reconstruction, all lanes were always used). This forces all lanes to contribute useful information to the solution of the problem. However, as we show and discuss in Section 4.3.3, this made the training process more laborious and caused all lanes to learn similar features.

4.3 Experimental Results

Digit capsule vectors, which are constructed by concatenating the output of all lanes, should have encoded information to entirely reconstruct the input image. During the training process, the longest digit capsule (its length encodes one class probability) is used as input to a fully connected Neural Network with two layers of 512 and 1024 neurons, which learns to reconstruct the original image. As seen in Figure 4.2, the output of 16 lanes (Mlc1) can be used to reconstruct with high-fidelity the input image from the Fashion-MNIST dataset. Therefore, it shows that dividing CapsNet into lanes does not prevent it from converging and learning image characteristics.

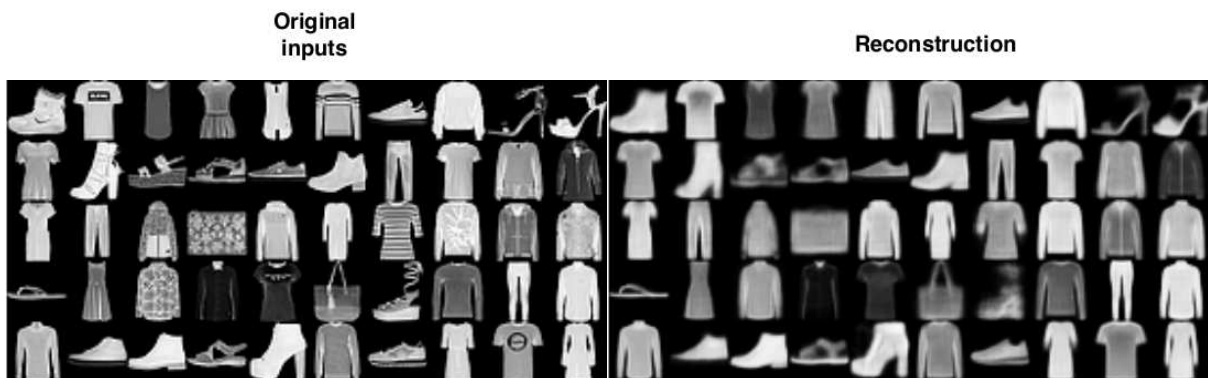


Figure 4.2: Reconstruction of the original Fashion-MNIST inputs using the capsule output by the MLCN network after trained. The reconstruction is done using a fully-connected network.

Figure 4.3 shows the effect on the reconstruction when we vary the output of each lane by adding -0.25 , -0.2 , -0.15 , -0.1 , -0.05 , 0 , 0.05 , 0.1 , 0.15 , 0.2 , and 0.25 to each dimension of the vector, one dimension at a time. We can observe the effect of this change on different features of the original image. Two key points from this: first, as each lane is entirely independent, we may know that, for instance, lane5 and its associated convolutions are used to extract the size of the jacket. Further, we noticed that similar properties were

being extracted from different classes. So, for example, lane5 also extracts the size of the shoes. Second, we notice that adding more than five lanes for the MNIST dataset would result in lanes with no impact on the reconstruction and produce essentially similar outputs. Therefore, these lanes would not help to solve the classification problem for that dataset. Larger and more complex datasets would benefit from larger networks.

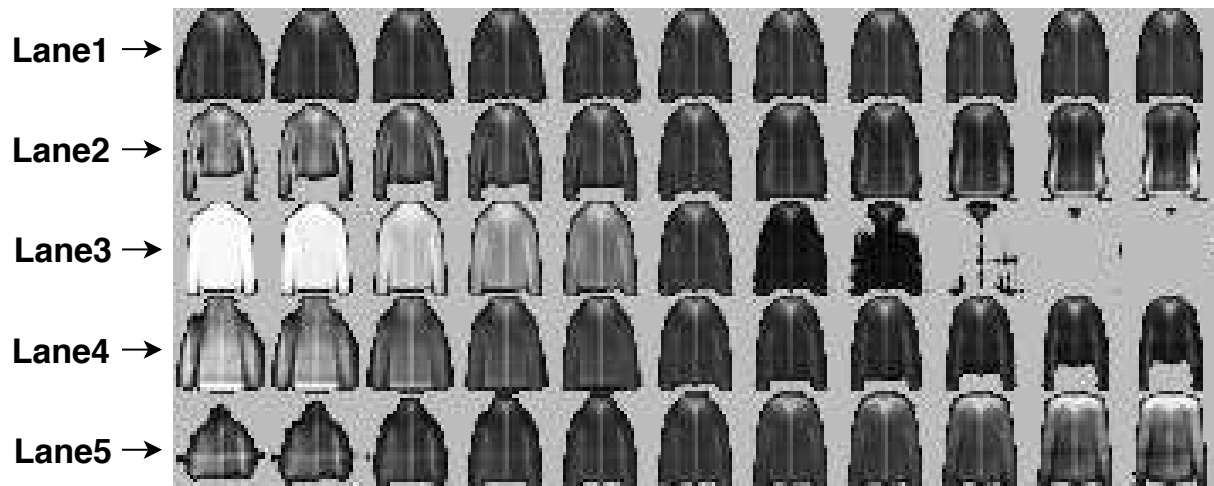


Figure 4.3: Impact on the reconstruction of the input when synthetically varying each of the lanes output. It shows that different lanes are learning different features of the image, such as the color in Lane3.

Although the second configuration, Mlcn2, improved the classification problem and increased the number of useful lanes to solve a specific problem (as we show in the next subsection), it did not have any visible impact on the reconstructed images; thus, we only present images from Mlcn1.

4.3.1 Number of Lanes vs. Model Accuracy

As we noticed from the reconstructed images, adding more lanes does not always improve the model. After a certain limit, new lanes stop learning new features. For example, in Fashion-MNIST (Figures 4.4a and 4.4b), adding more lanes was not always beneficial. We observed that when using the first type of lanes (4.4a), 2 or 4 lanes were better than 8 or 16. When exploring why, we noticed that in the Fashion-MNIST case, even when using more, only four lanes were learning-rich features. So, adding more lanes would not result in better features being extracted, but it would make it more difficult to train. That is because increasing the final vector dimension size (what is defined by the number of lanes) increases the training difficulty level, making it slower. Thus, after a limit, it is not only useless to add new lanes, but it actually makes the training slower. We also tested using Mlcn2 (4.4b), and the scenario improved (now with eight lanes limit for Fashion-MNIST), but it is still the case that using more lanes, such as 16 or 32 lanes, was not beneficial.

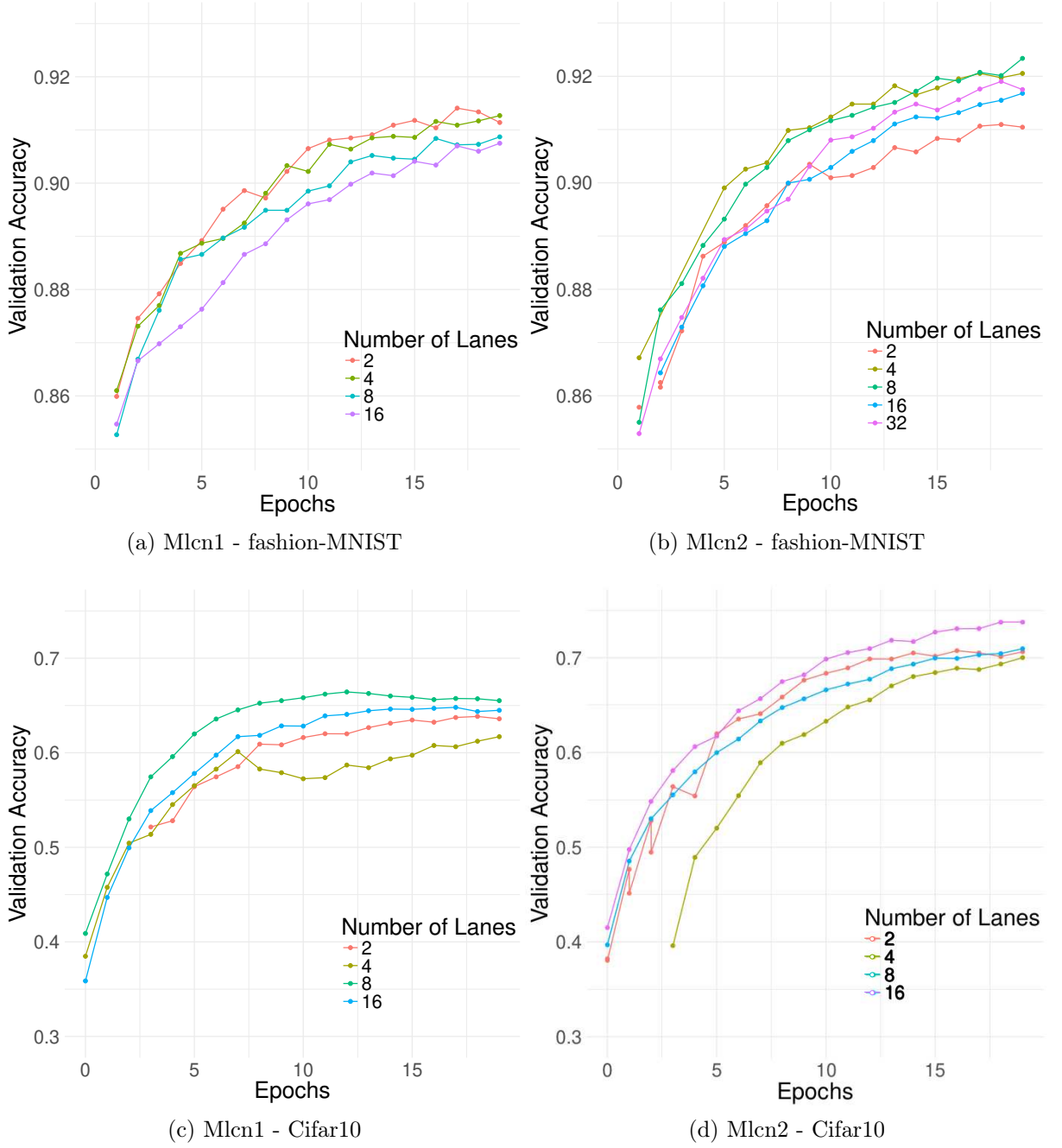


Figure 4.4: Accuracy on the Fashion-MNIST and the CIFAR-10 datasets by MLCN (1 and 2) networks with different number of lanes.

One thing to notice is that the number of lanes which will maximize the network performance is not only related to the lanes configurations but also to the dataset. While Fashion-MNIST appears to only need four lanes to represent its data with Mlcn1, CIFAR-10, a more complex dataset, requires 8 (4.4c) and with Mlcn2, it actually benefits from using 32 lanes (4.4d). Another important point from Figures 4.4a, 4.4b, 4.4c, and 4.4d is that MLCN achieved better accuracy levels when using lanes of the second type. When we compare the accuracy achieved by the two lane types for both datasets with the original

CapsNet (Table 4.1), we see that Mlcn1 achieves similar accuracy rates as the CapsNet, but Mlcn2 is better.

4.3.2 Lane’s Width vs. Model Accuracy

We also experiment with varying the lanes width. In other words, increasing the number of convolution kernels, and in this way, adding more PCs. See results in Figure 4.5. For all the results presented above, we used lanes of width four because it demonstrated for both datasets to be the better choice. We observed that increasing the lane’s width increases the performance of the network, but for lanes deeper than 4, it shows that this improvement starts to cease and the training rapidly overfits given the large number of PCs.

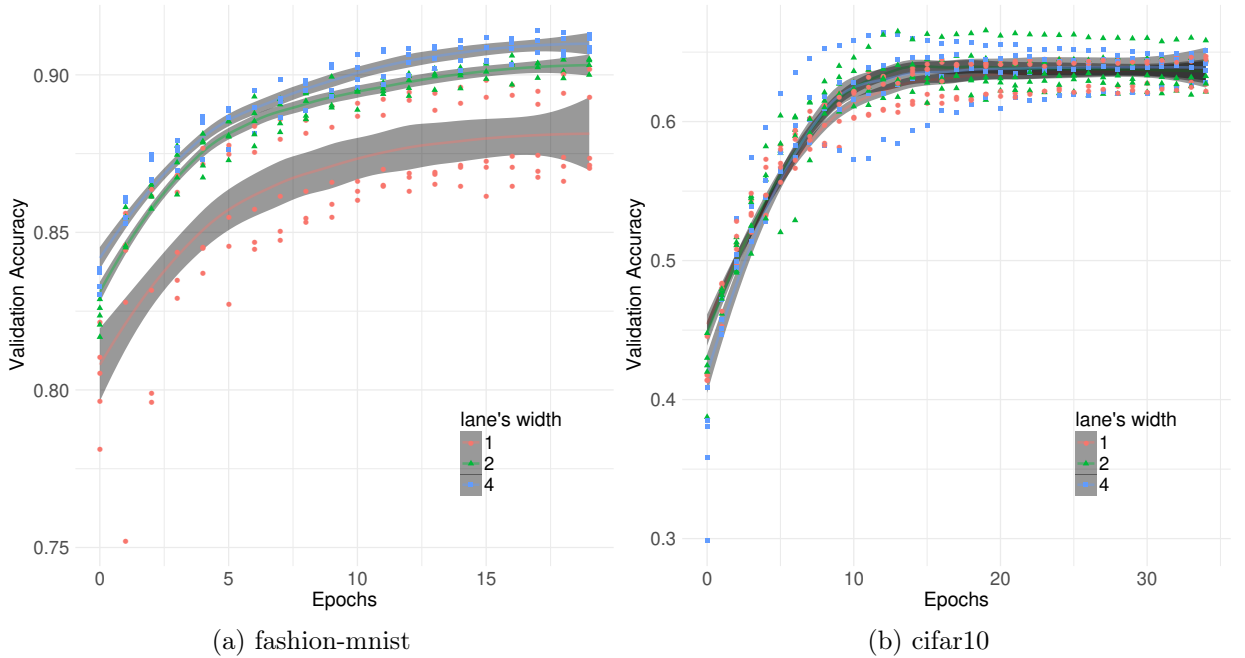


Figure 4.5: Accuracy on the Fashion-MNIST and CIFAR-10 dataset by MLCN networks with 8 lanes but different lanes width.

4.3.3 Lane Dropout Trade-off

To improve the number of lanes that learn useful features, we added a lane dropout mechanism (only for classification loss). Then, to test how individual lanes impact the classification, we removed one of the lanes from a *trained* Fashion-MNIST MLCN with 16 Mlcn1 lanes and width 4. We then measure that lane’s impact on the accuracy by calculating the new accuracy and computing the difference. Repeating this for all lanes results in a list of all lanes by individual accuracy impact.

Figure 4.6a shows the impact of removing lane by lane sorted by the accuracy impact with and without lane dropout. Applying dropout generally reduced the maximum

obtained accuracy. One of the reasons for this is that all lanes are thus forced to independently learn to classify the input, thus learning redundant information. This can be observed by the fact that when using the lane dropout, we only have a significant reduction of accuracy after removing 15 lanes. In other words, many lanes are redundant for classification, so one can remove subsets and continue having a good result. However, without using dropout, we have better maximum accuracy, and, as seen in Figure 4.6a, there is less redundancy and more lanes contribute information for the classification, so their removal will severely impact accuracy.

Independently of using dropout, it is seen that, after the training process, one can reduce the size of the network by removing the least significant lanes without drastically impacting the performance of the network. This affects not only the size of the stored network but also its inference time and speed. Figure 4.6b shows how the inference speed (normalized with respect to the version with all lanes) increases as some lanes are removed.

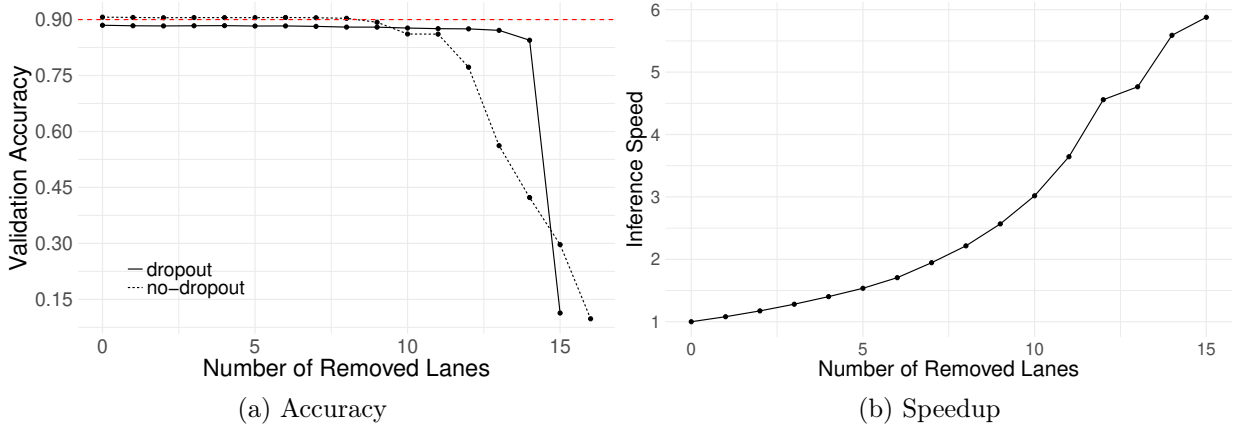


Figure 4.6: Impact of lane pruning on the accuracy and execution speedup of the network. Starting from the lane with least impact to the one with the largest impact.

4.3.4 MLCN Training and Inference Time

To facilitate the comparison between Mlcn1 and Mlcn2, we chose their configurations to have the same amount of trainable parameters. Thus, both configurations had similar performance both in training and inference. Furthermore, when compared to the CapsNet, MLCN needs a smaller or similar number of parameters to achieve the same accuracy. Added to the fact that the lanes are data-independent and have a higher amount of parallelism, we could train the MLCN much faster than the CapsNet. On average, for the same or better accuracy, one achieves a 2.4x speedup. Some of these results are shown in Table 4.1.

Table 4.1: Comparison between the performance on the CIFAR-10 and Fashion-MNIST dataset of the original CapsNet and the two MLCN architecture organization proposed (Mlcn1 and Mlcn2).

Dataset	DL model	# of Lanes	Lane's Width	# of params	Train Time (seconds/epoch)	Accuracy
CIFAR-10	CapsNet	-	-	11,769,600	240	66.36%
	Mlcn1	4	4	5,250,816	54	63.88%
	Mlcn1	32	2	14,259,712	205	66.56%
	Mlcn2	4	4	5,250,816	53	69.05%
	Mlcn2	32	2	14,259,712	204	75.18%
Fashion-MNIST	CapsNet	-	-	8,227,088	220	91.30%
	Mlcn1	2	4	3,655,376	21	91.14%
	Mlcn1	8	4	10,633,232	90	90.87%
	Mlcn2	2	4	3,655,376	20	91.01%
	Mlcn2	8	4	10,633,232	92	92.63%

4.3.5 MLCN Scalability and Performance Study

To understand how each approach to the parallelization of CapsNet scales, we studied their performance with 1, 2, 4, and 8 NVIDIA Tesla K80 GPUs.

The graph in Figure 4.7 shows the performance comparison between the base (CapsNet baseline), mlcn-data, and mlcn-model configurations. MLCN is faster than the baseline even in a single GPU, as reported earlier. However, it is interesting to notice that the advantage does not increase when scaling to more GPUs with data parallelization, as the speedup difference between mlcn-data and baseline remained constant. This indicates that the reorganization proposed by MLCN does not improve scaling via data parallelism, as expected, but provides benefits for model parallelism. In this latter case, the Mlcn-model has a visible advantage, scaling with higher efficiency and achieving a near 7.2 speedup with 8 GPUs over the single GPU baseline. Thus, MLCN is not only faster than the original CapsNet (baseline) but, because it allows model-parallelism, scales more efficiently.

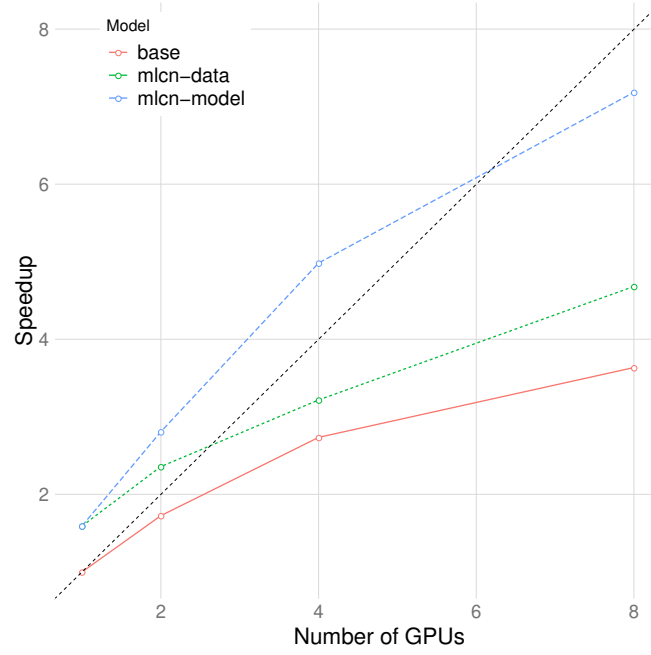


Figure 4.7: Speedup of the three parallelization approaches: original CapsNet with data parallelism (base), MLCN with data parallelism (mlcn-data), and MLCN with model parallelism (mlcn-model). All speedups are relative to the original CapsNet on one Tesla K80 GPU.

Impact of Batch Size

The size of the minibatch, or batch size, has a significant impact on the performance of a DNN as a better ratio of computation and communication is available, enabling a more efficient hardware use. Also, batch size has a significant impact on data parallelism performance as more data per computation is available to be divided among the GPUs. To study the advantage of MLCN model parallelism over the data parallelism method, we tested both approaches with batch sizes equal to 100, 150, 300, and 600. The graphs in Figures 4.8a and 4.8b show the speedup versus a single GPU with a 100-sized batch size. For both methods, we observe similar efficiency gain as the batch size grows. So, for different batch sizes, the relative advantage of MLCN with model parallelism stays the same, as increasing the batch size equally increases the efficiency of data and model parallelism approaches.

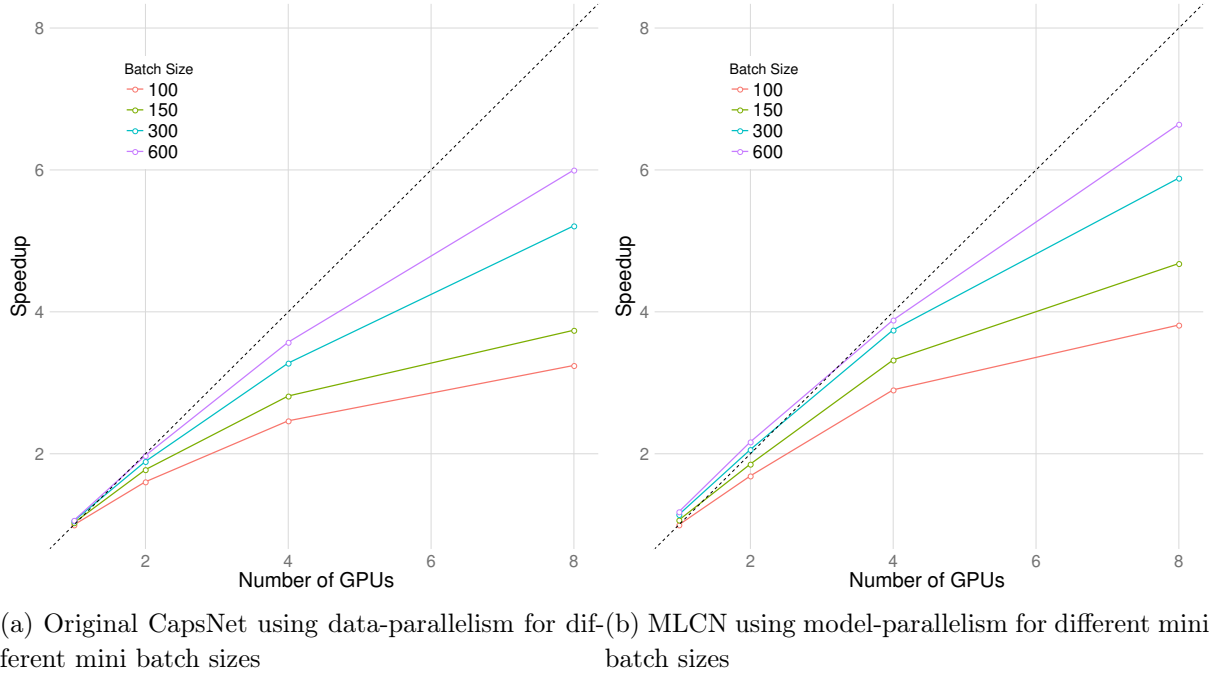


Figure 4.8: MLCN and original CapsNet speedup scalability for 1, 2, 4, and 8 NVIDIA K80 GPUs using a Google Cloud VM with 24 vCPUs and 90GB of RAM.

We also studied the impact of batch size on both baseline and MLCN accuracy, shown in Figure 4.9. Increasing batch sizes has the same impact on the accuracy for both cases. The magnitude of this impact is sensitive to the dataset as shown by the differences between Fashion-MNIST and CIFAR-10 results and not to the parallelism model. Thus, as model parallelism has better performance over data parallelism, it has the advantage of scaling better with smaller batch sizes, reducing the pressure to trade accuracy for efficiency.

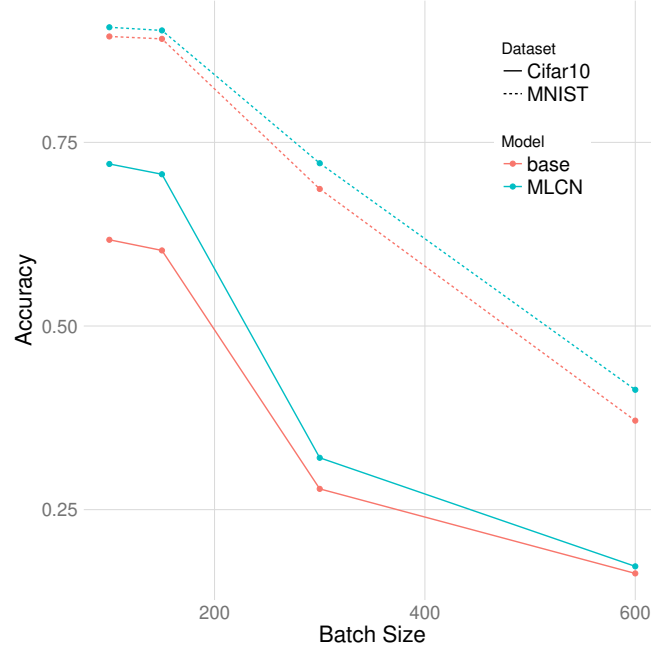


Figure 4.9: Accuracy impact with the increase of training batch size for the original CapsNet and MLCN in the CIFAR-10 and Fashion-MNIST datasets. Both are equally sensible with the increase of it.

Impact of Lane Characteristics

The previous results explored the suitability of MLCN and its model parallelization. We also explore how the characteristics of the MLCN lanes can affect performance and scalability by varying the three main hyperparameters associated with MLCN lanes: their width, depth, and their quantity. The results are shown, respectively, in Figures 4.10a, 4.10b and 4.10c.

The width and depth of lanes have a direct impact on the number of parameters per lane and, consequently, the amount of computation per lane. With more computation per lane, the efficient use of multiple GPUs becomes advantageous. This is shown in Figures 4.10a and 4.10b as larger lanes increase efficiency. However, increasing the width had a much more significant increase in efficiency, at a similar increase in the number of parameters. This indicates that besides the number of parameters, the type of computation affects performance. In the case of MLCN lanes wider lanes result in better performance than deeper lanes with the same number of parameters.

Another interesting point was the fact that increasing the number of lanes did not significantly increase performance, as shown in Figure 4.10c. Even though increasing the number of lanes also increases the amount of computation available between batches, there is an overhead of having these computations separable. So, having several lanes in one GPU is less efficient than having a single extremely large lane.

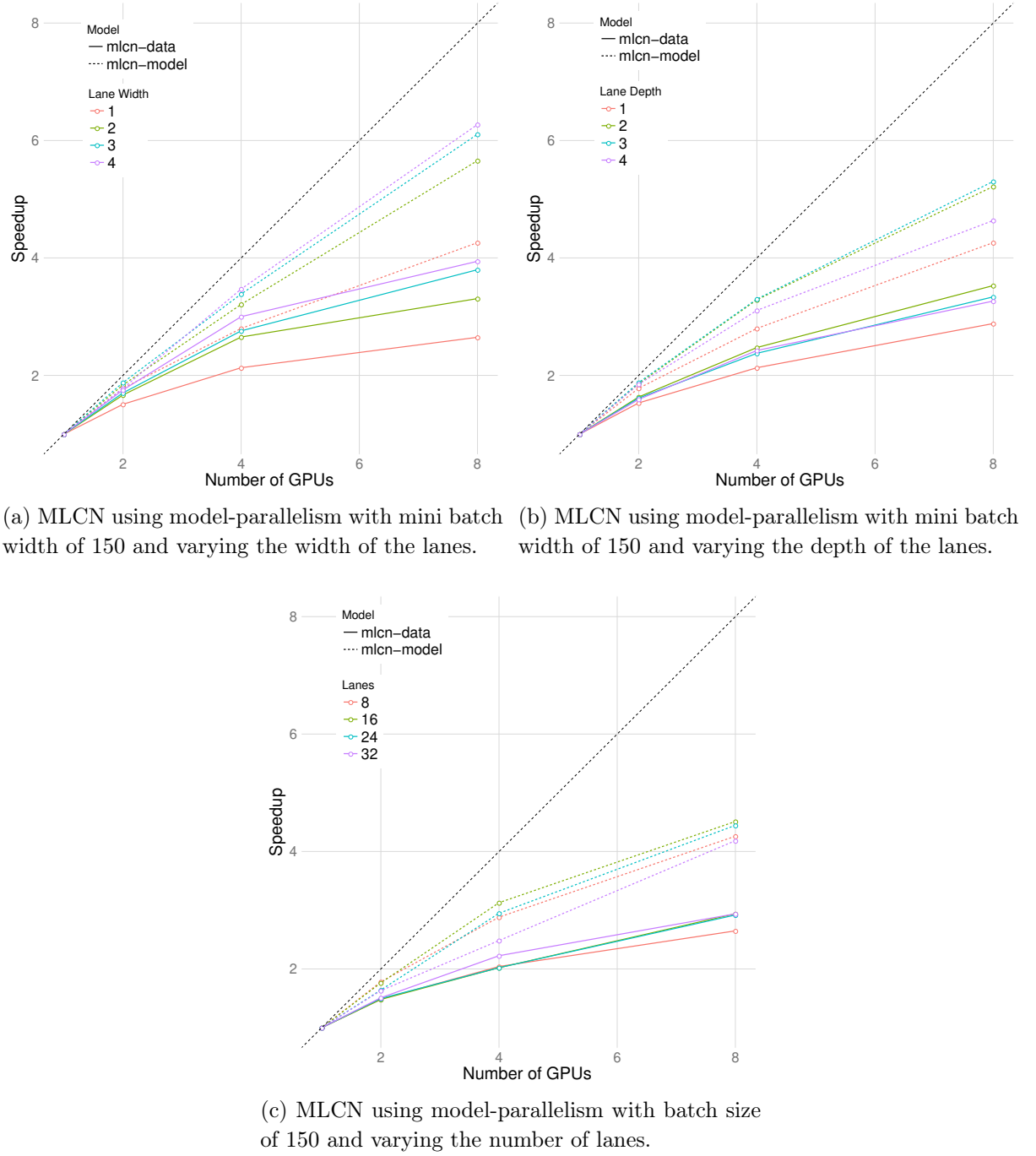


Figure 4.10: Impact of different MLCN's lanes characteristics in the speedup scalability of the network model.

4.4 Summary and Considerations

The Capsule Network was proposed with the goal to present a new mechanism to encode and learn positional information of inputs. It uses not scalar values in its atomic representations but vectors or matrices. We showed in this chapter that the dimensions of these vectors can learn different features of the images, and they can be data-separated without affecting their performance or properties.

We call these, now data-separated, paths in the Capsule Network as lanes. These lanes are a kind of independent Neural Network that can be constructed in many ways, varying their deepness, width, type of convolutions, among other things. In the computational efficiency usage point-of-view, more computation with less data dependence is better. However, we saw that increasing the number of dimensions of the capsule; thus, the lanes have a limit related to the size and complexity of the dataset. So, the parallelism and computational gains can be limited by the dataset.

Moreover, even for small and simple datasets, such as CIFAR-10 and Fashion-MNIST, we can achieve better execution times when using our Multi-Lane Capsule Network approach. The hardware and compilers automatically explored the lane parallelism even in a single-hardware environment and achieved better performance than the original CapsNet.

However, going further, we manage to show that we can use model parallelism with the lanes to achieve better performances and scale to the use of multiple hardware substrates during training. In the end, MLCN was able to achieve more than 7x faster training times when using 8 NVIDIA K80. In a homogeneous set of hardware substrates, scheduling the lanes is as simple as equally distributing them. However, when we explore model parallelism in heterogeneous hardware, we would have to consider the limits and performance of each hardware substrate in order to schedule the lanes in an equally balanced manner. We explore that problem in the next section proposing a lane schedule mechanism.

Chapter 5

Optimizing MLCNs on Heterogeneous Scenarios

One of the main advantages of having data-independent lanes is that these lanes can be deployed separately into multiple hardware substrates. If we have multiple equal lanes and multiple, identical, hardware substrates, deployment is as simple as dividing the lanes equally over the hardware resources, only being concerned with the communication cost involved. If, in other cases, we have lanes with different shapes, characteristics, and computational intensity or/and we have multiple hardware substrates with different characteristics or computational power, efficient deployment becomes more involved. First, because it now involves load balancing the computational intensity of the lanes and the computational power of the computing devices and, second, because now there is also the chance to apply different optimizations to different pairs of hardware and lanes. This scenario is illustrated in Figure 5.1, which shows how multiple lanes can be deployed on different hardware substrates with different compilation stacks.

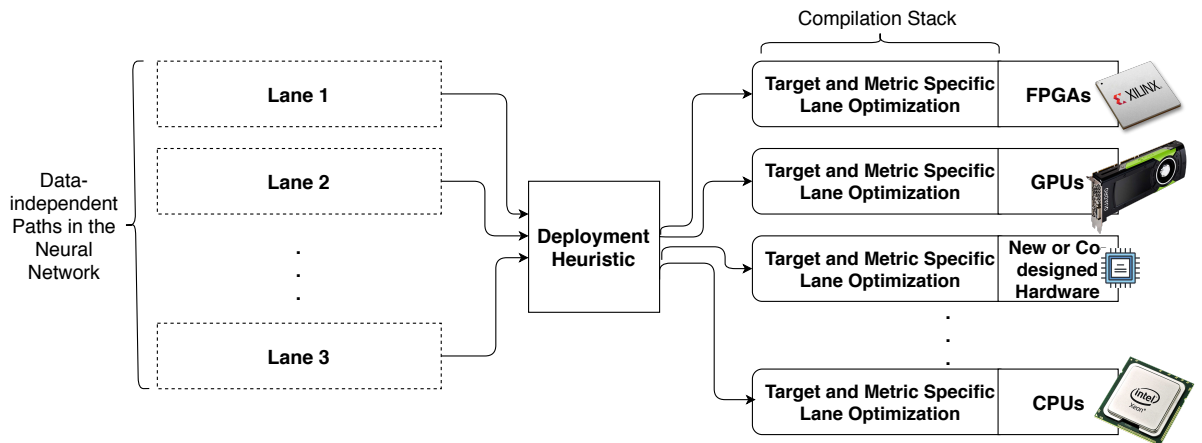


Figure 5.1: Diagram showing how multiple MLCN’s lanes can be trained in parallel using multiple computing devices even in heterogeneous scenarios.

Deciding where to execute each lane or what optimizations to apply to each lane/-computing device pair is not trivial. In this section, we present an approach to address the first problem using a scheduling heuristic.

5.1 Execution Cost Model for MLCN Lanes

Finding, statically, the optimal solution to deploy a lane given a set of hardware resources is a complex task. For example, aspects such as the version of the compiler being used or what other lanes (and their characteristics) are being executed concurrently on the same hardware can have a significant impact on the final performance. These are only two of the many aspects that can affect performance. However, we observed in our experiments that, at least for MLCN, we do not need to know exactly the accurate final performance to make a good scheduling decision. Our experiments have shown that reasonably approximate predictors can provide acceptable results.

We experiment running and taking the average execution time of 10 executions of MLCN lanes with different widths and depths using three different NVIDIA GPUs (K80, P100, and V100). For the same number of parameters, independently of the GPU used, the performance displays a well-behaved pattern. It varies linearly when increasing the depth, quadratically when varying the width, and it was multiplied by a factor when changing the GPU.

Thus, for MLCN lanes with NVIDIA GPUs and compilers, predicting the performance on a given hardware substrate can be approximated by Equation 5.1, which achieves a 0.901 Pearson correlation with our experimental data.

$$lane_{cost} = (lane_{width})^2 \times lane_{depth} \times GPU_{speed} \quad (5.1)$$

The GPU_{speed} in Equation 5.1 is the speed factor of the GPU being used. It only has any significance when deploying to a heterogeneous set of GPUs, and the GPU_{speed} constant for each GPU can be inferred by simply measuring the execution time of a tiny lane in each GPU and normalizing it. This can be done before the execution, and it has an insignificant cost in the final execution time. In the case of our experiments, we collect the GPU_{speed} by executing a 512x512 fully connected network with a small set of data. Normalized by K80, we found the following GPU_{speed} values for M40, P100, and V100: 3.1, 4.2, and 6.

5.2 Scheduling Heuristic

We can make good execution cost predictions for NVIDIA GPUs and MLCN lanes as described in 5.1. However, there is still the problem of how to schedule a set of lanes with different depths and widths on a set of GPUs with different speeds. We can model this problem as a numerical set partition with N bins, each bin corresponding to a target GPU. The cost of each lane being deployed (inserted into the bin) is equal to the lane cost (Equation 5.1) multiplied by previous execution speed prediction on host hardware via execution of a tiny lane (GPU_{speed}).

The numerical set partition problem is NP-Hard, but very good results can be achieved using heuristic/approximative algorithms, and it can even be solved in pseudo-polynomial time using dynamic programming, making it one of "The Easiest Hard Problem" [71, 72].

One of such heuristics that achieved good results and is very simple to implement is greedy scheduling which always inserts the remaining lane with the highest cost in the least loaded bin. Algorithm 3 shows this greedy algorithm, including the pre-execution used to calculate the GPU_{Speed} .

Algorithm 3 Greedy Scheduling Algorithm.

```

if using heterogeneous hardware:
    for each hardware:
        execute a tiny lane
        GPUSpeed[i] = runtime of the tiny lane on hardware
    // normalize all GPUSpeed
    GPUSpeed[*] = GPUSpeed[*] / MIN(GPUSpeed[*])

def GreedyPartition(lanes, NumGPUs, GPUSpeed):
    GPUBins = [[] for i in range(NumGPUs)]
    for lane in reverse sorted lanes:
        sort GPUBins by GPUBins[i][j]*GPUSpeed[i]
        GPUBins[0].append(lane)
    return GPUBins

```

5.3 Experimental Setup

In our experiments, we used virtual machines from Google Cloud. All virtual machines instantiated had 24 vCPUs with 50GB of RAM and a default network interface. We used different GPU setups, including NVIDIA M40, K80, P100, and V100, all with CUDA 10.0, Intel MKL-DNN, and Tensorflow 1.13.1.

The experimental results did not show sensitivity to the input data set (tested with Fashion-MNIST and CIFAR-10); thus, we chose to use the Fashion-MNIST data set. Execution time was measured by executing ten Fashion-MNIST epochs, excluding the first, and using the average time for the others. All results had a very small variation. The execution time between epochs consistently had a very similar value. Thus, for simplicity, we present averages.

In this work, we tested three configurations for the CapsNet parallelization, as follows:

- Original with Data Parallelism (**CapsNet or baseline**): we simply applied the original concept of CapsNet for the Fashion-MNIST dataset parallelized using Keras data parallelism support.
- MLCN with Data Parallelism (**mlcn-data**): we used the same approach as in the baseline (Keras data parallelism), but with the MLCN organization.
- MLCN with Model Parallelism (**mlcn-model**): we parallelize the execution by executing each lane on different GPUs. When using multiple machines, we used the Horovod MPI framework to do handle communication.

For the NAS experiments, we used K80 NVIDIA GPUs and executed the training only until the 10th epoch with batch size equal to 100. The accuracy reported is the geometric mean of each generated model trained four times. We generated 64 models per tested configuration. The best value is the best average of all iterations, and the average is the average of all iterations.

5.4 Experimental Results

One interesting observation about MLCN is that having lanes with different characteristics, such as lanes with different widths and depths, increases the generality of the network. Here, resembling ensemble learning increase of diversity by having first-level stacking models with different characteristics and strengths. A similar result was reported by Canqun *et al.* [50] with the MS-CapsNet organization. However, deploying lanes in multiple GPUs when the lanes have different computational footprints can be challenging. To study a proposed heuristic to deploy lanes with different widths and depths, we tested 4 MLCN networks with 6, 9, 12, and 24 lanes. Each lane may have pairs of depth and width values ranging from 1 to 5. As shown in Figure 5.2, we obtain a shorter execution time with our proposed heuristic than when randomly distributing the lanes between the GPUs. The advantage increases with the number of lanes, showing that the larger the number of lanes the harder it is to randomly find a good distribution. Notice that the time accounted for the greedy heuristic includes the (almost negligible) time to run the heuristic.

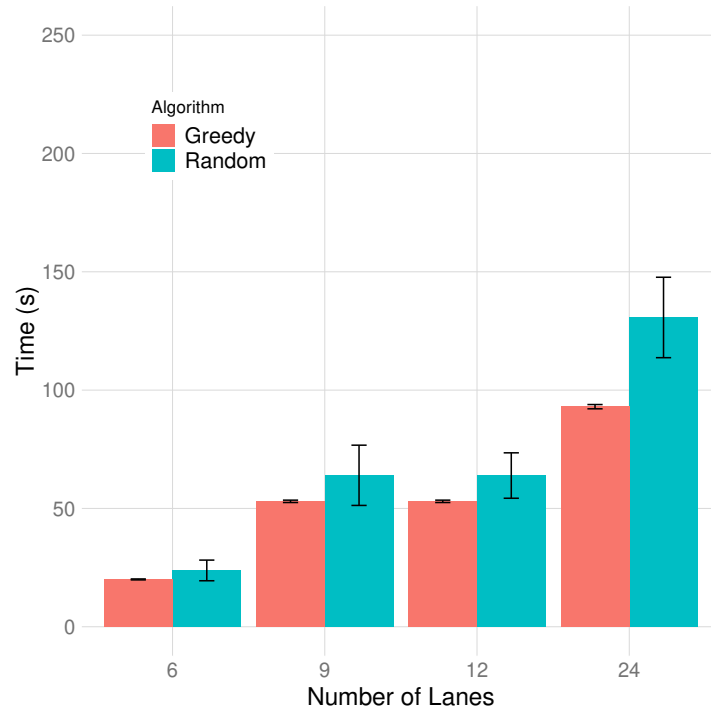


Figure 5.2: Average execution time of heterogeneous lanes running on four NVIDIA K80 GPUs with random and the greedy lanes’ scheduling approaches. All lanes varying on width and depth.

5.4.1 Heterogeneous Lanes with Heterogeneous GPU

Besides having heterogeneous lanes we also tested a scenario with heterogeneous hardware. Rather than 4 NVIDIA K80, we deployed four systems, each with a different GPU: one M40, one K80, one P100, and one V100. The results are in Figure 5.3. For total execution time, there was a significant increase because of network communication between the systems. Moreover, the difference between random deployment and our greedy heuristic becomes larger, showing that for more complex scenarios with many lanes or heterogeneous hardware, carefully computational deployment is key to performance.

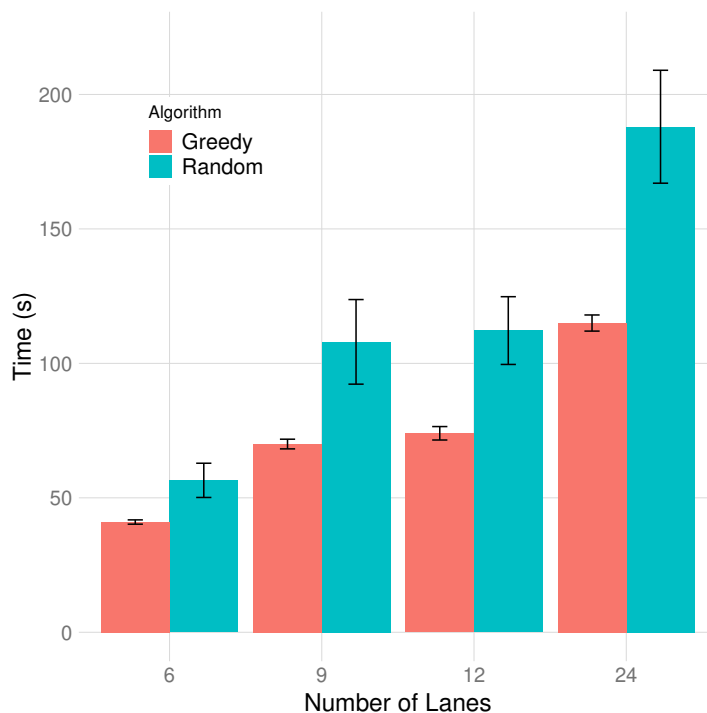


Figure 5.3: Average execution time of heterogeneous lanes running on one K80, one P100, one V100, and one M40 NVIDIA GPU in multiple machines communicating using MPI with a random and a greedy scheduling of lanes execution distribution. All lanes varying on width and depth.

5.5 Summary and Considerations

We showed that MLCN allows an easy way to achieve good results with model parallelism because of its data-independent lanes. However, that is not so straightforward when neither the lanes nor the hardware are homogeneous.

In that scenario, we need to decide which lane is going to be executed in which hardware such as to achieve a load balance. One way of modeling that problem is as a numerical set partition problem. We showed in this section how to model such a problem and one possible solution using a greedy approach.

So, lanes can be different in an MLCN model, and the hardware in each of the lanes that are going to be executed can also be diverse. Thus, one next problem that we explore is if we can create a model (such as in a Neural Architectural Search) that fits a given set of hardware available, so we use it at its maximum power and efficiency. That is described in the next chapter.

Chapter 6

Optimizing MLCN to Computing Devices with NAS

Previously, we presented the problem of deciding on which device to execute each lane in an MLCN model. In the previous chapter, we showed that we could create a simple equation that describes the performance behavior of MLCN models to guide a load balance scheduling algorithm. It is known that when one can balance the lanes onto the devices, we may achieve better overall performance. Moreover, larger models with heterogeneous lanes were shown to have better accuracy, especially in more complex datasets such as CIFAR-10. Thus, in this chapter, we explore a slightly more general problem: we still want to balance the load of an MLCN model in a set of devices, but now we do not have the model defined but we want to build it. The goal is to build MLCN models with heterogeneous lanes that when schedule into a a given set of hardware substrates, fully uses their resources; to that, we perform a random Neural Architectural Search (NAS) to find good models in accuracy while fitting the devices resources. So, in the end, we can select the best model that both improves the accuracy while uses the devices' resources smartly.

To achieve that, we first introduce a new equation to compute the memory cost of an MLCN given its lane configurations. With this new equation, along with the execution cost equation introduced in Section 5.1, we can predict both memory and FLOPS/time necessary to execute a lane or an MLCN model in a given device. So now the notion of a device is a lane's bucket that has both a memory and FLOPS/time limit. Next, we introduce the idea of how to generate random lanes that can respect the memory or computational load limit, in FLOPS. This mechanism is used to randomly generate lanes to build a MLCN model that fully uses the available set of devices resources. In the end, we have heterogeneous lanes (that we showed are able to achieve better accuracy), and, as the lanes are filled, having in mind the device resources, those lanes can be executed on those devices in a balanced way. Finally, we can generate numerous MLCN models and then select the model that gives the best accuracy for a fixed number of epochs.

All these concepts and ideas are illustrated in the diagram from Figure 6.1. Note that the buckets are illustrated in equal size, but nothing prevents them from being different in terms of FLOPS and memory limits, for instance, in a heterogeneous hardware scenario.

Also, notice that we solve the problem from the previous section in such a way that the randomly generated MLCN is kept balanced as we fill the buckets.

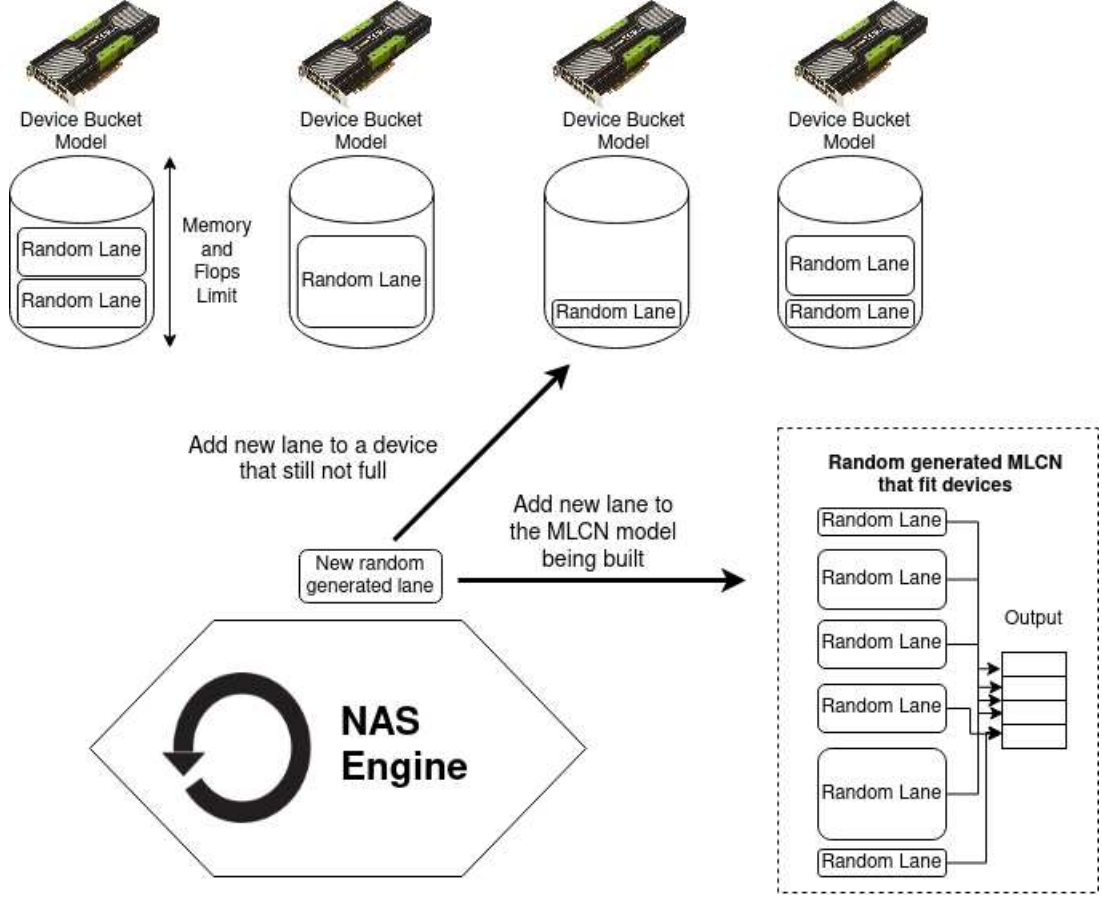


Figure 6.1: Diagram of NAS engine mechanism used to generated random MLCN models with heterogeneous lanes that naturally fit a given set of devices.

The remainder of this chapter is organized as follows: Section 6.1 presents the memory heuristic and Section 6.2 discusses the algorithm to perform this architectural search. Finally, the results section explores how limiting the memory or the FLOPS for an MLCN configuration search or having heterogeneous limits impacts the final accuracy.

6.1 Memory Cost Model for MLCN Lanes

To implement the bucket approach previously described, we need not only to estimate the computational time/resources/FLOPS as described in Section 5.1, but also to estimate the amount of memory necessary to execute the lane. For that, we use the NVIDIA *nvidia-smi* tool to capture different lane configurations and create a model of memory consumption based on the lane characteristics. It became clear from our experiment that the memory consumption is deterministic and can be approximate by the following equation:

$$lane_{memory} = ((lane_{width})^{lane_{depth}/2} \times 256 + 1832)MiB \quad (6.1)$$

For the lanes that we tested, this equation gave a memory consumption that was always an upper ceiling for the real value and with an average error of less than 256 MiB. We know that this value could have been analytically calculated from the network model layers and configurations, but we want to show that with some experiments, we can come up with an equation that is good enough and does not require knowledge of the layers' actual functionality. Moreover, this equation provides a fast way to calculate the memory consumption, allowing us to quickly test each $lanes_{width}$ and $lane_{depth}$ that can be used to fit a given memory requirement.

Memory consumption and the execution time have clearly different behaviors because the number of trainable parameters grows differently from the layers output and activation's size. While the first is more related to the execution time, the second is more related to memory consumption.

6.2 Lane Configuration Search

Given the performance and memory consumption results of the MLCN model, its relation with the lane characteristics, and the fact that in our experiments we had evidence that MLCN models with heterogeneous lanes have higher accuracy than homogeneous ones, we proposed an approach to make a Neural Architectural Search (NAS) of such MLCN models that fit a given set of memory and performance constraints.

Our NAS tries to find an MLCN model with the best possible accuracy or at least a good one that, at the same time, fully uses all resources available in a balanced way. Ensuring that we are testing and generating the largest possible models (as larger models tend to have better accuracy performance) and that can be executed using the full capacity of the devices (load balanced execution with higher performance). In other words, we want to find a model M that maximizes the accuracy in a given dataset. Moreover, the lanes L from this model M when schedule to a hardware A uses their a memory $mem(A_j)$ to the limit and uses the maximum load $load(A_j)$. A , $mem(A_j)$ and $load(A_j)$ are parameters of the search algorithm, they describe the capabilities of the hardware available. So, let $lanes(A_j)$ be the set of all lanes assigned to be executed in A_j , the model M needs to maximize the accuracy while its lanes still fit the devices, *i.e.*:

$$\forall j, \sum_{i=0}^{\#(lanes(A_j))} mem(L_i) < mem(A_j) \wedge \sum_{i=0}^{\#(lanes(A_j))} load(L_i) < load(A_j)$$

The memory restriction ensures that we are only generating models that can fit the devices and the load limit guarantees that we are using the same load for all devices in a balanced way.

To perform such a search respecting the given constraints, we developed a random search approach to implement the NAS engine. First, we develop a generator of random lanes that can fit in a given memory limit and have a limit of FLOPS usage. This is done using the memory and performance equations. Using the lane generator, we then build

a random model that fits a set of devices in a balanced way. For this, we select a device that is not yet full and randomly generate a lane that can fit such a device, then assign this new lane to the device and the MLCN model being built. This is repeated until all devices are full. Finally, we test the accuracy of this generated model and update the best model found so far, if necessary. This repeated process of generation and test of random generated MLCN networks is used to explore the space of possible MLCNs that fits the given devices to find the best one.

This random search algorithm is described in the pseudo-code from algorithms 4, 5, and 6.

The first, Algorithm 4, describes the structure used to maintain how much of a device has already been used. *memory_{limit}* and *flops_{limit}* are the *mem*(A_j) and *load*(A_j) from last Equation. The structure maintain the available load and memory in the device in *free_{memory}* and *free_{flops}*. The function *add_{lane}* can be used to update the structure and *is_{full}* to check if the device is full or still have available resources.

Then, Algorithm 5, describes the process of generating an MLCN model that fits all devices. It construct an random MLCN by generating lanes that can fit devices that are not yet full. Every time a new lane is generated, the device usage is updated and if it has became full, it is remove from the list of devices. This repeats until there is no device to be fill.

Finally, Algorithm 6 describes the random NAS, in which n models are generated, and the most accurate one is returned.

Algorithm 4 Device Memory Bucket and FLOPS Bucket Representation.

```

1  class Device(memory_limit , flops_limit ):
2      free_memory = memory_limit
3      free_flops = flops_limit
4      def is_full():
5          if free_flops <= 0 or
6              free_memory <= 0:
7              return true
8          return false
9      def add_lane(lane):
10         free_memory -= lane.mem
11         free_flops  -= lane.flops

```

Algorithm 5 Generator of random MLCNs that fit devices.

```

1 def gen_random_mlc_n_that_fits:
2     MLCN = empty MLCN model
3     # Initialize list of devices with their
4     # respective memory and FLOPS limit
5     devices = init_list_of_devices()
6     while len(devices) != 0:
7         dev = get a device from devices
8         # gen a random lane that uses memory and FLOPS
9         # less or equal than dev.free_memory and
10        # dev.free_flops
11        lane = rand_lane(dev.free_memory, dev.free_flops)
12        dev.add_lane(lane)
13        MLCN.add_lane(lane)
14        if dev.is_full():
15            remove dev from devices
16    return MLCN

```

Algorithm 6 MLCN Random Network Architecture Search.

```

1 def MLCN_RANDOM_NAS(N):
2     best_accuracy = 0
3     best_MLCN = none
4     for 1 to N:
5         random_MLCN = gen_random_mlc_n_that_fits()
6         train(random_MLCN)
7         accuracy = test(random_MLCN)
8         if accuracy > best_accuracy:
9             best_accuracy = accuracy
10            best_MLCN = random_MLCN
11    return best_MLCN

```

6.3 Experimental Setup

In our experiments, we used virtual machines from Google Cloud. All virtual machines instantiated had 24 vCPUs with 50GB of RAM and a default network interface. We used different GPU setups, including NVIDIA Tesla M40, K80, P100, and V100, all with CUDA 10.0, Intel MKL-DNN, and Tensorflow 1.13.1.

For the NAS experiments, we used K80 NVIDIA GPUs and executed the training only until the 10th epoch with batch size 100. The accuracy reported is the geometric mean of each generated model trained four times. We generated 64 models per tested configuration. The best value is the best average of all iterations, and the average is the average of all iterations.

6.4 Experimental Results

Next, we tested how the strategy presented can use information about the scalability and performance of the MLCN models to build new ones that fit the hardware. With the ability to generate random models, we can search for good models that achieve good accuracy for CIFAR-10 while fitting systems with multiples NVIDIA K80. The algorithm behaved well by creating models that never exceed the GPU memory limits and have a very well-fitted load balance. Given this search with the ability to generate models with the presented characteristics, we explored how increasing the memory limit, computational power limit, and the number of hardware substrates would affect the average and best model found by executing the random search for 64 iterations, ten epochs per model and batch size equal to 100. Our experiments indicate that these parameter values obtain the best accuracy performance for most of the models. The hyperparameter search could be done for each model generated, but it would require much more execution time to the search. From the plot from Figure 6.2, we can observe how accuracy increases when we allow the use of more computational power. We started by using the amount of computation from a baseline model with four lanes, width two, and depth 4. When we generate other models with a similar number of FLOPS, we end up with the average, and the best result is presented as “1x”. Following, we allowed the creation of models 2-fold more computationally intensive, then 4- and 8-fold. We can see that the model’s average and best accuracy found by the random search increases for larger networks but with a tendency to slow after some point. Similar to what we mentioned earlier: for a given dataset and number of epochs, there seems to be an ideal network size that achieves the best performance.

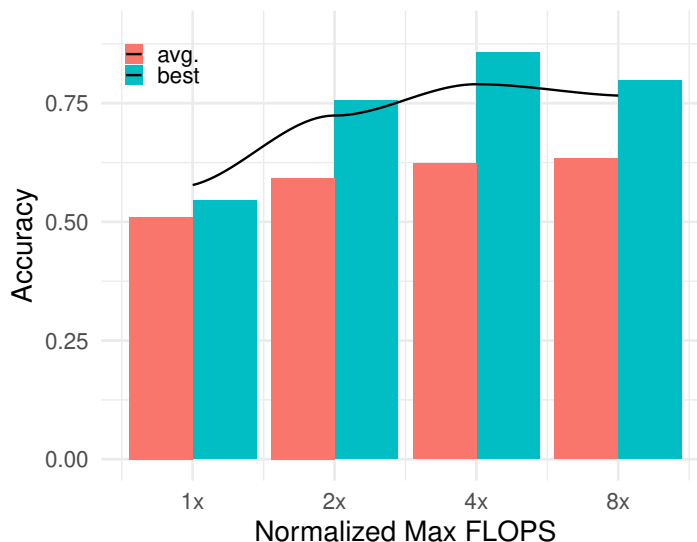


Figure 6.2: Average and best model’s accuracy for 30 random generated models for each max FLOPS limit: 1x, 2x, 4x, and 8x. 1x starts with a previously given baseline size. All experiments were executed allowing the use of up to 12GB of memory and 8 buckets (GPUs).

We then repeated the experiment with a fixed maximum FLOPS in 8x the baseline but varying the memory limit. As both FLOPS and memory consumption are tightly connected, we see a similar behavior: an increase, until a point, of the accuracy as we increase the amount of memory allowed to the networks. This result is presented in Figure 6.3.

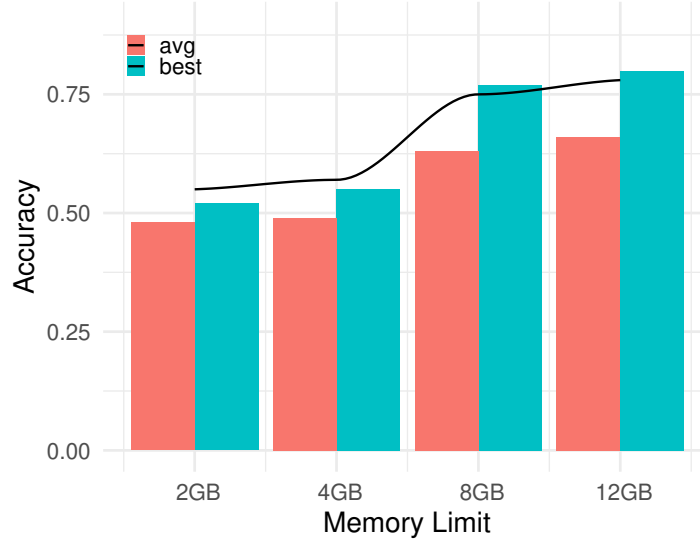


Figure 6.3: Average and best model's accuracy for 30 random generated models for each memory limit: 2GB, 4GB, 8GB, and 12GB. All experiments were executed allowing the use of up to 8x FLOPS baseline and 8 buckets (GPUs).

The same behavior appears, Figure 6.4 when we fix the amount of memory and maximum FLOPS but increase the number of available buckets (the hardware devices capabilities representation).

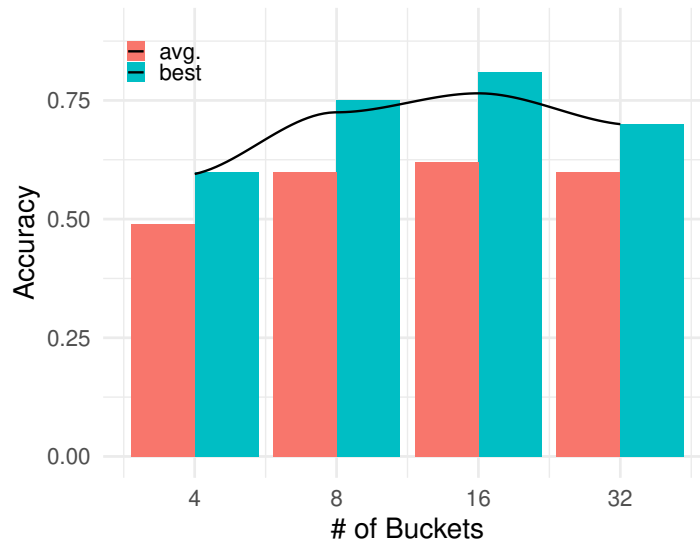


Figure 6.4: Average and best model's accuracy for 30 random generated models using 4, 8, 16, and 32 K80 GPUs. All experiments were executed allowing the use of up to 8x FLOPS baseline and 12GB of memory.

One interesting fact is that by using our random search, we were able to generate models that used the hardware’s resources efficiently and that the best found model had an 18.6% better accuracy than we had previously achieved. So, the results provide evidence that this NAS approach is useful to search for better MLCN models while creating load-balanced well-fitted models. In Figure 6.5, we show an example of the best-so-far accuracy result per iteration in the execution of the random search. In this execution, the average model accuracy was 62.0%. It took only eight iterations to find a significantly better than the average model.

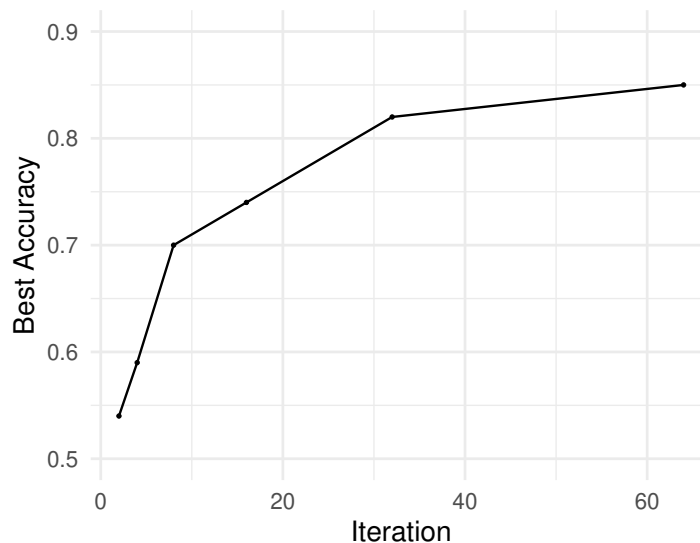


Figure 6.5: Best model’s accuracy found by Random Search throughout its iterations for 4x FLOPS experiment.

6.5 Summary and Considerations

We presented in this chapter the idea of personalizing the MLCN hyperparameters, such as the number of lanes with a Neural Architecture Search that is aware of the hardware in each network that is going to be executed. For that, more than knowing the performance of the network, we have to understand the memory usage of the network in such a way that we can create a network that fits the hardware and that easily balances the load on them.

In our experiments with CIFAR-10, the NAS approach led to networks with 18.6% better accuracy for a set of hardware. It took eight iterations to find a better accuracy than the original. The network created was naturally balanced to the hardware, thus having good execution efficiency.

Chapter 7

Conclusions

In 2017, Hilton and his team proposed the Capsule Network [1]. The goal was to mimic the human brain’s biology and capabilities, using vectors instead of scalars, avoiding pooling operations and still addressing the Picasso problem. The capsules in the network agree (routing algorithm) to form the capsule of the next layer.

In this work, we proposed an improvement to the original Capsule Network facing one of its main challenges: low execution speed. We made the last capsule to have each of its dimensions (that should be learning different features) in data-independent Neural Networks that can be easily parallelizable—creating what we call Lanes in a Multi-Lane Capsule Network (MLCN).

Lanes are a kind of independent Neural Network that can be constructed in many ways, varying their deepness, width, type of convolutions, among other things. We tested varying its convolution types, width, height, among other characteristics. For instance, we showed that when showing different convolutions types (mlcn1 and mlcn2), we could achieve very different accuracy performances. Mlcn2, the best configuration found for Fashion-MNIST and CIFAR-10, achieved 75.18% accuracy on CIFAR-10 vs. 66.36% for the original CapsNet and 92.63% accuracy on Fashion-MNIST vs. 91.30% for the original CapsNet. Furthermore, given that the lanes are data-independent, single GPUs could execute it faster—17% training speedup for CIFAR-10 and 130% for Fashion-MNIST.

In the computational efficiency usage point-of-view, more computation with less data dependence is better. However, we saw that increasing the number of dimensions of the lanes has a limit that seems to be related to the size and complexity of the dataset. So, the parallelism and computational gains can be limited by the dataset.

For CIFAR-10 and Fashion-MNIST, we manage to show that we can use model parallelism with the lanes to achieve better performances and scale to the use of multiple hardware substrates during training. In the end, MLCN was able to achieve almost $7.2\times$ faster training times when using 8 NVIDIA K80 than the original Capsule Network in a single GPU.

The parallelism of the lanes can be easily explored when the lanes are equal or when the hardware substrates are equal. However, in our experiments, we noticed that using lanes with different configurations could lead to better accuracy performance. To schedule now, this different lanes with different computational requirements into the hardware becomes a different problem. To face that, we model the scheduling problem as a numerical set

partition problem; we showed that a greedy approach could surpass a random or naïve approach by $2\times$ when training with 24 lanes in $4\times$ GPUs.

Moreover, lanes can be different in an MLCN model, and the hardware in each of the lanes that are going to be executed can also be diverse. Thus, one next problem that we explore is if we can create a model (such as in a Neural Architectural Search) that fits a given set of hardware available so we use it at its maximum power and efficiently while maximizing its accuracy. For that, we presented the idea of personalizing the MLCN hyperparameters using a Neural Architecture Search aware of the hardware in each network that is going to be executed. For that, more than knowing the performance of the network, we have to understand the memory usage of the network in such a way that we can create a network that fits the hardware and that easily balances the load on them.

The NAS approach led to networks with 18.6% better accuracy for a set of hardware. It took eight iterations to find a better accuracy than the original. The network created was naturally balanced to the hardware, thus having good execution efficiency.

Our improvements to the CapsNet work and the exploration of its parallelization opened the opportunity to explore Capsule Networks to larger datasets. Since its first publication, different improvements to the MLCN were proposed in the literature, and MLCN was already used in many different contexts and problems.

More than improving the Capsule Network, this dissertation brings to attention the fact that Neural Network design could benefit from the awareness of its execution performance and parallelism. In a world with huge networks becoming natural, it is necessary that not only high-performance computing or compiler specialists improve the performance of the networks but also the AI specialists that are first creating them.

7.1 Challenges

In the year that this work had started to be developed (2019), CapsNet had almost no open-source good implementation. Frameworks such as Tensorflow were changing rapidly, making the configuration and reproduction of scripts and experiments hard to follow. A huge amount of tensor compilers and frameworks were surging, such as TVM, XLA, Glow, among others. Many with a low amount of documentation, mainly when related to parallelism.

Doing experiments with multiple GPUs as expensive as V100 and others used in this dissertation was also a huge challenge. More than 5 thousand Euros were used to run experiments on the Google Cloud. These were covered by a Google Cloud Grant. However, if we wanted to repeat or do the experiments proposed here for larger datasets with larger size images, the cost would become unfeasible for academic proposes. The need for cheaper and faster implementation is need not only to save money and time but also to help democratize these huge network access.

7.2 Future Work

Explore transforming others DL models in a more data-independent organization: we showed in this work that CapsNet can be reorganized in such a way that it allows a better exploration of execution parallelism. However, this could be extended to other Neural Networks types and architectures. For example, even NAS could be used to explore different data-flow graphs or different Neural Network models, having as an optimization metric the data independence of the network.

Explore CapsNet with larger datasets using multiple hardware substrates: giving that we present ways to accelerate CapsNet and execute it with larger datasets, it becomes now feasible to explore CapsNet with more complex and large datasets.

Explore others parallelization manners: we explore using data and model parallelism independently. But many others techniques could be applied, and even it could be treated as an optimization problem where we try to find the best parallelism scheme for an MLCN set of lanes.

Optimize MLCN for cloud computing: with the increasing popularity of cloud computing, it would be interesting to explore it with MLCN. One example would be to search for the best cloud machine configuration to train an MLCN model. Having in mind that different machines could be used at the same time, splitting the lanes through those machines.

Bibliography

- [1] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic routing between capsules,” in *Advances in Neural Information Processing Systems*, pp. 3856–3866, 2017.
- [2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [3] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.
- [4] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” tech. rep., Citeseer, 2009.
- [5] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” *Advances in neural information processing systems*, vol. 32, pp. 103–112, 2019.
- [6] R. Mehta, Y. Huang, M. Cheng, S. Bagga, N. Mathur, J. Li, J. Draper, and S. Nazarian, “High performance training of deep neural networks using pipelined hardware acceleration and distributed memory,” in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pp. 383–388, IEEE, 2018.
- [7] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.
- [8] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” in *Proceedings of Machine Learning and Systems* (A. Talwalkar, V. Smith, and M. Zaharia, eds.), vol. 1, pp. 1–13, 2019.
- [9] V. M. do Rosario, E. Borin, and M. Breternitz, “The multi-lane capsule network,” *IEEE Signal processing letters*, vol. 26, pp. 1006–1010, 2019.
- [10] V. M. do Rosario, M. Breternitz, and E. Borin, “Efficiency and scalability of multi-lane capsule networks (mlcn),” *Journal of Parallel and Distributed Computing*, vol. 155, pp. 63–73, 2021.

- [11] V. M. d. Rosario, M. Breternitz, and E. Borin, “Efficiency and scalability of multi-lane capsule networks (mlcn),” in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 152–159, 2019.
- [12] C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018.
- [13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [14] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
- [15] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, “The deep learning compiler: A comprehensive survey,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.
- [16] A. Daniely, R. Frostig, and Y. Singer, “Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity,” in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.
- [17] L. Bottou and O. Bousquet, “13 the tradeoffs of large-scale learning,” *Optimization for machine learning*, p. 351, 2011.
- [18] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, “Pruning and quantization for deep neural network acceleration: A survey,” *Neurocomputing*, vol. 461, pp. 370–403, 2021.
- [19] T. Yu and H. Zhu, “Hyper-parameter optimization: A review of algorithms and applications.” arXiv preprint 2003.05689 cs.LG, 2020.
- [20] C. Zhang and Y. Ma, *Ensemble machine learning: methods and applications*. Springer, 2012.
- [21] Z.-H. Zhou, *Ensemble methods: foundations and algorithms*. CRC press, 2012.
- [22] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994.
- [23] R. E. Schapire, “Explaining adaboost,” in *Empirical inference*, pp. 37–52, Springer, 2013.
- [24] J. H. Friedman, “Stochastic gradient boosting,” *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.
- [25] G. E. Hinton, A. Krizhevsky, and S. D. Wang, “Transforming auto-encoders,” in *International Conference on Artificial Neural Networks*, pp. 44–51, Springer, 2011.

- [26] W. L. Jia Xudong, "Text classification model based on multi-head attention capsule networks," *Journal of Tsinghua University(Science and Technology)*, vol. 60, no. 5, p. 415, 2020.
- [27] H. Ren and H. Lu, "Compositional coding capsule network with k-means routing for text classification," *arXiv preprint arXiv:1810.09177*, 2018.
- [28] T. Iesmantas and R. Alzbutas, "Convolutional capsule network for classification of breast cancer histology images," in *International Conference Image Analysis and Recognition*, pp. 853–860, Springer, 2018.
- [29] S. Srivastava, P. Khurana, and V. Tewari, "Identifying aggression and toxicity in comments using capsule network," in *Proceedings of the First Workshop on Trolling, Aggression and Cyberbullying (TRAC-2018)*, pp. 98–105, 2018.
- [30] F. Deng, S. Pu, X. Chen, Y. Shi, T. Yuan, and S. Pu, "Hyperspectral image classification with capsule network using limited training samples," *Sensors*, vol. 18, no. 9, p. 3153, 2018.
- [31] M. K. Patrick], A. F. Adekoya], A. A. Mighty], and B. Y. Edward, "Capsule networks – a survey," *Journal of King Saud University - Computer and Information Sciences*, 2019.
- [32] A. Shahroudnejad, P. Afshar, K. N. Plataniotis, and A. Mohammadi, "Improved explainability of capsule networks: Relevance path by agreement," in *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 549–553, IEEE, 2018.
- [33] A. Jaiswal, W. AbdAlmageed, Y. Wu, and P. Natarajan, "CapsuleGAN: Generative adversarial capsule network," in *European Conference on Computer Vision*, pp. 526–535, Springer, 2018.
- [34] A. Jiménez-Sánchez, S. Albarqouni, and D. Mateus, "Capsule networks against medical imaging data challenges," in *Intravascular Imaging and Computer Assisted Stenting and Large-Scale Annotation of Biomedical Data and Expert Label Synthesis*, pp. 150–160, Springer, 2018.
- [35] A. Mobiny and H. Van Nguyen, "Fast capsnet for lung cancer screening," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 741–749, Springer, 2018.
- [36] Y. Kim, P. Wang, Y. Zhu, and L. Mihaylova, "A capsule network for traffic speed prediction in complex road networks," in *2018 Sensor Data Fusion: Trends, Solutions, Applications (SDF)*, pp. 1–6, IEEE, 2018.
- [37] R. Mukhometzianov and J. Carrillo, "Capsnet comparative performance evaluation for image classification," *arXiv preprint arXiv:1805.11195*, 2018.
- [38] H. H. Nguyen, J. Yamagishi, and I. Echizen, "Use of a capsule network to detect fake images and videos," *arXiv preprint arXiv:1910.12467*, 2019.

- [39] Y. Li, M. Qian, P. Liu, Q. Cai, X. Li, J. Guo, H. Yan, F. Yu, K. Yuan, J. Yu, *et al.*, “The recognition of rice images by uav based on capsule network,” *Cluster Computing*, vol. 22, no. 4, pp. 9515–9524, 2019.
- [40] M. Singh, S. Nagpal, R. Singh, and M. Vatsa, “Dual directed capsule network for very low resolution image recognition,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 340–349, 2019.
- [41] Z. Zhu, G. Peng, Y. Chen, and H. Gao, “A convolutional neural network based on a capsule network with strong generalization for bearing fault diagnosis,” *Neurocomputing*, vol. 323, pp. 62–75, 2019.
- [42] Z. Chen and T. Qian, “Transfer capsule network for aspect level sentiment classification,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 547–556, 2019.
- [43] C. Li, C. Quan, L. Peng, Y. Qi, Y. Deng, and L. Wu, “A capsule network for recommendation and explaining what you like and dislike,” in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 275–284, 2019.
- [44] X. Zhang, P. Li, W. Jia, and H. Zhao, “Multi-labeled relation extraction with attentive capsule network,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 7484–7491, 2019.
- [45] C. Xiang, Z. Wang, S. Tian, J. Liao, W. Zou, and C. Xu, “Matrix capsule convolutional projection for deep feature learning,” *IEEE Signal Processing Letters*, vol. 27, pp. 1899–1903, 2020.
- [46] P. M. Kwabena, B. A. Weyori, and A. A. Mighty, “Exploring the performance of lbp-capsule networks with k-means routing on complex images,” *Journal of King Saud University-Computer and Information Sciences*, 2020.
- [47] P. Afshar, S. Heidarian, F. Naderkhani, A. Oikonomou, K. N. Plataniotis, and A. Mohammadi, “Covid-caps: A capsule network-based framework for identification of covid-19 cases from x-ray images,” *Pattern Recognition Letters*, vol. 138, pp. 638–643, 2020.
- [48] P. Afshar, A. Mohammadi, and K. N. Plataniotis, “Bayescap: A bayesian approach to brain tumor classification using capsule networks,” *IEEE Signal Processing Letters*, vol. 27, pp. 2024–2028, 2020.
- [49] P. Shamsolmoali, M. Zareapoor, L. Shen, A. H. Sadka, and J. Yang, “Imbalanced data learning by minority class augmentation using capsule adversarial networks,” *Neurocomputing*, vol. 459, pp. 481–493, 2021.
- [50] C. Xiang, L. Zhang, Y. Tang, W. Zou, and C. Xu, “Ms-capsnet: A novel multi-scale capsule network,” *IEEE Signal Processing Letters*, vol. 25, no. 12, pp. 1850–1854, 2018.

- [51] K. Sun, L. Yuan, H. Xu, and X. Wen, “Deep tensor capsule network,” *IEEE Access*, vol. 8, pp. 96920–96933, 2020.
- [52] S. Venkatraman, S. Balasubramanian, and R. R. Sarma, “Building deep, equivariant capsule networks.” arXiv 1908.01300 cs.LG, 2019.
- [53] J. Chen and Z. Liu, “Mask dynamic routing to combined model of deep capsule network and u-net,” *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–12, 2020.
- [54] T. Jeong, Y. Lee, and H. Kim, “Ladder capsule network,” in *International Conference on Machine Learning*, pp. 3071–3079, PMLR, 2019.
- [55] A. Punjabi, J. Schmid, and A. K. Katsaggelos, “Examining the benefits of capsule neural networks.” arXiv preprint 2001.10964 cs.LG, 2020.
- [56] B. Jia and Q. Huang, “De-capsnet: A diverse enhanced capsule network with disperse dynamic routing,” *Applied Sciences*, vol. 10, no. 3, p. 884, 2020.
- [57] S. Yang, F. Lee, R. Miao, J. Cai, L. Chen, W. Yao, K. Kotani, and Q. Chen, “Rs-capsnet: An advanced capsule network,” *IEEE Access*, vol. 8, pp. 85007–85018, 2020.
- [58] M. Edraki, N. Rahnavard, and M. Shah, “Subspace capsule network,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 10745–10753, 2020.
- [59] M. Jampour, S. Abbaasi, and M. Javidi, “Capsnet regularization and its conjugation with resnet for signature identification,” *Pattern Recognition*, p. 107851, 2021.
- [60] K. Sun, X. Wen, L. Yuan, and H. Xu, “Dense capsule networks with fewer parameters,” *Soft Computing*, vol. 25, no. 10, pp. 6927–6945, 2021.
- [61] Q. Ren, “Grouping capsules based different types,” *arXiv preprint arXiv:1911.04820*, 2019.
- [62] B. Mandal, R. Sarkhel, S. Ghosh, N. Das, and M. Nasipuri, “Two-phase dynamic routing for micro and macro-level equivariance in multi-column capsule networks,” *Pattern Recognition*, vol. 109, p. 107595, 2021.
- [63] P. Shiri, R. Sharifi, and A. Baniasadi, “Quick-capsnet (qcn): A fast alternative to capsule networks,” in *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*, pp. 1–7, IEEE, 2020.
- [64] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.
- [65] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

- [66] S. Chang and J. Liu, “Multi-lane capsule network for classifying images with complex background,” *IEEE Access*, vol. 8, pp. 79876–79886, 2020.
- [67] M. Amer and T. Maul, “Path capsule networks,” *Neural Processing Letters*, vol. 52, pp. 545–559, 2020.
- [68] P. K. Mensah, B. A. Weyori, and M. A. Ayidzoe, “Evaluating shallow capsule networks on complex images,” *International Journal of Information Technology*, vol. 13, no. 3, pp. 1047–1057, 2021.
- [69] S. Sridhar and S. Sanagavarapu, “Ba1,” in *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*, pp. 385–390, IEEE, 2021.
- [70] T. Ben-Nun and T. Hoefer, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surv.*, vol. 52, Aug. 2019.
- [71] B. Hayes, “Computing science: The easiest hard problem,” *American Scientist*, vol. 90, no. 2, pp. 113–117, 2002.
- [72] R. E. Korf, “Multi-way number partitioning,” in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

Appendix A

Publications

This appendix presents a list of journal and conference articles published during this Ph.D. research.

Publications which resulted from this project:

1. The Multi-Lane Capsule Network (MLCN). **Rosario VM**, Borin E, Breternitz M. IEEE Signal processing letters. 2019 May 8;26(7):1006-10.
2. Efficiency and scalability of multi-lane capsule networks (MLCN). **Rosario VM**, Breternitz Jr M, Borin E. 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'19), 2019.
3. Efficiency and scalability of multi-lane capsule networks (MLCN). **Rosario VM**, Breternitz Jr M, Borin E. Journal of Parallel and Distributed Computing. 2021 Sep 1;155:63-73. This paper was an extension from a paper publish and presented at SBAC-PAD 2019 with new contributions.

Publications which resulted from previous studies and from the collaboration with other students and researchers as a part of their work:

1. Beyond the fog: Bringing cross-platform code execution to constrained iot devices. Pisani F, Brunetta JR, **Rosario VM**, Borin E. In 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2017 Oct 17 (pp. 17-24). IEEE.
2. Uma Análise da Facilidade de Emulação de Binários RISC-V. Lupori L, **Rosario VM**, Borin E. In Anais da IX Escola Regional de Alto Desempenho de São Paulo 2018 Apr 13 (pp. 77-80). SBC.
3. Fog-assisted translation: towards efficient software emulation on heterogeneous IoT devices. **Rosario VM**, Pisani F, Gomes AR, Borin E. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2018 May 21 (pp. 1268-1277). IEEE.
4. A Methodology for Optimization of Interpreters. **Rosario VM**, Hato MM, Azevedo R, Borin E. In 2018 Symposium on High Performance Computing Systems (WSCAD) 2018 Oct 1 (pp. 205-212). IEEE.
5. Evaluation and Mitigation of Timing Side-Channel Leakages on Multiple-Target Dynamic Binary Translators. Napoli OO, **Rosario VM**, Aranha DF, Borin E. In High Performance Computing Systems: 19th Symposium, WSCAD 2018, São Paulo, Brazil, October 1–3, 2018, Revised Selected Papers 2020 Feb 13 (Vol. 1171, p. 152). Springer Nature
6. Towards a high-performance RISC-V emulator. Lupori L, **Rosario VM**, Borin E. In 2018 Symposium on High Performance Computing Systems (WSCAD) 2018 Oct 1 (pp. 213-220). IEEE.

7. Fog vs. cloud computing: should i stay or should i go?. Pisani F, **Rosario VM**, Borin E. Future Internet. 2019 Feb;11(2):34.
8. Smart selection of optimizations in dynamic compilers. **Rosario VM**, Faustino da Silva A, Aparecida Silva Camacho T, Napoli OO, Breternitz M, Borin E. Concurrency and Computation: Practice and Experience. 2020:e6089.
9. Fast and Low-cost Search for Efficient Cloud Configurations for HPC Workloads. **Rosario VM**, Camacho TA, Napoli OO, Borin E. arXiv preprint arXiv:2006.15481. 2021. Under Review.
10. Accelerating Multi-attribute Unsupervised Seismic Facies Analysis With RAPIDS. Napoli OO, **Rosario VM**, Navarro JP, Borin E. European Association of Geoscientists and Engineers, 2021.
11. Employing Simulation to Facilitate the Design of Dynamic Code Generators. **Rosario VM**, Zinsly R, Rigo S, Borin E. 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'21), 2021.
12. Predição de Desempenho com Graph Neural Networks. **Rosario VM**, Zanella AF, da Silva A, Borin E. In Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho 2020 Oct 21 (pp. 1-12). SBC.
13. New Optimization Sequences for Code-Size Reduction for the LLVM Compilation Infrastructure. da Silva, A.F., Borin, E., Pereira, F.M.Q., Nápoli, O. O., and **Rosario, VM**. 25th Brazilian Symposium on Programming Languages (SBLP), 2021.

List of projects and other relevant aspects of the Ph.D.:

1. (2017-2018) I had a scholarship from Samsung where I worked in the development of static analysis in the Android JIT compiler to detect energy bugs.
2. (2018-2019) I earn a grant to study abroad from CAPES and stayed at INESC-ID/IST in Lisbon under the supervision of Prof. Mauricio Breternitz.
3. (2019-2019) I did a Google Summer of Code internship working on the QEMU virtual machine/emulator.
4. (2019-2019) I did an internship at Microsoft Research working with the Tensorflow JIT compiler: XLA.
5. (2020-2020) I had a scholarship from Petrobras working with virtual machines resource exploration for high-performance application.
6. (2021-2021) I did and finish the 6 month DeepLearning.AI Deep Learning Specialization course.
7. (2020-current) I have been working at Cadence Design System as an AI Compiler Engineer developing a deep learn compiler for the Tensilica deep learn accelerators.

Appendix B

Source Code and Code Usage

You can access the Multi-Lane CapsNetwork source code at <https://github.com/lmcad-unicamp/lanes-capsnet>

This MLCN implementation used the @XifengGuo CapsNet Keras implementation (<https://github.com/XifengGuo/CapsNet-Keras>) as its base. All source code is available with the MIT license.

Some of the details of how to install and use it can be found in the CapsNet-Keras project README. To support multi-lanes we introduced some new command line arguments:

- **-lane_size** size of the lanes (an integer that should be greater or equal to one)
- **-lane_type** type of the lanes (1 to mcn1 and 2 for mcn2)
- **-num_lanes** number of lanes (an integer that should be greater or equal to 2)
- **-dataset** mnist or cifar10 dataset
- **-dropout** percentage of lanes being dropped out per batch