Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Transforming Convolutional Neural Networks for Model Parallelism

Author: Felix Brakel (11270381)

1st supervisor:	Ana-Lucia Varbanescu
daily supervisor:	Uraz Odyurt
2nd reader:	Tiziano de Matteis

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 30, 2024

"I am the master of my fate, I am the captain of my soul"

from Invictus*, by William Ernest Henley*

Abstract

Neural networks have become increasingly complex, driving the need for efficient execution. This need is particularly pronounced in environments where computational resources are limited and the size of the models poses a challenge to traditional execution methods. One way to address this is through model parallelism where we partition a model over multiple devices. This thesis presents a method for enhancing model parallelism in existing neural neural network architectures, specifically focusing on the Inception-ResNet-v2 model. Our method consist of 3 transformations: placing sequential cells in parallel (ParallelResNet), replacing regular convolutions with grouped convolutions (GroupedResNet), and splitting operators along the channel dimension (SplitResNet). We analyze our proposed transformations on the Inception-ResNet-v2 model and show that they introduce a straightforward way to partition the model that minimizes communication overhead and creates partitions of equal computation complexity. We conclude that the classification accuracy is reduced for ParallelResNet, and increases for GroupedResNet. SplitResNet is either unaffected or reduced depending on the dataset. However, overall the effect of the transformations on classification accuracy is very small and further evaluation on more complex classification problems is necessary to quantify it.

Contents

List of Figures

List of Tables

LIST OF TABLES

1 Introduction

In recent years the field of machine learning has continued to make significant strides. From recognizing birds by sound[1] running locally on a phone, to recognition of over 90000 flora and fauna taxa around the world using a complex model running in the datacenter[2], mainstream adoption continues to grow. Neural networks are a powerful driving force behind the advancements in machine learning, capable of learning complex patterns given a sufficiently large dataset. Despite their effectiveness on complex problems, or possibly because of it, the inner workings of neural networks are not fully understood. Yet there is a large amount of existing neural network architectures that are known to provide good results in their respective domains.

Convolutional Neural Networks (CNNs) have become a cornerstone in computer vision tasks, such as image classification, object detection, and semantic segmentation. While CNNs compared to more recent transformer architectures are relatively small; designing, training, and deploying one from a computer science perspective still has substantial requirements in both compute and memory capacity. Fortunately, neural networks contain a lot of parallelism, as their operations are mostly SIMD (Single Instruction, Multiple Data) in nature, and their control flow is mostly static. This maps very well to existing accelerators, predominantly GPUs, which have been used ever since the original AlexNet to speed up training and inference workloads (14).

To further accelerate training and inference multiple GPUs can be used. A straightforward way to do this is by storing a replica of the model on each GPU and partitioning the input data over these replicas. While this will slightly decrease the efficiency of the training process, because the weights have to be averaged at the end of every batch, the

[1]https://www.macaulaylibrary.org/2021/06/22/behind-the-scenes-of-sound-id-in-merlin/

[2]https://www.inaturalist.org/

increased computation means that we can simply train for more epochs to overcome this. In the end, this approach can still greatly increase throughput during both training and inference (15).

This however only takes advantage of the compute scaling from going parallel, not the memory scaling.While this isn't an issue for CNNs on GPUs, larger models or CNNs on more resource starved hardware do need to take advantage of this memory scaling. For this we need model parallelism, and we will explore this concept in more detail in Chapter 2 by reiterating the main findings of our previous literature study into the subject (3). For presenting the core of this thesis however we shall for now simply state that the efficacy of a particular model parallelism implementation depends on the technical implementation (both hardware and software), as well as the architecture of the model. The core hypothesis of this thesis is then that we can define generic transformations on existing CNN model architectures to make them more suitable for model parallelism, thereby opening up new deployment options that are otherwise unfeasible due to hardware limitations.

1.1 Approach

In order to test this hypothesis we address the following research questions:

1. *What transformations make an existing architecture more suitable for model parallelism?*
2. *What is the influence of these transformations on accuracy?*
3. *What is the generality of these transformations with regards to varying datasets?*

While model parallelism is used in both training and inference we limit the analysis of the transformations to the forward pass. This covers that use-case of inference, but only partially covers training. In order to answer the research questions we will propose 3 transformations on the Inception-ResNet-v2 model (19): ParallelResNet, GroupedResNet, and SplitResNet. Because an optimal model parallelism implementation is dependent not only on the model architecture, but also the hardware setup it is deployed on, we design our transformations to have a configurable degree of parallelism. We provide an analysis of the benefits of the transformation and test the accuracy on two datasets. This allows a more informed decision to be made on the tradeoffs between model accuracy and overall performance.

1.2 Outline

In Chapter 2 we explore the background and terminology of neural networks and CNNs. Section 2.2 reiterates the main findings of our previous literature study into model parallelism. In Section 2.4 we then apply both to provide an analysis and overview of Inception-ResNet-v2.

In Chapter 3 we outline our proposed solution consisting of 3 transformations. In Section 3.1 we put forth a number of requirements that the transformations should meet and introduce the conceptual framework with which we express the transformations in Section 3.2. Our proposed transformations are described and analyzed in Section 3.3, and Section 3.4 covers our implementations.

Chapter 4 discusses the training setup and the results of the transformed models on separate subsets of 100 classes of the ImageNet dataset. The conclusion to our work is presented in Chapter 5.

2

Background

In machine learning, we distinguish between two phases: training and inference. During training, we train a model on a dataset. In the inference phase, the trained model is tasked with making predictions on new, unseen data. One such model is a neural network, which dominates the state-of-the-art for complex prediction tasks. To understand the challenges our work addresses, it is necessary to understand a number of concepts about neural networks, as well as their execution model. Earlier we have conducted a study into model parallelism in neural networks (3) and the content of this thesis builds on the terminology introduced there, as well as our previous findings on the challenges of model parallelism.

2.1 Neural networks

Neural networks are a class of machine learning models inspired by the structure of the human brain. These networks are composed of layers of interconnected artificial neurons, which are mathematical functions designed to simulate the behavior of biological neurons. Neural networks have become the cornerstone of many state-of-the-art applications, including image recognition, natural language processing, and even playing games.

At a high level, a neural network consists of an input layer, one or more hidden layers, and an output layer. The input layer receives raw data, which is then processed through the hidden layers. These connections between layers are weighted, and it is through the adjustment of these weights that the network learns to map inputs to outputs.

2.1.1 Training phase

During the training phase, the neural network is exposed to a large dataset, often comprising millions of data points, such as labeled images or text samples. The network adjusts its weights based on this data to minimize the difference between its predicted output and the actual labels, a process known as backpropagation. This iterative process is computationally intensive, requiring many epochs (full passes through the training dataset) and vast amounts of computational resources.

The amount of data involved in training modern neural networks can be enormous. For example, training a network on the comparatively small ImageNet dataset already involves processing over 14 million images across 1,000 categories. Additionally the models themselves can contain millions, or even billions, of parameters that need to be optimized.

2.1.2 Inference phase

During inference, the network applies the learned weights to the input data to produce an output, such as classifying an image or translating a sentence. Unlike the training phase, inference is typically less computationally demanding, as it involves a single forward pass through the network without any weight updates. However, the scale of inference tasks can still be significant, particularly when deployed in real-time applications like self-driving cars or large-scale recommender systems, where the network may need to process thousands or millions of inferences per second.

Overall, neural networks are powerful tools that can learn from vast amounts of data, making them highly effective for a wide range of tasks. The following section will delve deeper into the technical structure of these networks.

2.2 Execution model and parallelism

Neural networks, are composed of artificial neurons organized in layers. Each neuron within a layer performs the same operation on its input data, often referred to as an 'operator'. A straightforward example of this is the fully connected layer, where every neuron is connected to all neurons in the previous layer. Operators are generally expressed as functions on n-dimensional arrays called tensors. Each operator, such as a convolution, takes one or more tensors as its input, performs a specific computation, and produces one or more output tensors. The input and output tensors are also referred to as activation tensors. This abstraction allows the neural network to be represented as a directed graph, where nodes represent the operators and their parameters, while the edges represent the data produced by these operations.

The execution model of neural networks, which is represented by this graph, is particularly useful for analyzing parallelism in neural networks. There are two main types of parallelism: inter-operator and intra-operator parallelism.

2.2.1 Inter-operator parallelism

Inter-operator parallelism assigns different operators, such as convolutional layers or fully connected layers, to different compute devices. This approach can be beneficial when a model is large and complex, allowing different parts of the network to be processed simultaneously on different devices. However, this approach requires a pipelining implementation to benefit from multiple devices. Additionally during training this approach suffers from reduced device utilizations, as there is a data dependency between the backward pass and the forward pass. An advantage of this approach is that only the tensors crossing the partition boundaries in the operator graph need to be communicated, which reduces the bandwidth requirements of this type of parallelism.

2.2.2 Intra-operator parallelism

Intra-operator parallelism, on the other hand, involves exploiting the inherent parallelism within individual operators. Since operators generally work in a SIMD (Single Instruction, Multiple Data) manner on tensors, this type of parallelism focuses on partitioning the workload within a single operator across multiple compute units. The exact partitioning strategy can vary greatly depending on the operator but generally involves scattering and gathering tensors for each data sample. This type of parallelism is typically limited to a single compute node due to the high communication overhead involved in distributing the workload across multiple devices. By considering both inter-operator and intra-operator parallelism, we can optimize the execution of neural networks in distributed or high-performance computing environments.

2.3 Convolutional neural networks

In this thesis, we focus on Convolutional Neural Networks (CNNs). CNNs are particularly well-suited for tasks that involve spatial data, such as images, making them widely used in computer vision. While recently dethroned by vision transformers[1], CNN-based architectures have dominated the imagenet benchmark ever since AlexNet was introduced in 2012 (14).

[1]https://paperswithcode.com/sota/image-classification-on-imagenet

Architecturally, the distinguishing factor of CNNs is their use of the convolution operator. This operator computes a weighted sum over a local region of the input, weights that the operator can learn from the training dataset. Unlike fully connected layers where each neuron is connected to every neuron in the previous layer, convolutional layers use a sliding window approach, where the same set of weights (called a filter or kernel) is applied to different parts of the input. This drastically reduces the number of weights in the network and allows it to learn to extract features from its input. By stacking multiple convolution operators in a row the network can learn more complex features.

Residual connections are a later addition to the CNN (9), addressing the vanishing gradients problem. This issue arises when the gradients used to update the network's weights diminish as they are propagated back through the network, leading to very small updates for the initial layers. Residual connections mitigate this by introducing skip connections, which carry outputs from earlier layers directly to later layers.

All the activation tensors of a CNN are 3-dimensional: two spatial dimensions (width and height) and one dimension for the different channels in the image. There is a fourth dimension called the batch dimension that determines how many data samples get passed through the network at a time. This is done to keep hardware utilization up and while this number does have an effect on the efficiency of the training process, we omit it here and in the analysis in this work.

As an example of a typical shape of activation tensors in a CNN: Inception-ResNet-v2's input tensor is $299 \times 299 \times 3$, one 299×299 image for every color channel, and the first convolution operator produces an output tensor of $149 \times 149 \times 32$.

2.4 Inception-ResNet-v2

The Inception-ResNet-v2 model is a hybrid architecture that combines the strengths of two influential neural network designs: the Inception architecture and the ResNet architecture. It incorporates residual connections into the Inception building blocks to facilitate faster training and improved accuracy. While the original model follows a specific configuration of these blocks, in our work, we employ these building blocks in a slightly different configuration to suit our specific objectives. As illustrated in Figure 2.1, the model is a feed-forward network with residual connections consisting of 3 main stages and a number of special blocks.

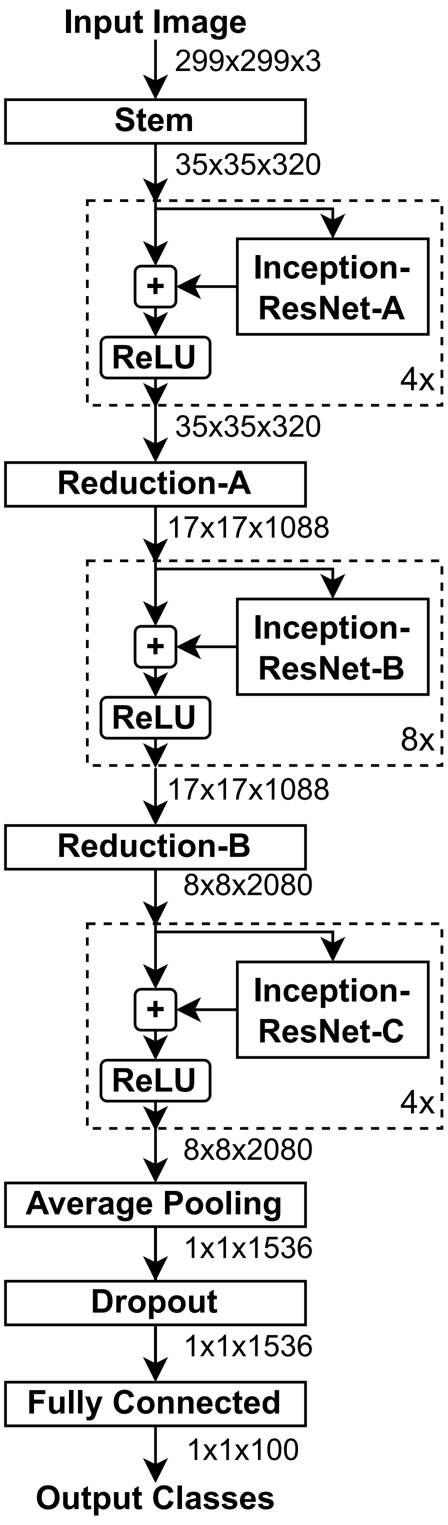


Figure 2.1: A high-level overview of different blocks in the Inception-ResNet-v2 model.

Stem: This initial part of the network processes the input image and performs basic convolution operations to reshape the input tensor for the first stage.

Inception-ResNet-A: The first stage made up of a series of Inception-ResNet-A blocks. Each block consists of three paths that are concatenated together. The input and the output of these blocks have the same shape.

Reduction-A: This block reduces the spatial dimensions and increases the channel dimension of the input data, reshaping the tensor for the next main stage.

Inception-ResNet-B: The second stage, similar to the "Inception-ResNet-A" block, but with 2 convolutional paths. Again the input and output are of the same shape.

Reduction-B: Another block that reduces the spatial dimensions and increases the channel dimension of the input data, this time to reshape the tensor for the Inception-ResNet-C blocks.

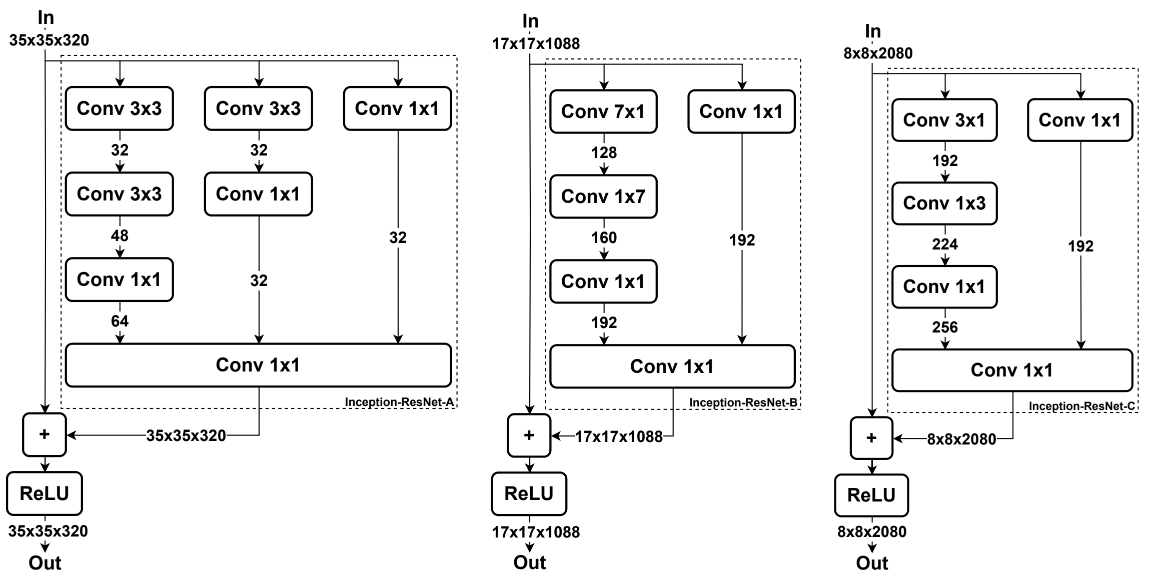
Inception-ResNet-C: The final stage of Inception-ResNet blocks in the architecture. This one features convolutions with largest kernels. Once again the input and the output of these blocks have the same shape.

Average Pooling: A global average pooling layer that reduces each feature map to a single value, summarizing the presence of features across the spatial dimensions.

Dropout: A regularization technique used to prevent overfitting by randomly setting a fraction of input units to zero during training.

Fully Connected: The final operator used for classification, going from 1536 to however many classes the model needs to distinguish between. In our case this is 100 classes.

2.4.1 Inception-ResNet-A



(a) Inception-ResNet-A. **(b)** Inception-ResNet-B. **(c)** Inception-ResNet-C.

Figure 2.2: The structures of the Inception-ResNet blocks: (a) Inception-ResNet-A, (b) Inception-ResNet-B, and (c) Inception-ResNet-C. The links between the operators represent the input-output tensors that get produced and consumed by the operators. The numbers on the links represent the size of the channel dimension at that point. The spatial dimensions and the batch dimension are not listed as they are dependent on the input data and the configuration respectively.

The Inception-ResNet-A block is a key component of the Inception-ResNet-v2 architecture. The structure of this block is depicted in Figure 2.2a. It consists of three parallel branches, each performing a series of convolutions, whose outputs are concatenated and then merged with the input of the block. The ouputs of the branches have the same spatial dimensions and are all concatenated along the channel dimension and passed through a “filter-expansion layer” (1×1 convolution without activation), which expands the channel dimension from 128 to 320. This expanded output is added to the original input via the residual connection, and followed by a ReLU activation function to introduce non-linearity.

2.4.2 Inception-ResNet-B

The Inception-ResNet-B block is another essential component of the Inception-ResNet-v2 architecture. These blocks make up the biggest part of the model. The structure of this block is depicted in Figure 2.2b and features two parallel branches instead of three, as well as significantly more channels (1088 against 320). Again the outputs of both branches are concatenated along the channel dimension and passed through a filter-expansion layer. As noted before the connection to the residual layer is identical to the Inception-ResNet-A block.

2.4.3 Inception-ResNet-C

The Inception-ResNet-C block is the last major component of the Inception-ResNet-v2 architecture, and is depicted in Figure 2.2c.The Inception-ResNet-C block, similar to the Inception-ResNet-B block, features two parallel branches. However, the convolution operations in these branches have differently sized kernels. This block has again significantly more branches, 2080, compared to the previous 1088.

2.5 Related work

The approach of altering existing neural network models to fit new requirements is well-established in the field, with various techniques focusing on different aspects of optimization. One such technique is pruning (2), which aims to reduce memory usage by selectively removing neurons from the network. This technique is particularly useful for deploying models on memory-constrained hardware. Among the different pruning strategies, *structured pruning* (8, 10), where entire channels or filters in convolutional neural networks (CNNs) are pruned based on their importance, in particular is relevant. Our work draws inspiration from this approach, as two of our proposed transformations also target the channel dimension of CNNs. Unlike pruning our method requires retraining the network.

Related to our work are a number of notions from transfer learning. Chen et al. (5) defines two transformations, 'Net2WiderNet, and 'Net2DeeperNet', which allow for adding additional operators to a pre-trained neural network. The transformed model can then be trained further for increased accuracy. While we don't share the goal of improving accuracy, the idea of transforming neural networks stems from this work. Additionally, an approach that allows us to reuse the parameters from already trained models would be beneficial to us as it requires evaluation a large amount of slight model variations. This work however is not directly applicable to ours because it relies on the transformations being local to a single layer. Our transformations are applied on multiple layers throughout the model. Another relevant transfer learning technique is model distillation (11), which reduces memory usage by training a smaller, more efficient "student" network using the output generated by a larger "teacher" network. The student network is designed to mimic the performance of the teacher network while being significantly smaller and less computationally intensive.

Although model distillation and pruning share our goal of optimizing models for specific hardware constraints, our work differs in its focus on enhancing model parallelism. Thus while pruning and model distillation focus on removing elements with minimal impact on accuracy, our approach also targets elements that hinder model parallelism, such as inter-partition connections.

A more automated approach to model optimization is found in the field of hardware-aware neural architecture search (NAS) (18). This area of research employs techniques such as reinforcement learning (7) and evolutionary computing (24) to automatically search for neural network architectures that are optimized according to a hardware cost model. While these methods excel at tailoring neural networks for single resource-constrained devices, such as a smartphone's System-on-a-Chip (SoC) or a single FPGA, they do not typically address the challenges of parallelism.

Additionally there is a large amount of research in the systems we use to parallelize models without altering their structure. GPipe (12) and PipeDream (17) both describe methods to apply pipeline parallelism to models. Although suboptimal during training this can be a straightforward way to partition the model and has been integrated in PyTorch and Tensorflow.

The XLA compiler (XLA), provides a framework for optimizing neural network execution on GPUs and TPUs. Originally introduced for TensorFlow it now has a framework built specifically for it: JAX. By enabling automatic differentiation and just-in-time compilation, JAX allows models defined in Python-based frameworks to be efficiently executed, exploiting both data and model parallelism. GSPMD (22) is built as an extension to XLA and allows for tensors to be annotated by the programmer in order to partition them over multiple devices. Alpa (23) is an automated approach to partitioning that has similarly been integrated into XLA. XLA has also been integrated in PyTorch.

Alpa is not the only work that attempts to automate the parallelization of neural networks. Auto-parallelism (16) frameworks dynamically determine the best way to parallelize the model based on the specific hardware and workload. A notable example is FlexFlow (13), a deep learning framework that uses a novel execution planning algorithm to explore and identify efficient parallelization strategies. It should be noted that while these frameworks, although they advertise wide compatibility, are still very experimental and we found that using them as a is a significant challenge in itself. More experimental approaches are TensorOpt (4), RaNNC (20), FTPipe (6), and Double Recursive (21).

While (auto-)parallelism frameworks address the challenges of model parallelism from a system-level perspective, our work focuses on architectural transformations that inherently improve a model's suitability for parallel execution. These approaches are complementary.

3

Solution Design

We propose three transformations to enhance the suitability of existing neural network models for model parallelism. Embedded within our three research questions is the notion that we can transform existing models to improve their characteristics, making them better suited for parallel execution without compromising accuracy. The transformation process involves modifying the structure of a model to produce a new variant that meets specific requirements. In our case, this means adapting the model to maximize parallelism while maintaining an acceptable level of accuracy. In this chapter, we define the transformations, and develop a conceptual framework to analyze the impact of these transformations, allowing us to evaluate their effectiveness in terms of both parallelism and accuracy. The following sections will detail the specific transformations we propose, their underlying principles, and the framework within which they are applied.

3.1 Transformation requirements

What does it mean to increase model parallelism suitability? When do we consider accuracy is maintained?

3.1.1 Increasing parallelism

As discussed in Section 2.2, there are two types of model parallelism: intra-operator and inter-operator. To make a model more suitable for either of these types, the transformations applied should address the specific challenges each type faces.

Homogeneity For a model to be effectively parallelized across multiple compute devices, it must be easy to load balance. A highly heterogeneous model, with vastly different computational requirements across its components, presents significant challenges for partitioning over multiple compute devices with similar capabilities. Thus, the transformations should aim to create computationally identical partitions, facilitating balanced distribution of the model's workload across all available devices.

Minimized communication As discussed in Chapter 2, communication between devices is a major bottleneck in distributed computing. Bandwidth and latency of links between compute devices are orders of magnitude slower than the link of a chip to its DRAM. Thus, in order to make a model more suitable for model parallelism a transformation should create partitions in such a way that the communication between them is minimized.

Configurable partitioning The transformation must be flexible in the amount of parallel partitions that are created as the desired amount can depend on various factors not considered in this thesis such as hardware configuration or specific accuracy requirements. Thus the transformations should support a configurable degree of parallelism.

Maintaining accuracy Changing the architecture of a model will invariably affect its accuracy and another requirement of the transformations is to ensure that the impact on the model's accuracy is such that it remains useful for its intended tasks. Exactly what constitutes a "useful for its intended task" will vary from case to case. We aim to minimize the loss in accuracy as much as possible to cover as many use cases as possible.

3.2 Conceptual framework

To define the transformations that meet our requirements, we must first establish a framework to formally describe the structure of a model. In Chapter 2, we explored the structure of the Inception-ResNet-V2 model as a hierarchical graph. Consecutive convolutional layers form a branch, multiple branches constitute a block, and the blocks together make up the final model. Our framework closely follows this structure and defines a model as a graph of components at various levels of abstraction.

This framework allows us to reason about a model's suitability for parallelism as per the requirements. By describing it as a graph of operators, we can partition the graph and assess the homogeneity of the partitions by looking at the combined computational load of the operators within them. Communication requirements can be assessed by examining the tensors on the edges that cross partition boundaries. Additionally, this framework allows us to define operations on the structure in a generic manner, such that the transformations can be applied to models other than the one used in our case study.

Lastly there is the practical consideration of keeping track of the many models produced by the transformations in this study, therefore we also introduce a naming scheme based on the concepts introduced in this section.

3.2.1 Abstraction levels: operators, components, macro graph

Because neural networks are made up of many operators, we will describe their structure at various levels of abstraction. This approach allows us to maintain oversight in these massive networks. At the lowest level of abstraction, we define operators. These operators include a two-dimensional convolution operator, or *Conv* for short, and a fully connected layer using the rectified linear unit ($ReLU$) as the activation function. We also define more straightforward operations, such as addition, which performs an element-wise addition of multiple identically-sized tensors, and concatenation, which combines two or more tensors into a single one along a certain axis.

These operators are connected together in a directed graph to form a custom component, or *block*, such as the Inception-ResNet-A block. A component takes, like an operator, a tensor as input and produces a tensor as an output. Components are connected together in what we call a *macro graph* to form the final network. Thus, there are two levels of abstraction: operators, which form components, and components, which form the model as defined by the macro graph.

3.2.2 Naming scheme

We name our model variants based on the macro graph. For example, the Inception-ResNet as described in the original paper has 5 Inception-ResNet-A blocks, 10 Inception-ResNet-B blocks, and 5 Inception-ResNet-C blocks. Thus, we would name this model ResNet_5_10_5. In our work, because we are dealing with degrees of parallelism, we like to have this easily divisible. Consequently, we compare to a ResNet variant we call ResNet_4_8_4, which consists of 4 Inception-ResNet-A blocks, 8 Inception-ResNet-B blocks, and 4 Inception-ResNet-C blocks.

Since the transformations can produce various models based on the configured amount of parallelism we also include this parameter in the name. For example: Parallel-ResNet-

2-4-8-4 is the model produced by the first proposed transformation (Section 3.3.1) with 2 degrees of parallelism, 4 A, 8 B, and 4 C blocks.

3.3 Proposed model transformations

We propose 3 transformations: placing sequential cells in parallel, replacing regular convolutions with grouped convolutions, and splitting the entire cell along the channel dimension. Because the largest part of the network is made up of the repeated instances of the A, B, and C blocks we only apply our transformations to these parts of the network.

To properly argue that these transformations are an effective way to increase a CNN's suitability for model parallelism would require applying the transformation to many different models.

This task would demand an amount of compute that is not available to us. Thus we choose to provide insight into the transformations by conduction an in-depth case study in applying them to the Inception-ResNet-v2 model.

3.3.1 ParallelResNet: place sequential cells in parallel

To address partition homogeneity in the Inception-ResNet architecture, we propose a transformation that increases the network's width while reducing its depth. Currently, the model organizes cells in a feed-forward sequence, where each cell processes the output of its predecessor. For instance, in the initial stage, 8 Inception-ResNet-A cells are processed sequentially. Our transformation places cells into a parallel structure where every cell gets the same input and their outputs are added together. By consolidating 2 or more cells into a single parallel block, we remove the data dependency between the cells.

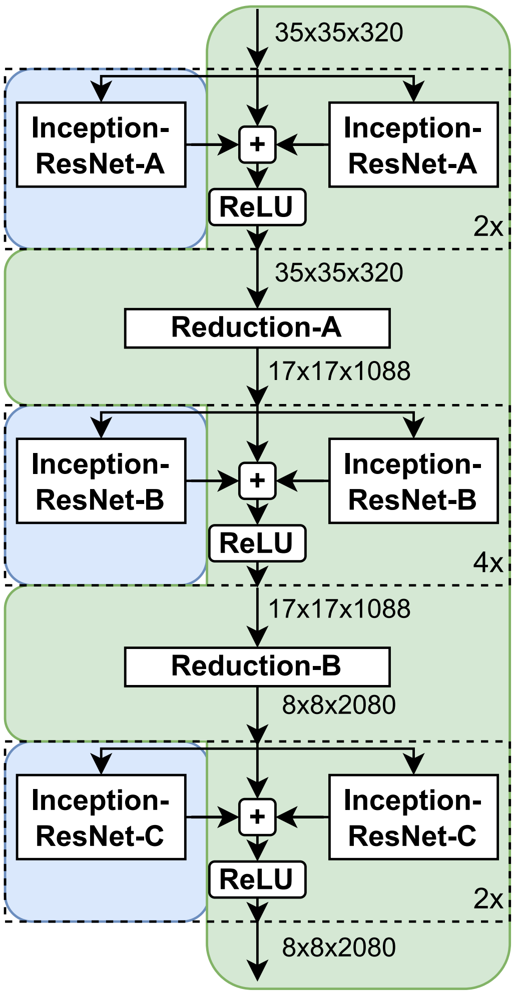


Figure 3.1: ParallelResNet with 2 partitions. The area marked in green is the main partition.

This means we can exploit parallelism over multiple cells instead of the over the branches within a cell.

To illustrate this transformation, consider the Inception-ResNet-A subsection. In the original model, Inception-ResNet-A is executed in sequence four times. By increasing the width, we duplicate the block and place it in parallel within a single stage, effectively merging multiple Inception-ResNet-A cells into a wider structure. This parallel architecture, as shown in Figure 3.1, simplifies load balancing to assigning an equal number of cells to each compute device, since all of these are identical in computational requirements. Device utilization is also increased by assigning full cells, allowing compute devices to efficiently manage larger and more consistent workloads. An important consideration in this transformation is determining the optimal number of cells to place in parallel, which directly impacts workload distribution, device utilization efficiency, and overall model performance in diverse computational environments.

3.3.1.1 Fulfillment of requirements

Partitioning Figure 3.1 shows the proposed partitioning scheme for this transformation. This partitioning scheme is limited by the amount of sequential blocks, meaning we can't go above 4 degrees of parallelism. The main partition needs to do slightly more work as it has to sum the tensors and process the activation function.

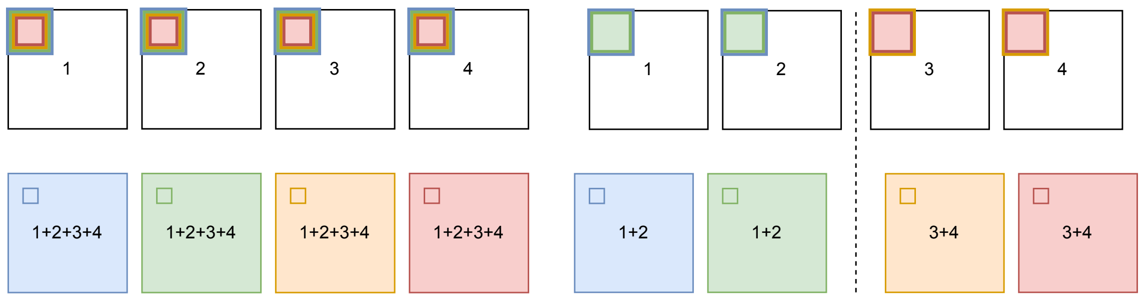
Homogeneity Since we've placed identical cells in a parallel structure, we can now assign entire cells to compute devices since there is no more data dependency between them. This significantly improves the homogeneity of the partitions. There is still a slight imbalance between them in the processing of the results of the cells. These are added together in an element-wise fashion, together with a residual connection, and then passed through an activation layer every parallel stage. While the element-wise addition could be done through a sum-reduction, the activation layer at least has to be computed on a single device. Still, compared to the computational load of the cells, this is relatively minimal.

Communication For communication, the input to a parallel cell needs to be broadcast to all partitions since every subcell needs this entire input. And then, when we add them all together, we need to send all these output tensors of the cells back to one partition to be calculated. This means that for every parallel cell, $P-1$ input tensors need to be send to the partitions (The main partition does not need to communicate with itself), and we need to then also receive $P-1$ output tensors on the main partition. This then repeats for every sequential repetition. We can do a similar calculation for the other 2 stages with their respective input tensor sizes to calculate the total communication overhead.

$$\begin{array}{ccccccccc}
\overbrace{35 \times 35 \times 320}^{\text{input tensor size}} & \times & \overbrace{(P-1)}^{\text{non-main partitions}} & \times & \overbrace{4/P}^{\text{sequential repetitions}} & \times & \overbrace{2}^{\text{send \& receive}} & + \\
17 \times 17 \times 1088 & \times & (P-1) & \times & 8/P & \times & 2 & + \\
8 \times 8 \times 2080 & \times & (P-1) & \times & 4/P & \times & 2 &
\end{array}$$

For 2 partitions the total communication requirements for a single forward pass would be $1,568,000 + 2,515,456 + 532,480 = 4,615,936$.

3.3.2 GroupedResnet: replace regular convolutions with grouped convolutions



(a) Full Convolution **(b)** Grouped Convolution with 2 groups

Figure 3.2: A full convolution (a) and a grouped convolution (b). Both convolution have 4 input channels (1, 2, 3, and 4) and 4 filters producing 4 output channels (blue, green, orange, and red). While the full convolution takes all the input channels for all filters, the filters in the grouped convolution only operate on the input channels in their respective groups.

As discussed in Chapter 2 intra-operator parallelism has a very large communication overhead. We address this by transforming convolutions into grouped convolutions in the 3 main blocks. Grouped convolutions modify the traditional convolution process by dividing the input channels into distinct groups, each processed by a separate subset of filters. A grouped convolution operation partitions the input channels into separate groups and associates each group with its own set of convolutional filters. This is in contrast to a full convolution which employs filters that interact with all input channels.

For example, consider a convolution scenario with four input channels labeled 1, 2, 3, and 4. In a full convolution setup (Figure 3.2a), each filter processes information from all four input channels to produce an output channel. This means that every output channel results from a combination of all input channels processed through each filter.

In a grouped convolution, however, the same four input channels are divided into groups. Figure 3.2b shows what this looks like for 2 groups. Channels 1 and 2 are in the first group, and channels 3 and 4 in the second. The filters assigned to the first group (blue and green) only process data from channels 1 and 2. Similarly, filters in the second group (orange and red) only process data from channels 3 and 4. This setup restricts the data each filter can access, resulting in output channels that are a product of the data each group independently.

This method of partitioning not only reduces the computational demands of each filter but also reduces the communication overhead required when distributing the computation over multiple devices because only a part of the input tensor is for every group. Furthermore if the number of groups is kept constant for consecutive convolutions the input channel in the group of the next convolution are the output channels produced by the same group of the previous convolution, meaning no communication is necessary between the devices at this point.

Grouped convolutions, while efficient, can reduce the model's expressive power by limiting each filter's access to the full range of input features, potentially missing important cross-channel interactions. This approach might also lead to a slight decrease in accuracy and increased implementation complexity, particularly when handling uneven channel groups. Additionally, the reduced data per filter could raise the risk of overfitting, as the model may become too specialized to the patterns within each group.

Although the number of groups in a convolution is implemented as just a parameter on the 'Conv' operator in PyTorch we can represent the concept in the operator graph as a separate convolution operator for every group, operating on an input tensor that has been split.

Figure 3.3 shows what ResNet-A block looks like if we use 2 groups for every convolution. Because the convolutions in the branches all use the same group count this results in 2 subgraphs without any data dependencies. The macro graph is identical to regular ResNet.

3.3.2.1 Fulfillment of requirements

Partitioning Figure 3.3 shows the partitioning scheme of Inception-ResNet-A using grouped convolutions. As long as the partition count clearly divides all the channel counts within the cell, which for powers of 2 up to 32 is the case, we can configure our partitioning without any problems. If the amount of partition does not clearly divide these groups, we

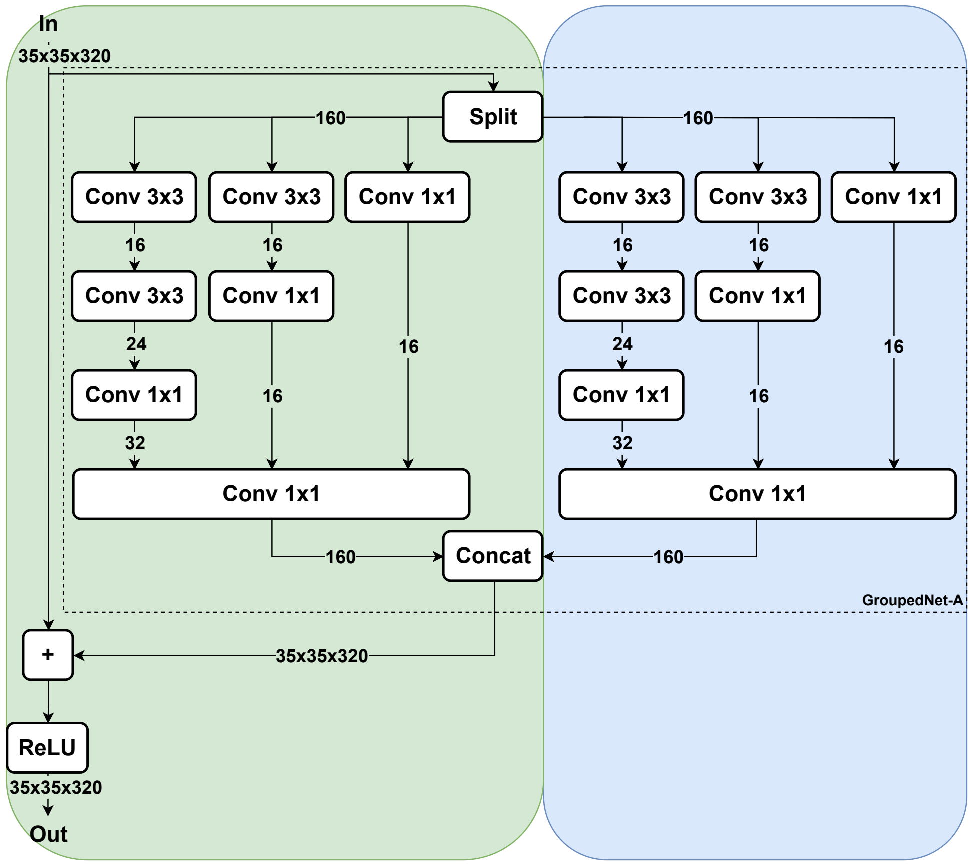


Figure 3.3: Inception-ResNet-A block using grouped convolutions such as shown in Figure 3.2b. While these are usually implemented as a single operator, we express it as multiple operators in the graph here.

need to handle the edge cases in which one of the groups is smaller, which would create additional implementation work.

Homogeneity The partitions consist of the groups in the convolution layers, as well as the filter expansion layer for the main partition, and the residual connection. Compared to the previous transformation where we placed cells in parallel, these partitions are less homogeneous. Since we added the filter expansion layer to the main partition, as well as the addition of the output tensors and the activation layer, we do not, however, need to add as many output tensors, since we only add the output of the filter expansion layer to the residual connection.

Communication Since every group in the grouped convolution only needs the channels of the input tensor that are relevant to said group, this method reduces the communication requirements substantially compared to the previous. Continuing with the Inception-ResNet-A block as example, every partitions takes $320/P$ channels and there are $P-1$ partitions that need their input tensor communicated to them. Thus the total communication cost for the input tensors is: $35 \times 35 \times (P-1) \times (320/P)$. The communication requirements for the output tensors can be computed similarly: $35 \times 35 \times (P-1) \times ((64/P)+(32/P)+(32/P))$. The total communication within a single Inception-ResNet-A block then is: $35 \times 35 \times (P-1) \times (128+320)/P$. For the entire model the communication requirements are:

$$\begin{array}{ccccccc}
\lq\overbrace{35 \times 35 \times (128+320)/P}^{\text{elements per partition}} & \times & \overbrace{(P-1)}^{\text{non-main partitions}} & \times & \overbrace{4}^{\text{sequential repetitions}} & + \\
{}'17 \times 17 \times (384+1088)/P & \times & (P-1) & \times & 8 & + \\
8 \times 8 \times (448+2080)/P & \times & (P-1) & \times & 4 &
\end{array}$$

For 2 partitions the total communication requirements would be $1,097,600 + 1,701,632 + 323,584 = 3,122,816$. A significant improvement over ParallelResNet.

3.3.3 SplitResNet: splitting entire blocs along the channel dimension

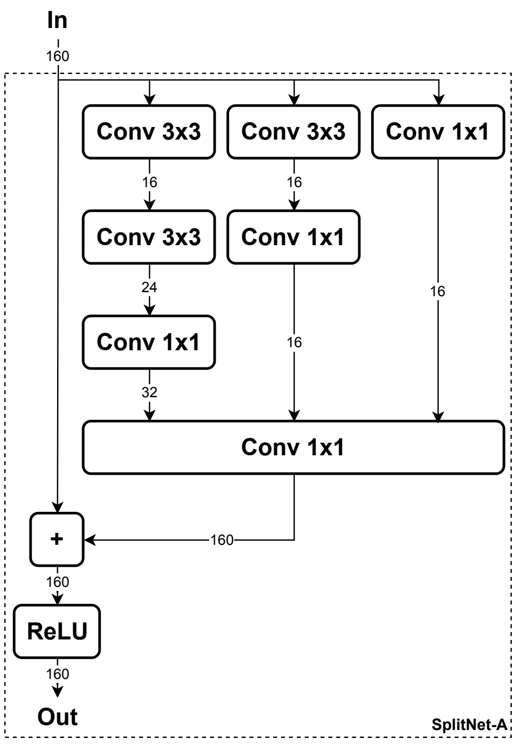


Figure 3.4: Split-ResNet-A block

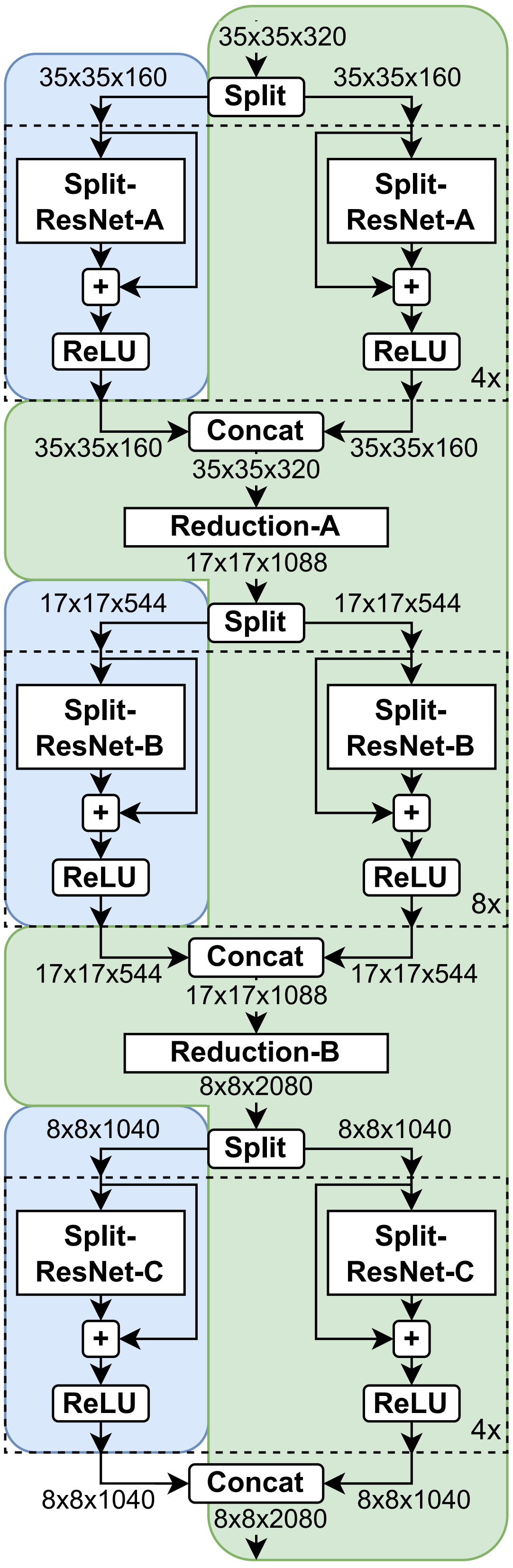


Figure 3.5: Model after SplitNet transformation with two partitions.

To further reduce communication we propose extending the concept of grouped convolutions to the rest of the cell by splitting all the layers along the channel dimension. This eliminates the need for communication not only within a single instance of a block but also between consecutive blocks as long as the group count remains constant. Figure 3.4 shows what the new SplitNet-A block looks like for the case of 2 partitions. Compared to Figure 2.2a the channel dimension has been reduced by a factor equal to the amount of partitions, in this case it is halved throughout the block.

Figure 3.5 shows SplitNet-A, B, and C placed in the macro graph.

3.3.3.1 Fulfillment of requirements

Partitioning The proposed partitioning scheme for this transformation can be seen in Figure 3.5. It's limitations are similar to that of GroupedResNet in that the degree of parallelism has to cleanly divide the channel counts. A benefit is that within the repeated sections there is no load imbalance.

Homogeneity Cells are completely identical, even the residual connection is split now. The split, and concatenation of the tensors are very cheap operations. The reduction cells are still done on main partition.

Communication This transformation eliminates all communication within the three major stages, only requiring the split tensors be distributed across the partitions at the start and gathered at the end. Thus in total for P partitions $3 \times (P-1) \times 2$ tensors have to be communicated. The total amount of tensor values that need to be communicated for P partitions is:

$$\begin{array}{cccccc}
\overbrace{35 \times 35 \times (320/P)}^{\text{elements per partition}} & \times & \overbrace{(P-1)}^{\text{non-main partitions}} & \times & \overbrace{2}^{\text{send \& recv}} & + \\
17 * 17 * (1088/P) & \times & (P-1) & \times & 2 & + \\
8 * 8 * (2080/P) & \times & (P-1) & \times & 2 &
\end{array}$$

This shows that, similar to GroupedResNet, the size of these tensors goes down as the amount of partitions increases.

For two partitions this would amount to $392,000 + 314,423 + 133,120 = 839,552$.

3.4 Implementation

Implementation was done in PyTorch. For the Inception-ResNet-v2 model, we used a publicly available implementation on GitHub[1]. The transformed models each have their own implementation.

Grouped ResNet was implemented simply by using the `groups` parameter in PyTorch's `Conv2d` operator. Parallel ResNet was implemented by wrapping the original ResNet blocks into a new parallel block that distributes the input to sub-blocks and then gathers the output. Split ResNet was created by using the parallel block from Parallel ResNet with new variants of the Inception-ResNet A, B, and C blocks, where all their channel counts are divided by the number of parallel partitions.

3.4.1 Automatic parameter exploration

Transformed models can be instantiated using a configurable number of parallel partitions, as well as configurable repetitions of the A, B, and C blocks. Given the large amount of configurations that need to be explored we have a wrapper script for the models that reads a list of parameters from a file and trains a model with every set of parameters.

[1]https://github.com/FelixBrakel/HPCNas

4

Solution Evaluation

4.1 Short, and automated training setup

The models presented in this thesis were implemented using PyTorch, a widely used deep learning framework. The training of these models was conducted on the Distributed ASCI Supercomputer 6 (DAS-6) (1). For every model we leveraged 4 NVIDIA A4000 GPUs placed on a single node, arranged in a data-parallel configuration[1]. With the DAS-6 having two such nodes, we were able to train up to two models at a time. To further accelerate the training process, we utilized automatic mixed precision (AMP)[2] during the training sessions. AMP is a technique that automatically uses lower precision datatypes where possible to reduce memory usage and improve throughput. As noted in Section 2.4 Incetpion-ResNet-v2 was trained and tested on the ImageNet dataset. This dataset contains 146GiB worth of images grouped into 1000 classes. Training all 6 model variants as well as the baseline ResNet model on this 146GiB dataset on the available hardware is not feasible for us. Thus we have come up with a shortened training setup where we train and evaluate our models on a subset of 100 out of the 1000 classes in the ImageNet dataset, which were selected randomly. This both reduces the time to do an epoch as well as the total amount of epochs needed to train the models.

In order to automate evaluation of the models some system is needed with respect to the learning rate. Because every model behaves slightly differently, we utilize a dynamic learning rate that decays by a factor of 0.8 every time the validation accuracy does not show an improvement for 2 epochs. If the validation accuracy does not improve after decaying the learning rate three times, then the training process stops. This trains all the models

[1]https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html

[2]https://pytorch.org/docs/stable/amp.html

until convergence. While it is possible to manually optimize this training setup in order to speed up training further, this is not the goal of this training setup. The goal of this study is to obtain insight of the performance *difference* of all the models, while requiring no manual intervention in the training process. Therefore a training setup that manages to showcase this differences for all models, without manually having to tune the learning process for every model, is sufficient.

We test our 3 transformed models: ParResNet, GroupedResNet, and SplitResNet, at 1, 2, and 4 degrees of parallelism. With a single degree of parallelism being the same as the baseline ResNet and serving as validation for the results.

Model	**Degree of Parallelism**	**Inception-ResNet-A**	**Inception-ResNet-B**	**Inception-ResNet-C**
ResNet	1	4	8	4
ParResNet	1	4	8	4
	2	2	4	2
	4	1	2	1
GroupedResNet	1	4	8	4
	2	4	8	4
	4	4	8	4
SplitResNet	1	4	8	4
	2	4	8	4
	4	4	8	4

Table 4.1: Model Configurations: Parallelism and Number of Inception-ResNet Blocks

Table 4.1 lists the structure of all ResNet model variants we evaluate, with their degree of parallelism and the number of Inception-ResNet-A, B, and C blocks. In total we train and evaluate 10 models: one baseline and three times three transformed models. The 1_4_8_4 variants across all models have the same configuration as the regular ResNet, serving as a sanity check to ensure the consistency of the modifications. All models, except ParResNet, maintain the structural design of the baseline ResNet while varying the degree of parallelism. ParResNet differs by distributing the same total number of blocks across the stages in a parallel manner. This parallelism reduces the depth of the network, as fewer blocks are stacked sequentially in each stage.

Three-fold cross-validation was employed to evaluate the performance of the different ResNet model variants. In this approach, the dataset was split into three subsets, train, validation, and test, in 3 different non-overlapping ways or “folds”. Each model was then trained and evaluated three times on every fold of which we take the average. This method helps to mitigate any bias resulting from the 3-way split and provides a more reliable measure of how the models are expected to perform on unseen data.

4.2 Classificantion Accuracy

4.2.1 Parallel ResNet

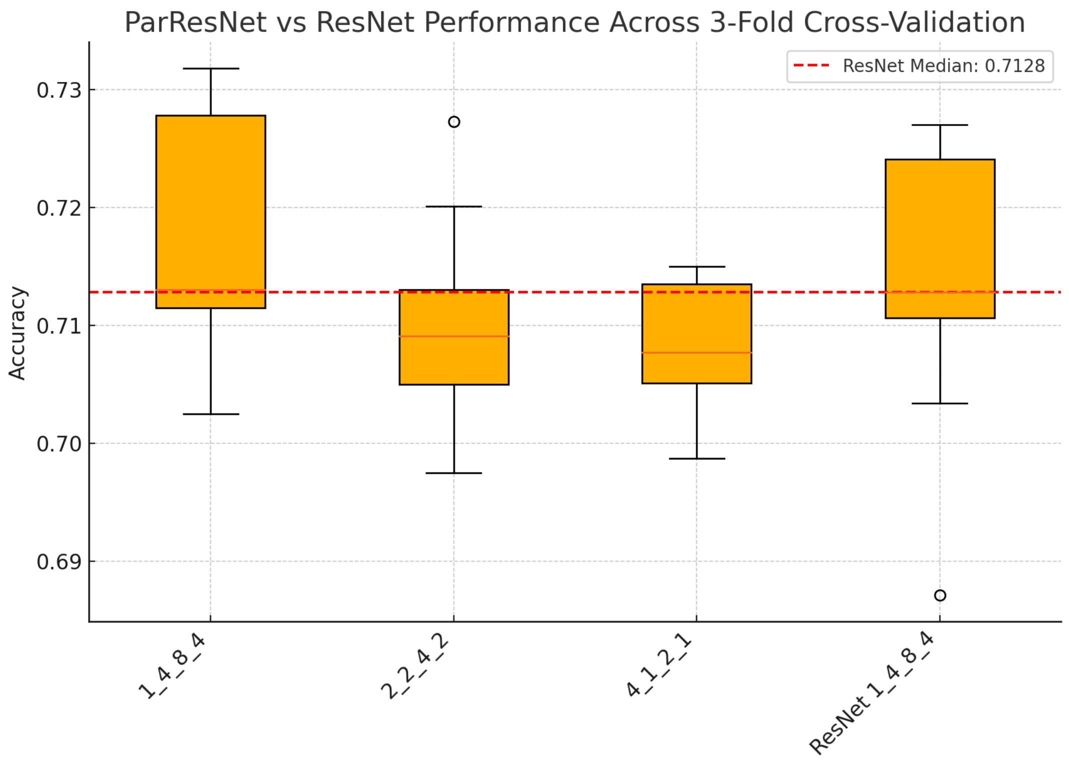


Figure 4.1: Parallel ResNet top1k accuracy against ResNet

Figure 4.1 compares the performance of different ParallelResNet configurations against the baseline ResNet model across three-fold cross-validation. The horizontal red dashed line indicates the median accuracy of the ResNet configuration (1_4_8_4), which serves as a baseline reference. As mentioned before, the 1_4_8_4 ParResNet configuration is identical to the baseline and serves as a sanity check. The median of both models are almost identical (0.7128 and 0.713) so this passes the sanity check. The 2_2_4_2 and 4_1_2_1 configurations of ParResNet have median accuracies slightly below the ResNet baseline (0.7091 and 0.7077 respectively), with each increase in parallelism slightly decreasing the classification accuracy. Interestingly, the variability observed in ParResNet's configurations suggests that while the newly introduced parallelism might reduce overall accuracy, it does perform more consistently compared to the baseline as the first quartile of these models are significantly closer to the median. Overall, while classification accuracy decreases, the difference between the model variants is very small ($\leq 0.51\%$).

4.2.2 Grouped ResNet

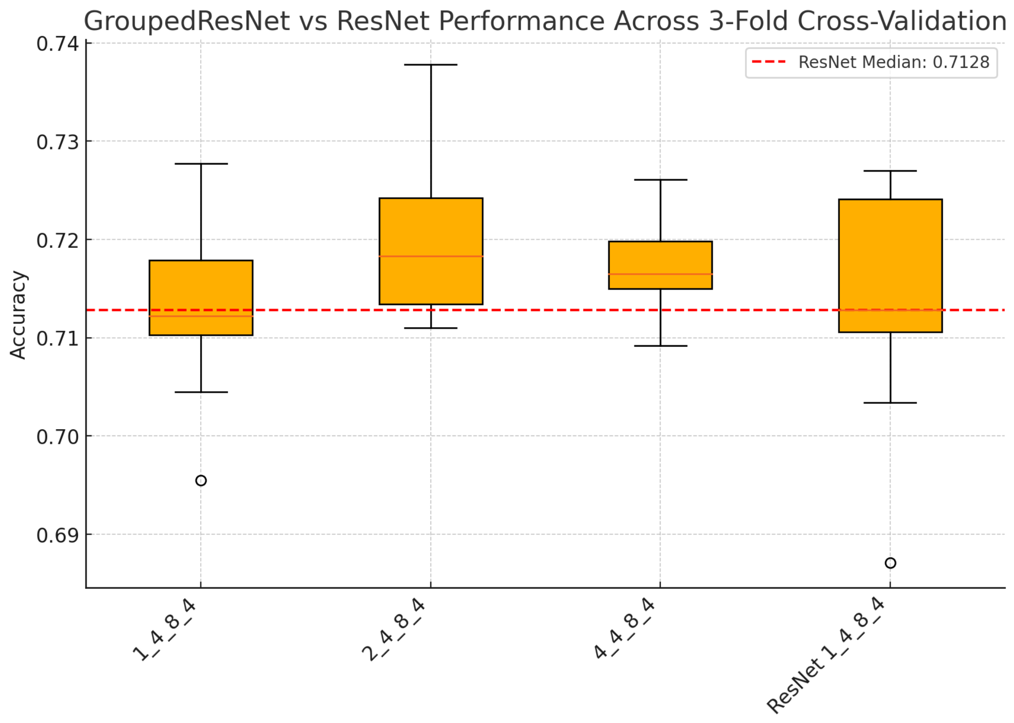


Figure 4.2: Grouped ResNet top1k accuracy against ResNet

Figure 4.2 compares the performance of different GroupedResNet configurations against the baseline ResNet model across three-fold cross-validation. Again, the red dashed line indicates the median accuracy of the baseline. The sanity check configuration of GroupedResNet, shows a slightly lower median accuracy of 0.7122 compared to the baseline. This minor difference still passes our sanity check. The 2_4_8_4 and 4_4_8_4 configurations of GroupedResNet demonstrate median accuracies of 0.7183 (the highest of all the models) and 0.7165, respectively, both slightly above baseline. This indicates that introducing more groups in these configurations can potentially lead to improved performance. Overall the GroupedResNet configurations offer a small accuracy improvement over the baseline ResNet. The difference in performance between the model variants is relatively small ($\leq$ 0.55%), but with the increased suitability for parallelism this transformation offers benefits on both fronts.

4.2.3 Split ResNet

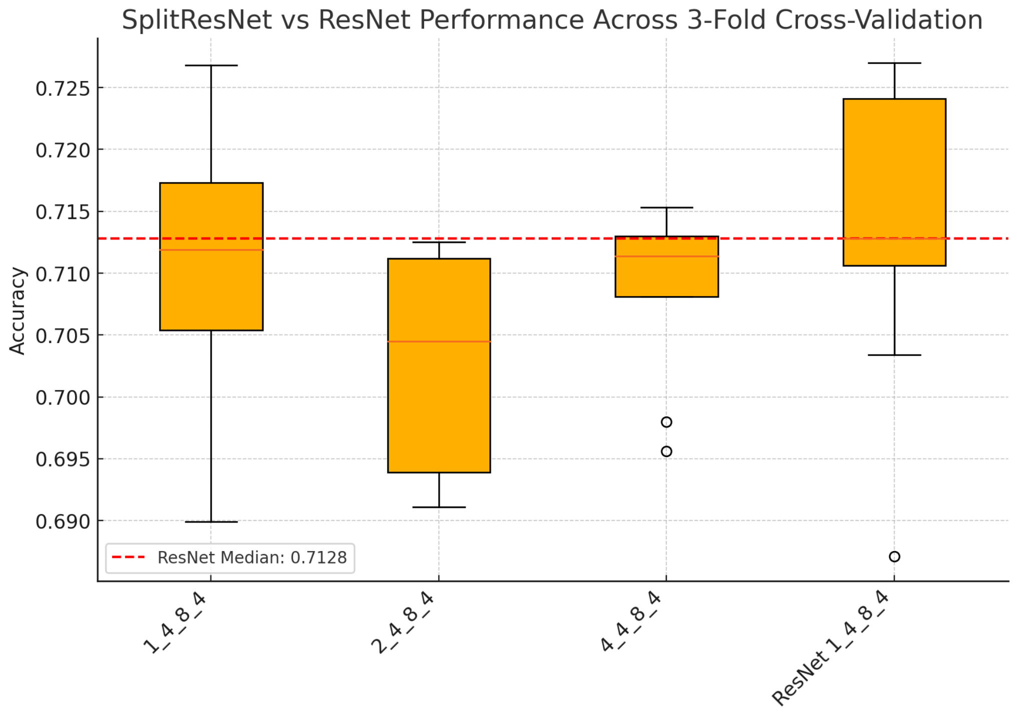


Figure 4.3: Split ResNet top1k accuracy against ResNet

Finally, Figure 4.3 compares the performance of different SplitResNet configurations against the baseline. The 1_4_8_4 configuration of SplitResNet shows a median accuracy of 0.7119. The 2_4_8_4 and 4_4_8_4 configurations demonstrate median accuracies of 0.7045 (the lowest of all the models) and 0.7114, respectively. Both configurations have medians that are slightly below the baseline. The variability in the SplitResNet configurations is more pronounced, particularly in the 1_4_8_4 setup, which shows a broader interquartile range compared to the other configurations. This suggests that while the SplitResNet approach can maintain performance close to the baseline, the split architectures introduce more inconsistency.

Thus, while the SplitResNet configurations generally perform close to the baseline, they introduce slightly more variability. Still the overall difference in accuracy between SplitResNet variants and the baseline is small ($\leq$ 0.83%), but larger than the other models.

4.3 Accuracy on second dataset

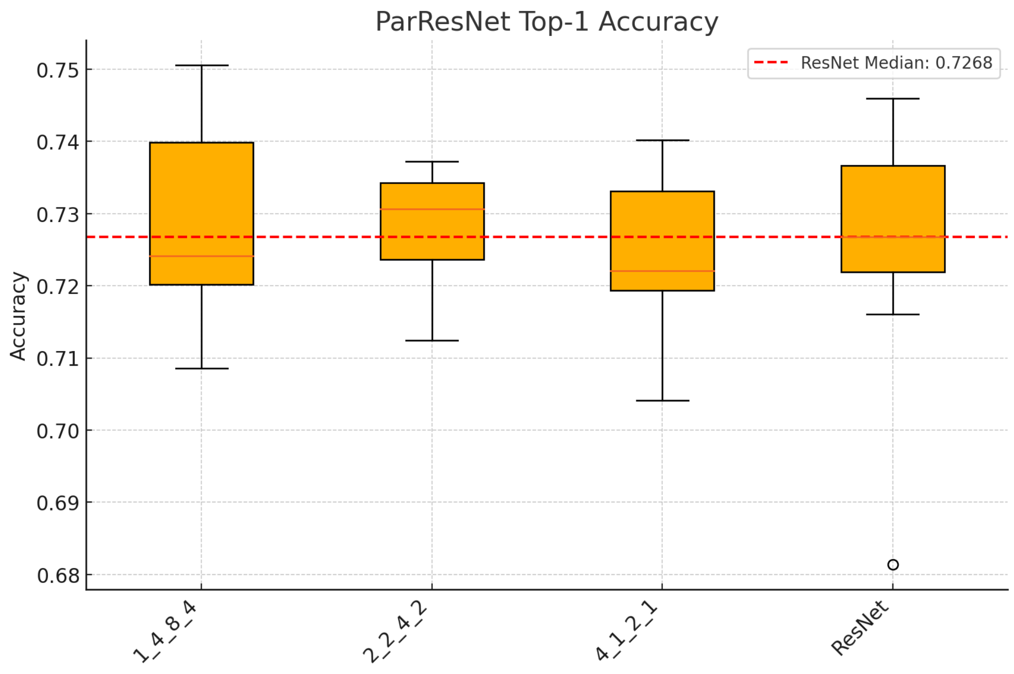


Figure 4.4: Grouped ResNet top1k accuracy against ResNet

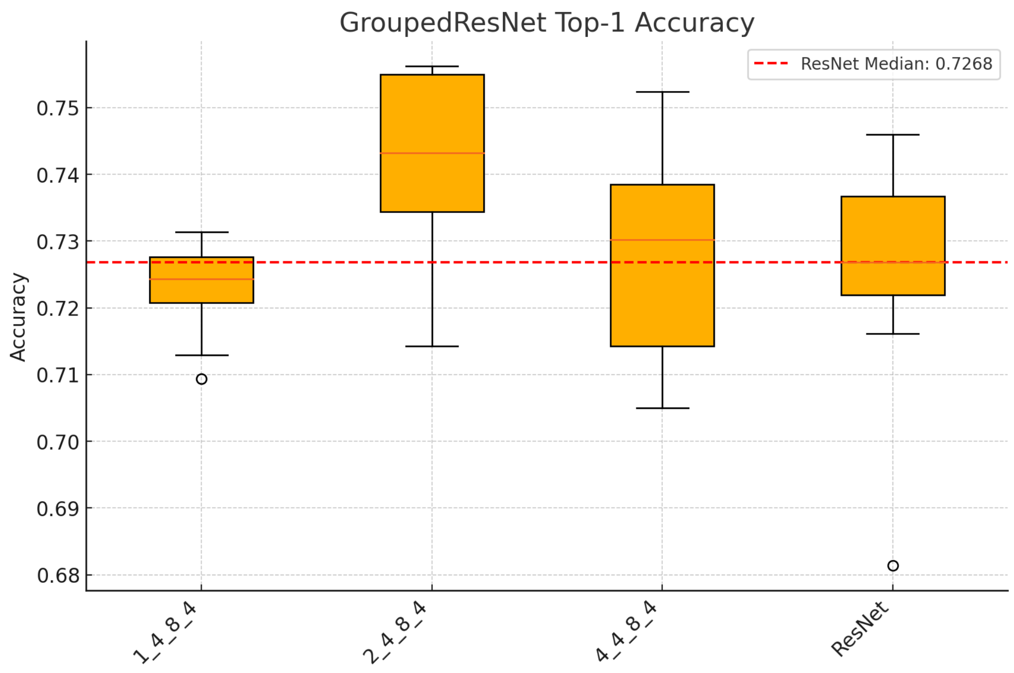


Figure 4.5: Grouped ResNet top1k accuracy against ResNet

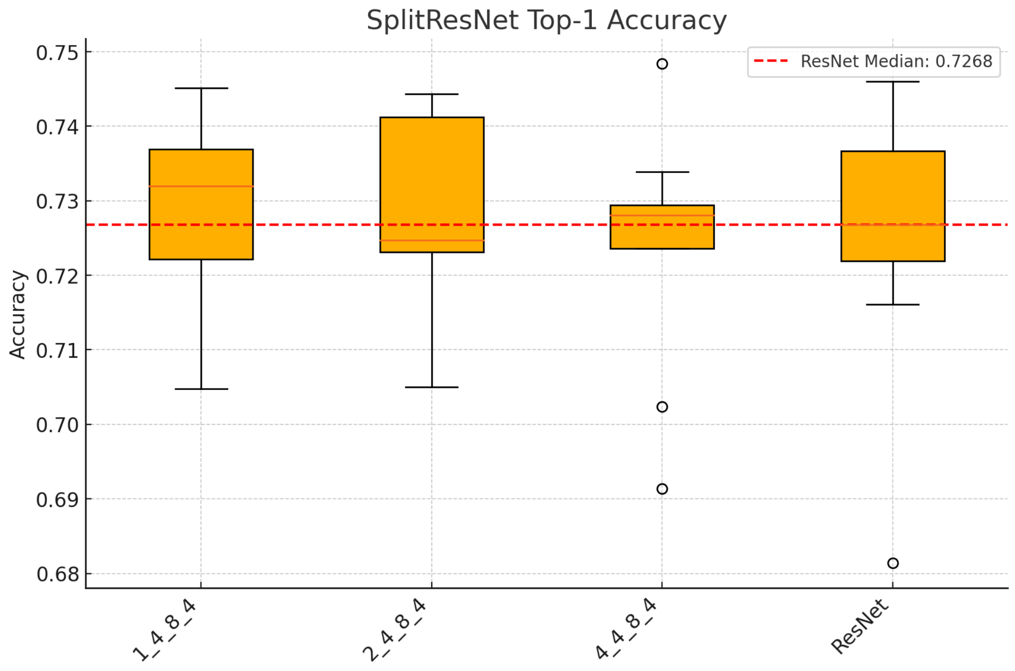


Figure 4.6: Split ResNet top1k accuracy against ResNet

Overall all the models perform slightly better on the second dataset and show a smaller overall spread in results. The ResNet baseline had a median accuracy of 0.7268. GroupedResNet in its 2_4_8_4 configutation is the only one that shows a significant difference compared to ResNet and performs best of all with a median of 0.7432. It also performed best on the first dataset. ParResNet 4_1_2_1, which was second lowest on the first datset, performs worst at 0.7221, only 0.0047 more than ResNet. SplitResNet performs significantly better on the second dataset and is within margin of error of ResNet across its configurations. A significant improvement compared to its performance on the first dataset where it performs worst of all.

5 Conclusion

This thesis delves into the challenges and opportunities involved in optimizing neural network models for model parallelism. To address the first research question, *"What transformations make an existing architecture more suitable for model parallelism?"*, we proposed three key transformations: ParallelResNet, which places sequential cells in parallel; GroupedResNet, which replaces regular convolutions with grouped convolutions; and SplitResNet, which splits entire blocks along the channel dimension. Each transformation was designed with a specific partitioning scheme in mind, aimed at minimizing communication overhead and ensuring a balanced distribution of computational complexity across partitions, thereby enhancing the model's suitability for parallel execution.

To explore the second research question, *"What is the influence of these transformations on accuracy?"*, we conducted a case study on the Inception-ResNet-v2 model, applying our transformations to introduce two-way and four-way parallelism. This study allowed us to evaluate the impact of the proposed transformations on model accuracy, demonstrating that while there may be some trade-offs, the overall performance of the transformed models remained competitive. Our findings confirm that these transformations can be effectively applied without significantly compromising the model's predictive capabilities.

Finally, we addressed the third research question, *"What is the generality of these transformations with regards to varying datasets?"*, by training the transformed models on two distinct datasets, each consisting of 100 classes. The results showed that our transformations are generalizable across different datasets, reinforcing the broader applicability of our approach. These insights suggest that the proposed transformations are not only effective but also versatile, making them valuable tools for optimizing neural network models for parallelism in a variety of contexts.

5.1 Future work

As noted in Section 3.1 there is a minimum accuracy that our transformations should exceed in order to be useful. However we found that on our classification task of 100 classes all models perform very close to each other, making it hard to show the true impact of our transformations. It is possible to adjust the model such that it shows more pronounced differences on the simplified problem, this required simplifying the stem and reduction cells, reducing the channel dimension by a factor of 4 across the entire model, and reducing the size of the final fully connected layer at the end by a factor of 4 as well. We consider these changes to be so drastic that we have opted not to use this model as a baseline, we do report the findings of our experiments in Section 5.1. However a more robust way to test the influence of the transformations on accuracy would be to train and evaluate on a more complex dataset, the full 1000 classes of ImageNet for example.

References

[XLA] XLA - TensorFlow, compiled- Google Developers Blog. https://developers.googleblog.com/en/xla-tensorflow-compiled/. 12

[1] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer*, 49(5): 54–63, May 2016. ISSN 1558-0814. doi: 10.1109/MC.2016.127. 27

[2] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the State of Neural Network Pruning? March 2020. doi: 10.48550/arXiv.2003.03033. 11

[3] Felix Brakel, Uraz Odyurt, and Ana-Lucia Varbanescu. Model Parallelism on Distributed Infrastructure: A Literature Review from Theory to LLM Case-Studies. March 2024. doi: 10.48550/arXiv.2403.03699. 2, 5

[4] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training With Auto-Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33 (8):1967–1981, August 2022. ISSN 1045-9219, 1558-2183, 2161-9883. doi: 10.1109/ TPDS.2021.3132413. 13

[5] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2Net: Accelerating Learning via Knowledge Transfer. April 2016. doi: 10.48550/arXiv.1511.05641. 11

[6] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 381–396, 2021. ISBN 978-1-939133-23-6. 13

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution. February 2019. doi: 10.48550/arXiv.1804.09081. 12

[8] Determine Filters'Importance. Pruning Filters for Efficient ConvNets. 2016. 11

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. December 2015. doi: 10.48550/arXiv.1512.03385. 8

[10] Yihui He, Xiangyu Zhang, and Jian Sun. Channel Pruning for Accelerating Very Deep Neural Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017. doi: 10.48550/arXiv.1707.06168. 11

[11] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. March 2015. doi: 10.48550/arXiv.1503.02531. 12

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. July 2019. doi: 10.48550/arXiv.1811.06965. 12

[13] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. July 2018. doi: 10.48550/arXiv.1807.05358. 12

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. 1, 7

[15] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. June 2020. doi: 10.48550/arXiv.2006.15704. 2

[16] Peng Liang, Yu Tang, Xiaoda Zhang, Youhui Bai, Teng Su, Zhiquan Lai, Linbo Qiao, and Dongsheng Li. A Survey on Auto-Parallelism of Large-Scale Deep Learning Training. *IEEE Transactions on Parallel and Distributed Systems*, 34(8):2377–2390, August 2023. ISSN 1558-2183. doi: 10.1109/TPDS.2023.3281931. 12

[17] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019. doi: 10.1145/3341301.3359646. 12

[18] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions. March 2021. doi: 10.48550/arXiv.2006.02903. 12

[19] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. August 2016. doi: 10.48550/arXiv.1602.07261. 2

[20] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. Automatic graph partitioning for very large-scale deep learning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1004–1013. IEEE, 2021. doi: 10.48550/arXiv.2103.16063. 13

[21] Haoran Wang, Chong Li, Thibaut Tachon, Hongxing Wang, Sheng Yang, Sébastien Limet, and Sophie Robert. Efficient and Systematic Partitioning of Large and Deep Neural Networks for Parallelization. In Leonel Sousa, Nuno Roma, and Pedro Tomás, editors, *Euro-Par 2021: Parallel Processing*, volume 12820, pages 201–216. Springer International Publishing, Cham, 2021. ISBN 978-3-030-85664-9 978-3-030-85665-6. doi: 10.1007/978-3-030-85665-6_13. 13

[22] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and Scalable Parallelization for ML Computation Graphs. December 2021. doi: 10.48550/arXiv.2105.04663. 12

[23] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. June 2022. doi: 10.48550/arXiv.2201.12023. 12

[24] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-Efficient Neural Architect. June 2018. doi: 10.48550/arXiv.1806.07912. 12

Appendix

Alternate Baseline Model Experiments

As noted in Section 3.1 there is a minimum accuracy that our transformations should exceed in order to be useful. We defined this as a model with half the amount of layers. However we found that our simplified problem of 100 classes instead of 1000 significantly changed the characteristics of the model. Figure 5.1 shows that the shallow models, that is models with fewer layers, performed better than the deep models.

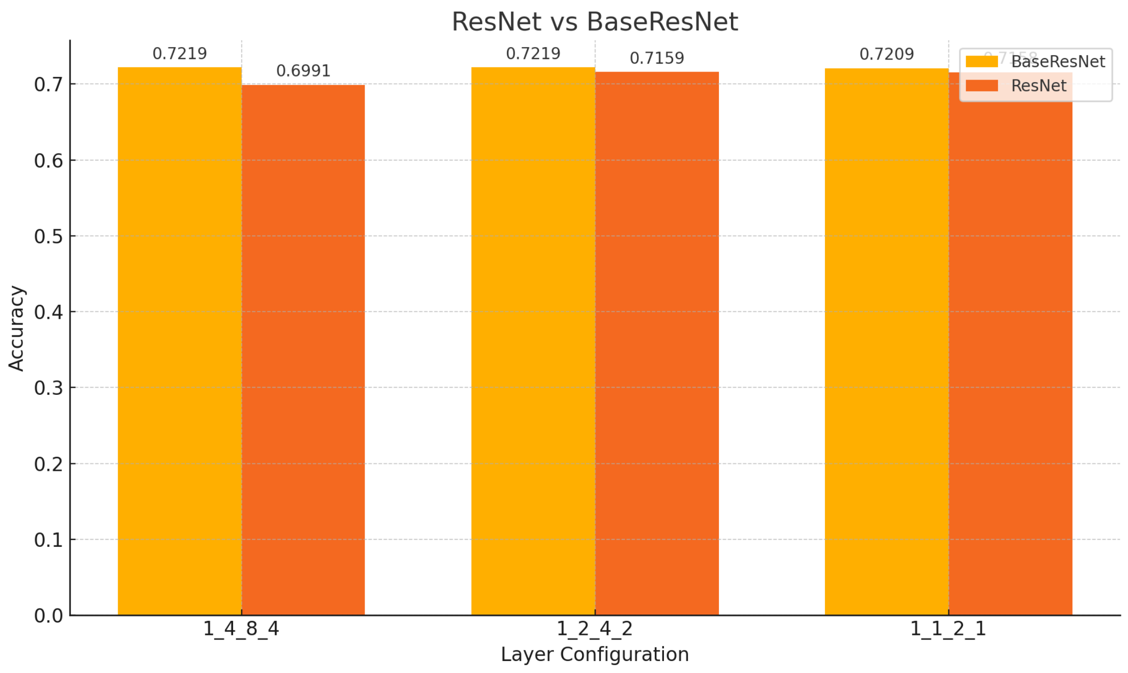
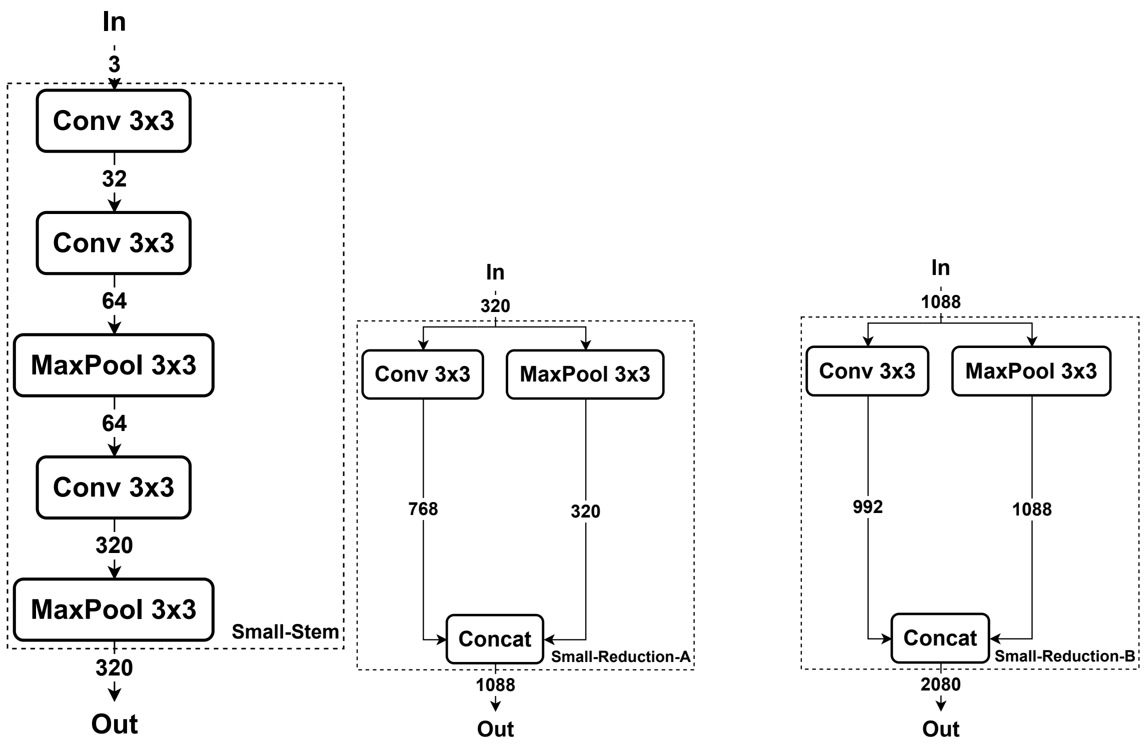


Figure 5.1: Performance of the Inception-ResNet-v2 models at various depths

While part of this behavior might be to our training setup, we opt to introduce changes into the model in order to establish a baseline, rather than reengineer the training setup.

In order to establish a baseline model we have made a number of adjustments that greatly simplify the Stem (Figure 5.2a), Reduction-A (Figure 5.2b), and Reduction-B (Figure 5.2c) cells. These new cells still transform the input and output tensors to the same shape, reducing the spatial dimensions and increasing the channel dimension, but feature significantly fewer operators.



(a) Small Stem **(b)** Small Reduction A **(c)** Small Reduction B

Figure 5.2: New Small Reduction Modules

The maxpool operator is retained as in our testing removing this would cause the training loss to not converge.

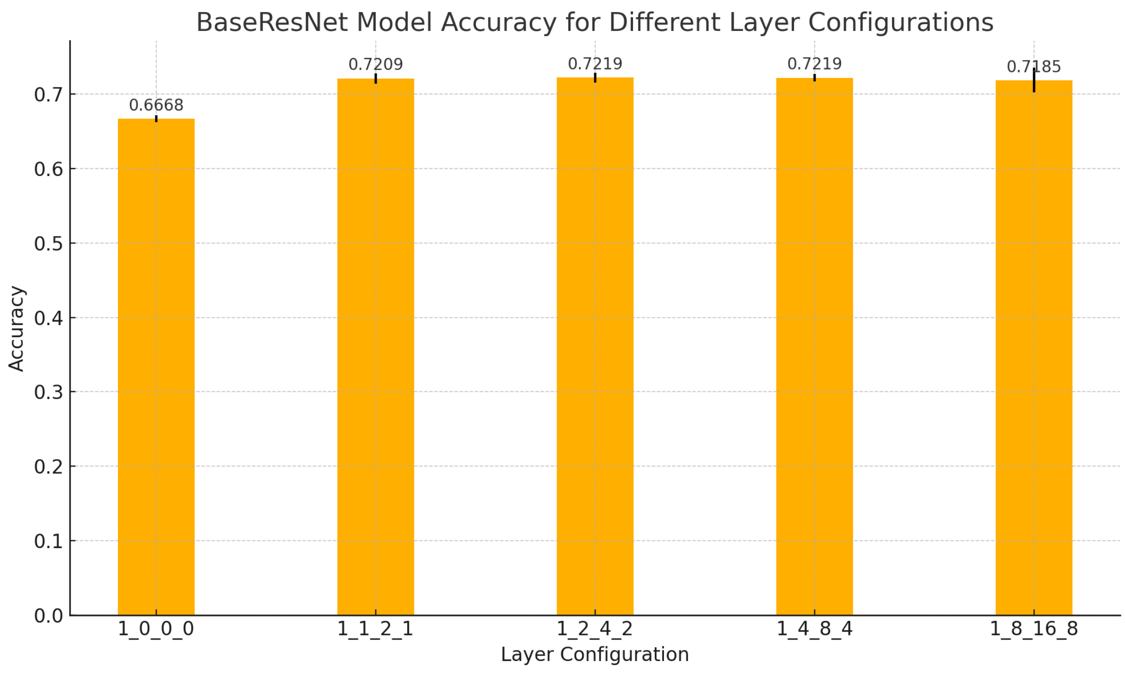


Figure 5.3: Performance of the Baseline models