Theofanis Petkos

Final Master's Project

Optimizing Neural Network Deployment Through Partitioning

Directed by Dr. Marc Sanchez

School of Engineering

Master's Degree in Security Engineering and Aritifical Intelligence

UNIVERSITAT ROVIRA i VIRGILI

Taragona, 2025

Contents

1 Introduction **3**

1.1 The Growth of Distributed AI 3

1.2 Challenges in Deploying DNNs Across Distributed Systems 3

1.3 Our Cost-Based Approach to Model Partitioning 4

2 State of the Art **4**

2.1 Strategies for Neural Network Partitioning in Distributed Inference 4

2.2 Performance Prediction Tools for Deep Learning Deployment 5

2.3 Our Contribution to the State of the Art 6

2.4 Comparison of Related Work 7

3 Background **8**

3.1 The ResNet Convolutional Networks 8

3.2 ResNet Variants 8

3.3 ResNet Architecture 8

3.4 The ResNet-18 Example 9

3.5 Convolutions in ResNet 11

3.6 Basic and Residual Blocks in ResNet 12

3.7 TorchVision 12

3.8 Partitioning Strategies 14

3.9 Definition of a Model Part 14

4 Methodology **17**

4.1 The Optimization Problem 17

4.2 Practical Optimization Approach. 18

4.3 Partitioning vs Splitting Algorithms 21

4.4 Balanced Number Partitioning Algorithm 21

4.5 Exhaustive Search Algorithm 23

5 Results **25**

5.1 Methodology 25

5.2 Pipeline Execution Performance Metric 25

5.3 ResNet-18 Results 27

5.4 ResNet-34 Results 27

5.5 ResNet-50 Results . . . 27
5.6 ResNet-101 Results . . . 29
5.7 ResNet-152 Results . . . 29
5.8 Impact of Group Execution Time on Partitioning Performance . . . 30
5.9 Impact of Group Transfer Size on Partitioning Performance . . . 30
5.10 Impact of the Convolution Score on Partitioning Performance . . . 31

6 Discussion and Future Work **32**

7 Conclusion **35**

1 Introduction

The recent growth of deep neural networks (DNNs) in both complexity and computational resources has created a fair amount of challenges for deployments in resource-limited environments.

Modern deep-learning models, specifically the ones used in computer vision and natural language processing, often contain billions of parameters, making their execution on devices impractical due to memory and/or other computational constraints (Parthasarathy et al., [1]).

1.1 The Growth of Distributed AI

Historically, the concept of distributed neural network computations began from a parallel computing approach in the 90s, where researchers initially researched data parallelism to accelerate training on systems with multiple CPUs and GPUs (Mayer et al., [11]). Early work in distributed machine learning mainly focused on splitting datasets across multiple nodes, rather than breaking the models themselves into parts (Verbraeken et al., [12]). Nowadays however, splitting up a model across devices (model-partitioning) for running AI tasks became much more common; thanks to the growth of edge computing and the increase of IoT devices during the 2010s (Zhou et al., [13]).

The move from centralized cloud computing to a more distributed edge-based approach happened when researchers realized that relying only on cloud servers causes problems like latency, high network bandwidth use, and potential privacy risks (Teerapittayanon et al., [14]). Early distributed deep learning systems, like the ones proposed by Teerapittayanon et al. [14], brought the idea of Distributed Deep Neural Networks (DDNNs), which split inference tasks across end devices, edge nodes, and the cloud. Their approach showed that early layers of a neural network could run close to the data source. For example, on a phone allowing quick, local predictions, while still uses the cloud for more complex processing when needed.

1.2 Challenges in Deploying DNNs Across Distributed Systems

The move towards to a more efficient AI model partioning across multiple devices has been taken forward by better edge computing architectures, as well as, the growth of systems that use different types of hardware together (Xu et al., [8]). Looking carefully at recent surveys, combining edge computing with deep learning has brought new ideas for distributed inference, where parts of a model can run on devices with different capabilities. Early methods usually split models in two - between an edge device and the cloud. However, having in mind the increasing compliexity of setups, is clear that we need more optimized tools that can handle a variety of devices with different speed, memory, and power limits (Mayer et al., [11]).

Additionally, the need for automated model partitioning comes from real-world deployment use cases where relying only on centralized processing is not enough. For example, in autonomous vehicle systems the vision models must analyze sensor data instantly and coordinate via vehicle-to-vehicle networks. That is something edge computing makes possible by pre-processing at roadside units and reducing delays (Ku et al., [15]). Healthcare wearables also highlight something very specific. The distributed processing on-device saves power and keeps personal data private instead of sending everything to the cloud (Tho et al., [16]).

1.3 Our Cost-Based Approach to Model Partitioning

In regards to our research, the suggested approach tackles a key problem in neural network deployment: *how to split a sequential model into groups in the best possible way.* More specifically, our goal is to balance three different objectives at the same time:

- Keep the execution time of each group low.
- Minimize group data transfer (reducing communication overhead between groups).
- Maximize the number of convolutions per group (increase computational density of each group)

Unlike earlier methods that focus on just one objective or even are based on trial-and-error strategies (Xu et al., [8]), our approach uses a specific algorithm based on a new cost function that plays an important role in the partitioning process. That said, in this master thesis, we aim to address the question:

- *How can we automatically partition sequential model across multiple homogenous devices, in a way that optimizes execution time, communication overhead, and computational efficiency?*

To answer this, we propose a cost-based partitioning framework that evaluates potential groupings of model parts using a cost function. Based on this cost, we designed and implemented two algorithms:

- An exhaustive search method.
- A balanced numbers partioning method.

Both algorithms help us to create efficient partitions that can be deployed in distributed environments. Finally, in our approach, we define a model part as a subset of layers within a neural network that functions as a modular block. Each part has exactly one input and one output, while containing operations like convolutions, activations, and residual connections.

2 State of the Art

As DNNs continue to grow in complexity, a fair amount of research has been done to address the challenges of executing them across distributed systems.

2.1 Strategies for Neural Network Partitioning in Distributed Inference

Modern artificial neural networks (ANNs) have become so large and complex that running them on a single resource-limited device is often not practical. Model partitioning has therefore become an important strategy to speed up inference by splitting a network across multiple devices (edge nodes, mobile devices, and cloud servers) and running each part where it works best.

As mentioned in the introduction, some early studies explored this idea. For example, Teerapittayanon et al.[14] introduced Distributed Deep Neural Networks (DDNNs), which split inference across end devices, edge servers, and the cloud. In their design, the first layers run close to the data source (e.g., on a phone) to deliver fast local predictions, while the remaining layers are offloaded to more powerful edge or cloud nodes for deeper processing. Around the same period, Kang et al.[17] proposed Neurosurgeon, a runtime system that automatically decides at which layer to split DNN execution between a mobile device and the cloud, aiming to minimize latency or energy use. This approach reduced end-to-end inference latency for mobile applications by up to three times.

These cases confirmed that partitioning a model between local and remote execution can reduce network communication overhead and response times, without sacrificing the accuracy of large models. Further research has made model partitioning more generic, allowing us to have more flexible deployments across distributed systems. The early implementations typically used a single split (edge vs. cloud), often chosen manually or through simple thresholds. More recent approaches support multi-tier and multi-device partitioning to make a better use of available hardware resources.

For example, distributed inference pipelines have been demonstrated in clusters of IoT or edge devices. Stahl et al. [18] (2021) presented a method to partition a convolutional neural network (CNN) across numerous microcontroller units, with each device running a different layer and passing intermediate results to the next. This parallelism at the edge can improve throughput and reduce latency by running concurrently, while always keeping an eye on the limited memory and the inter-device communication.

Researchers have also investigated adaptive strategies, like inserting early-exit branches into neural networks, to allow partial inference results to be used on the edge if a confidence threshold is met (Li et al. [19]). This type of dynamic partitioning ensures that each input uses only the necessary computation, reducing latency for simpler cases. With the wide variety of hardware available - from cloud GPUs to IoT accelerators - there is growing interest in automated partitioning frameworks that use performance modeling to determine optimal splits. To assist this, tools for execution time estimation are required.

Two modern examples of such tools are ANNETTE and MONNET. Wess et al. [21] (2021) developed ANNETTE (*Accurate Neural Network Execution Time Estimation*), a framework that learns to predict inference latency of a DNN on specific hardware platforms. Complementing ANNETTE, Osterwind et al. [22] (2022) introduced MONNET (*Model of Neural Network Execution Time*), an analytical performance library that estimates execution time for each network layer on a target processor.

The above research is widely cited and creates the basis for subsequent developments, as these works have been reviewed thoroughly and they are referenced oftenly. This makes them reliable and important sources for our research.

2.2 Performance Prediction Tools for Deep Learning Deployment

As ANNETTE and MONNET provided an important basis for our research, it is worth explaining these frameworks in more detail. After all, accurate performance modeling of deep neural network (DNN) inference is important for optimizing deployments and making good partitioning decisions.

ANNETTE (Wess et al., 2021) is a framework that predicts DNN inference latency on hardware

accelerators using a stacked modeling approach (Wess et al., [21]). It builds separate models for each type of layer and for inter-layer effects such as compiler layer fusion, then combines them to estimate the total runtime. ANNETTE collects performance data from small kernel tests and multi-layer benchmarks, capturing hardware behaviors that can be missed from simpler analytic models. This results in high accuracy: for example, on a Xilinx ZCU102 FPGA SoC and an Intel Neural Compute Stick 2 (NCS2), predictions were within about 3.5% and 7.4% of actual runtime across 12 different neural networks.

MONNET (Osterwind et al., 2022) is a Python library that scopes on predicting per-layer execution time on embedded accelerators (Osterwind et al., [22]). It works by running many small test networks on the target device to learn how parameters like input size and filter count affect latency. This process can take days for a new device but produces a very detailed performance model.

With tools like ANNETTE and MONNET, it is possible to estimate the time needed to run layers 1 through k on one device and the remaining layers on another, adding the communication cost between them. By comparing these estimations for different split points, the system can choose the partitioning that minimizes inference time or meets any given latency target. These tools, published between 2020 and 2022, represent state-of-the-art methods for accurate performance modeling in DNN deployment.

2.3 Our Contribution to the State of the Art

The approach proposed in this thesis builds on the above state-of-the-art basis and tries to address their limitations but also combinine their strengths. Prior performance modeling tools like ANNETTE and MONNET mainly focus on predicting latency, and prior partitioning methods typically optimize a single objective (most often end-to-end delay or, in some cases, energy). For instance, Kang et al. [17] (2017) optimized either latency or energy in their cloud-edge splitter, but not both at the same time.

Many DNN partitioning methods focus on just one goal. A recent survey found that most aim to improve either inference speed or throughput, while ignoring other important factors (Xu et al., [8]). Some researchers have started to address multiple objectives, but often in a limited way. For example, Jeong et al. [24] (2018) consider both computation time and the time to send intermediate data in their IoT partitioning method, and also Teerapittayanon et al. [14] (2017) propose a distributed DNN framework that balances accuracy with energy use.

Previous multi-objective methods often treat each metric separately and lack a single way to handle trade-offs. Many do not use a single cost function; instead, they apply step-by-step rules (for example, offloading a layer if its delay is too high) without a formal multi-metric optimization framework. Our approach instead uses one cost function that combines all performance factors into a single objective.

More specifically, we merge three metrics: latency, communication cost, and compute density. Latency is the time needed to run a partition of the network on certain hardware. Communication cost is the delay in sending intermediate data between partitions. Compute density measures how much computation is done relative to the size of the data, showing how efficiently a layer runs on a device. This combined cost helps decide whether it's better to run more work on a slower device or send extra data over the network.

We then solve this multi-objective problem with an optimization algorithm (see Chapter 4) that always considers speed, bandwidth overhead, and hardware efficiency together. This makes our

method different from much of the existing work, moving from single-goal tuning to a unified, multi-goal cost model.

2.4 Comparison of Related Work

To summarize the variety of contributions in model partitioning and distributed inference, Table 1 offers an overview and a way to compare most relevant papers discussed in this section. The comparison is based on six key criteria:

- **Multi-Device**: Whether the work supports deployment across more than two devices (e.g., edge, edge-cloud).
- **Latency Optimization**: Whether inference latency is explicitly minimized or used as a design objective.
- **Energy Aware**: Whether the approach includes energy usage or battery efficiency, particularly important for mobile or embedded systems.
- **Execution Prediction**: Whether the method includes or relies on predictive models for estimating execution time (e.g., ANNETTE, MONNET).
- **Communication Aware**: Whether the cost of transferring data (tensors) between devices is taken into account.
- **Multi-Objective**: Whether the optimization jointly considers more than one performance metric (e.g., latency, energy, communication).

Paper	**Multi-Dev.**	**Latency**	**Energy**	**Predict.**	**Comm.**	**Multi-Obj.**
Teerapittayanon (2017)	✓	✓	✓	✗	✓	✗
Kang (2017)	✗	✓	✓	✗	✗	✗
Stahl (2021)	✓	✓	✗	✗	✓	✗
Li (2018)	✓	✓	✗	✗	✗	✗
Wess (2021)	✗	✗	✗	✓	✗	✗
Osterwind (2022)	✗	✗	✗	✓	✗	✗
Jeong (2018)	✓	✓	✗	✗	✓	✗
Our Work	✓	✓	✗	✓	✓	✓

Table 1: Summary of key features supported by various neural network partitioning frameworks in the literature. The columns indicate whether each work supports multi-device deployment, latency optimization, energy optimization, performance prediction, communication cost reduction, and multi-objective optimization.

As shown in the table, most previous works focus on one or two main goals, usually reducing inference time or deciding when to offload between edge and cloud. Some, like Teerapittayanon et al [14], and Kang et al. [17], also consider energy use, especially for mobile devices. Only a few, such as Wess et al. (ANNETTE) [21] and Osterwind et al. (MONNET) [22], work on predicting execution time, and these usually do not handle partitioning directly.

Our approach builds on these ideas but aims to handle multi-device deployments from the start. It brings together latency, communication cost, and computational density into a single cost,

allowing proper multi-objective optimization for splitting models across devices. This makes it possible to go beyond simple two-way splits and supports more flexible distributed inference.

3 Background

3.1 The ResNet Convolutional Networks

Deep Residual Networks (ResNets), introduced by He et al. [27], are a family of convolutional neural networks (CNNs) designed to support the training of very deep architectures by introducing residual connections. These connections allow gradients to skip one or more layers during back-propagation, mitigating the vanishing gradient problem common in deep networks. The design is based on the observation that learning residual functions with reference to the layer inputs is easier than learning unreferenced mappings (He et al., [27]).

Each ResNet architecture contains a sequence of convolutional blocks, which is formed in groups with each block containing multiple layers, such as convolutions, batch normalizations, and activation functions. A standard ResNet includes an initial convolution and max pooling layer, followed by four main stages (groups of residual blocks), and ends with global average pooling and a fully connected layer.

Note that convolutions, basic and residual blocks will be described more in detail in sub-sections 3.5 and 3.6 respectively.

3.2 ResNet Variants

The most commonly used ResNet models are:

- ResNet-18
- ResNet-34
- ResNet-50
- ResNet-101
- ResNet-152

The difference between the variants can be found in their depth and the type of residual blocks they are using. ResNet-18 and ResNet-34 use *basic blocks*, which have two 3x3 convolutions, while ResNet-50 and bigger variants are using *bottleneck blocks*, which include a 1x1 convolution to reduce dimensionality, followed by a 3x3 convolution, and another 1x1 convolution to expand dimensionality (Wightman et al., [28]). This structure makes bigger models like ResNet-50, ResNet-101, and ResNet-152 more efficient in terms of computational resources than the ones using only basic blocks.

3.3 ResNet Architecture

The overall architecture of a ResNet model follows a high-level pattern, which applies to all variants. Each model begins with a initial block having a 7×7 convolutional layer with stride 2,

followed by batch normalization, a ReLU activation, and a 3×3 max pooling layer with stride 2. This initial block tries to reduce the resolution of the input while expanding its channel depth, preparing it to process more in depth.

After the initial block, the network is organized into four sequential stages (*layer1* to *layer4*), each containing a series of residual blocks. The first stage maintains the spatial resolution of the output received from the initial block, however every next stage reduces it in half, while it increases the number of channels. After *layer 4* we have a global average pooling layer, and finally a fully connected layer that returns the output.

The main difference between smaller ResNet models (e.g. ResNet-18 and ResNet-34) and its larger variants (like ResNet-50, ResNet-101, and ResNet-152) is the type of residual block they are using. Smaller models use the *basic block*, which has two consecutive 3×3 convolutional layers followed by batch normalization and a ReLU activation. On the other hand, the larger variants use the *bottleneck block*, which has an 1×1 convolution to reduce dimensionality, along with a 3×3 convolution for spatial processing, and another 1×1 convolution to restore dimensionality.

Regarding the *bottleneck block*, it is designed to make the network faster and lighter while still keeping its ability to learn useful features. This design decreases the number of floating-point operations (FLOPs) compared to stacking only 3×3 convolutions, which supports better the training of deeper networks like ResNet-101 and ResNet-152 (He et al., [27]).

3.4 The ResNet-18 Example

To support the previous sub-section for the ResNet architecture, here we will elaborate on the ResNet-18 architecture. This variant has 18 layers that can be trained, and they are all made up of basic blocks. As Figure 1 shows, the ResNet-18 structure is:

- **Initial Block (Stem):** 7×7 convolution (64 channels, stride 2), batch normalization, ReLU, 3×3 max pooling (stride 2).
- **Layer 1:** 2 basic blocks, each with two 3×3 convolutions (64 channels) and then a ReLU.
- **Layer 2:** 2 basic blocks (128 channels), first block down-samples using stride 2 in the first convolution and a ReLU.
- **Layer 3:** 2 basic blocks (256 channels), first block down-samples via stride 2 and then a ReLU.
- **Layer 4:** 2 basic blocks (512 channels), first block down-samples via stride 2 and then a ReLU.
- **Final Block:** Global average pooling, fully connected layer to the number of output classes (e.g., 1000 for ImageNet).

Skip connections in all blocks either directly connect the input to the output or use a 1×1 convolution when there is a mismatch in dimensions due to down-sampling. This architecture offers a good balance between depth and speed, and this is the main reason is one of the most popular choices when execution time is important.

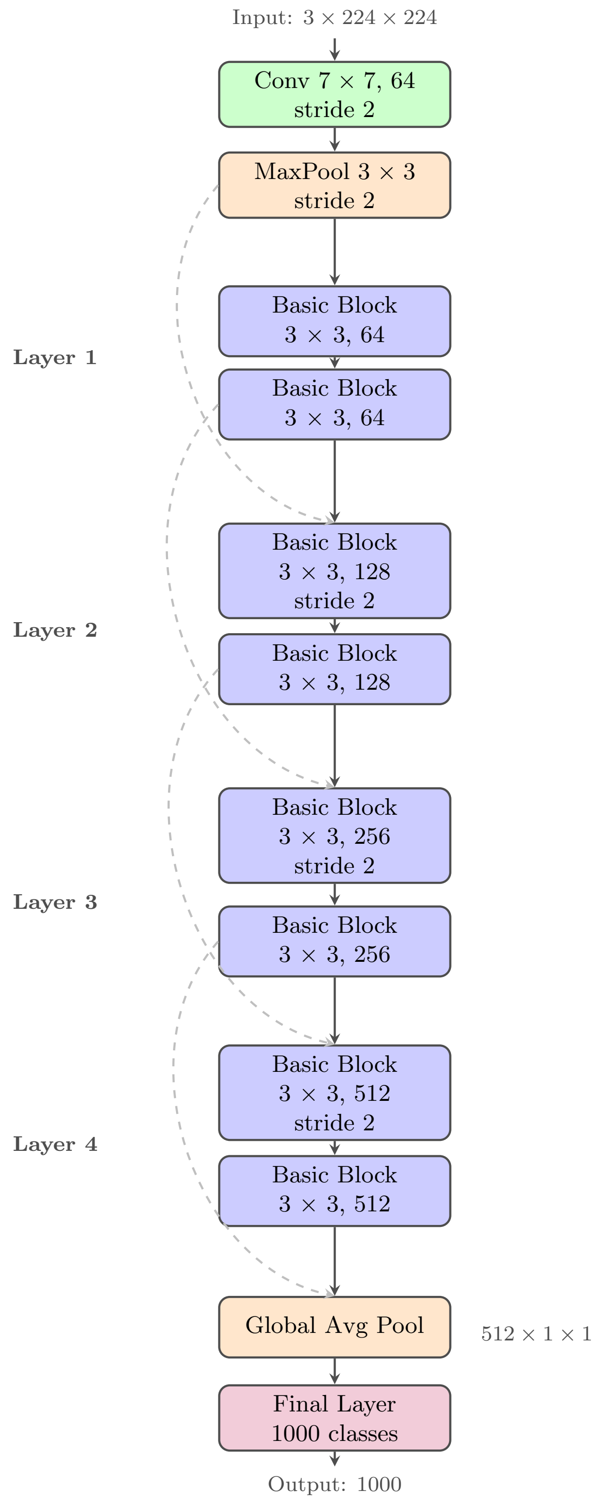

Figure 1: The ResNet-18 model takes the input, passes it through an initial convolution layer, then through four stages of basic blocks (Layers 1 - 4), and ends with the classification layers. Each basic block has two 3×3 convolutions with batch normalization and ReLU activation. The dashed lines show the skip connections.

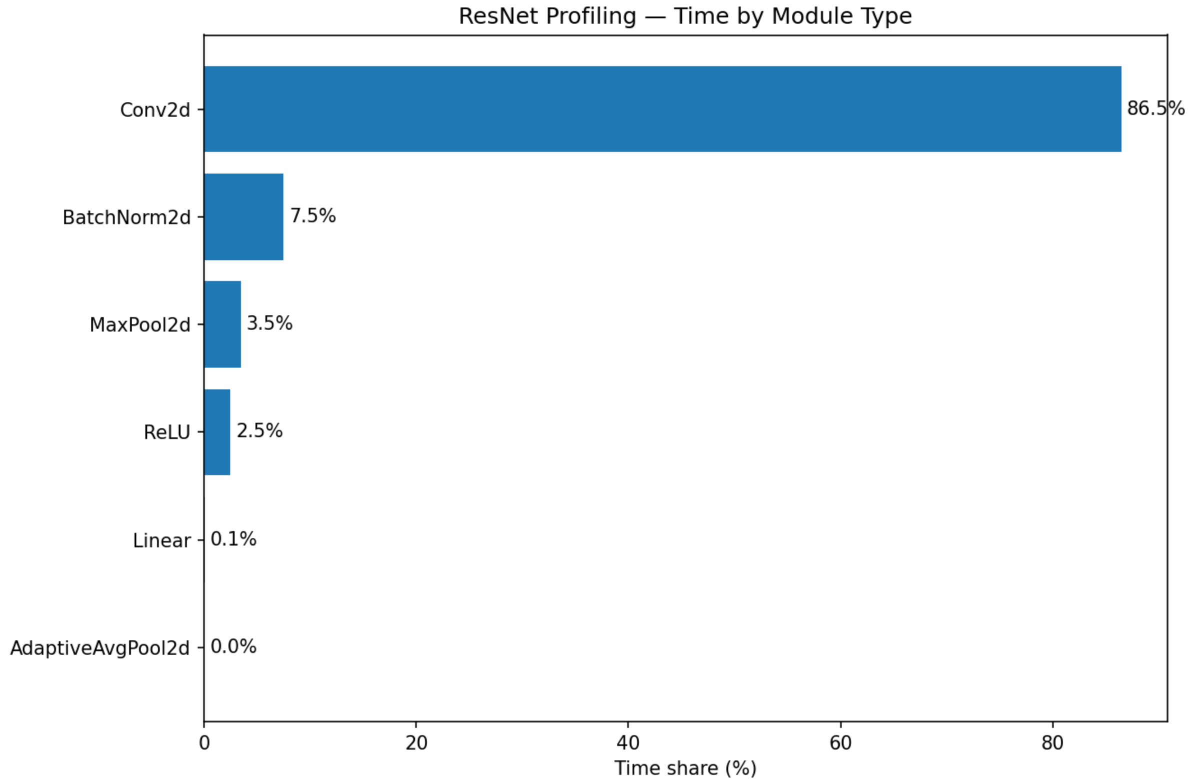

Figure 2: Time share by module type (ResNet). Convolutional layers dominate the runtime, taking over 85% of total execution time, whereas BatchNorm, pooling, ReLU, and the classifier each contribute only a few percent.

3.5 Convolutions in ResNet

Convolution is a mathematical operation that combines two functions to produce a third one, describing how the shape of one modifies the other in time or space. It serves as a tool for filtering, and system response characterization (Ji et al., [20]).

In convolutional neural networks (CNNs), convolution layers apply learned filters (kernels) on parts of the input - for example image patches or time-series segments - to extract local features. These filters slide over the input, performing dot products that produce feature maps, which are then passed through non-linear activations (Zhao, [26]). This approach adds support for parameter sharing and sparse connectivity, significantly reducing the number of weights compared to fully connected layers.

Importance of Convolutions in ResNets. To understand where execution time is spent inside a ResNet, we profiled the network by module type and aggregated the share of total runtime per layer family. Figure 2 shows that convolutional layers account for the large majority of time, while other modules (BatchNorm, pooling, activations, and the final linear layer) contribute only a small fraction. This supports using a convolution-aware term in our background cost model.

3.6 Basic and Residual Blocks in ResNet

ResNet architectures are built from repeated building units known as *basic blocks* and *residual blocks* (He et all., [27]). These structures are important to understand how all ResNet variants achieve their depth without suffering from the vanishing gradient problem.

The **Basic Block** is the simplest unit of ResNet and consists of:

- two stacked 3×3 convolution layers,
- each followed by Batch Normalization and a ReLU activation,
- and an identity shortcut that directly adds the block's input to its output.

Formally, if the input is x, the block computes:

$$y = F(x, W) + x$$

where $F(x, W)$ represents the two-layer transformation parameterized by weights W. This shortcut ensures that the gradient can flow directly through the block, making training much more stable.

On the other hand, the **Residual Block** is a more generic basic block. It uses the same idea of a shortcut connection but allows for:

- identity mappings (when input and output dimensions match), or
- projection shortcuts (a 1×1 convolution on the shortcut path) when dimensions differ between input and output.

Residual blocks are a core feature of ResNets. They let the network learn residual functions, such as $F(x) = H(x) - x$, where $H(x)$ is the desired mapping. This formulation allows much deeper networks to be trained without degradation.

As also mentioned above:

- **Basic block:** a simple two-layer residual block used in shallower ResNets (e.g., ResNet-18, ResNet-34).
- **Residual block:** the general design principle with shortcut connections, extended in deeper variants (e.g., ResNet-50 and beyond).

3.7 TorchVision

TorchVision is a library in the PyTorch ecosystem that provides standardized implementations of popular image models, pretrained weights, and dataset loaders (TorchVision Documentation: https://docs.pytorch.org/vision/stable/index.html). It includes implementations of all major ResNet variants, enabling users to load them easily with 'torchvision.models.resnet18()' and similar calls. Through this library we are able to see the internals of each model via a '.layers' or '.children()' methods, which allows us to manipulate, partition, and of course benchmark

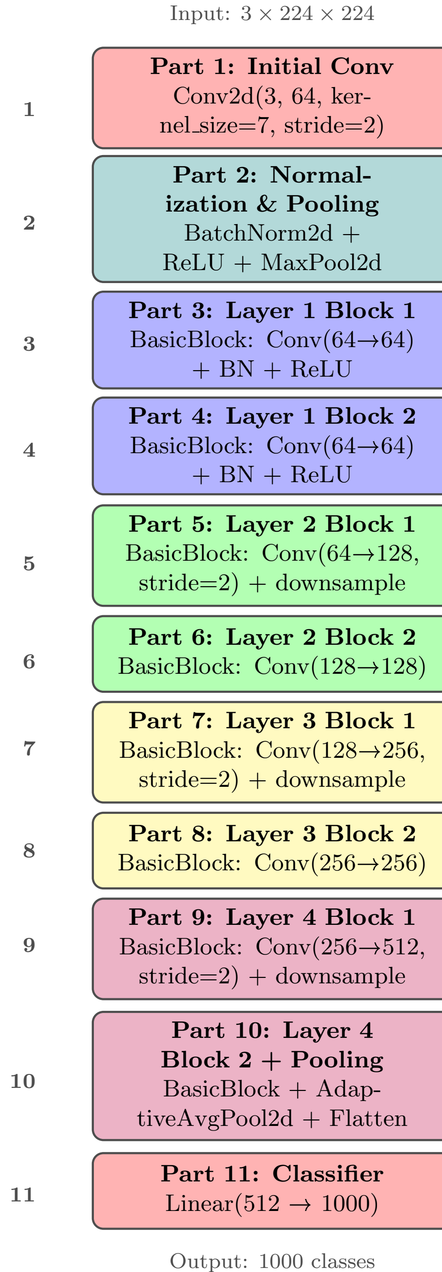

Figure 3: ResNet-18 architecture divided into 11 model part components for distributed processing. Each part represents a logical grouping of operations that can be processed independently. Parts 1-2 handle initial feature extraction, Parts 3-4 process Layer 1 blocks, Parts 5-6 handle Layer 2 blocks, Parts 7-8 manage Layer 3 blocks, Parts 9-10 process Layer 4 blocks, and Part 11 performs final classification.

model components. In our research, we used these TorchVision models, stripped them into specific parts, and then evaluated them for partitioning.

Figure 3 shows an example of how we split the ResNet-18 model from TorchVision into parts. We use this example in our research to explain and visualize how models can be divided for testing and benchmarking.

3.8 Partitioning Strategies

There are two major strategies for partitioning a neural network across multiple devices: *horizontal* and *vertical* partitioning (Kang et al., [17]).

- **Horizontal Partitioning** splits up each layer into smaller pieces either across its height, width, or channels. For example, one GPU might process the top half of the data, while another handles the bottom. This method allows for parallel processing, but it needs very precise coordination between devices and often results in high communication costs. It works best in environments where all devices are similar, like GPU clusters in data centers.
- **Vertical Partitioning** on the other hand, splits the model by dividing it into groups of layers. This approach works well in environments with different types of devices, like edge devices or mobile-cloud systems. Each group of layers is assigned to a different device, so they can run more independently without needing constant coordination.

3.9 Definition of a Model Part

Definition 3.1 (Model Part)**.** *A **model part** is a self-contained section of a neural network with exactly one input tensor and one output tensor. Each part groups together a sequence of operations, such as convolutional layers, activation functions, batch normalization, and residual connections, that are treated as a single unit during execution.*

To keep each part independent, we follow these rules:

- **Single input and output:** Each part takes one input tensor and produces one output tensor.
- **Internal dependencies:** All operations and skip connections inside the part are handled internally, so other parts do not need to manage them.
- **Consistent tensor formats:** The input and output tensors match what neighboring parts expect, allowing smooth connections between parts.

Example of Internal Dependency Resolution. Consider the **Part 5** in Figure 3, which corresponds to *Layer 2, Block 1* of ResNet-18. This block contains:

- A 3×3 convolution with stride 2,
- A batch normalization layer,
- A ReLU activation, and

- A downsampling skip connection.

The skip connection produces an extra tensor, but it is combined inside the part by adding it to the main convolution output. After this step, the part sends a single output tensor to the next part.

From our framework's perspective, the whole block behaves as a **single, self-contained unit**:

- It takes one input tensor,
- Handles all computations and merges internally,
- Produces one clean output tensor for the next part.

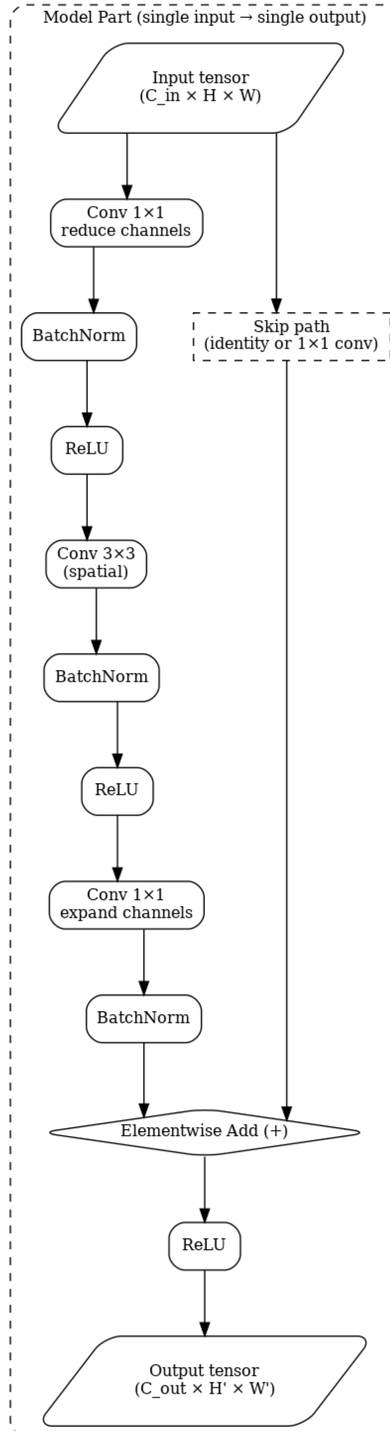

Figure 4: A model part with a single input and single output. Internally it contains a bottleneck residual block: Conv1×1 - Batch Norm - ReLU, Conv3×3 - Batch Norm - ReLU, Conv1×1 - Batch Norm, plus a skip path (identity or 1×1 downsample). The skip path and main path are merged using element-wise addition before applying ReLU activation. All intermediate operations are resolved inside the part, so the partitioning algorithm only sees one input and one output.

This design makes the partitioning process simpler because each part works independently. That said, the partitioning algorithm does not need to track internal skip connections or intermediate results.

4 Methodology

To automatically split a model into k sequential groups, we use a cost function that rates each potential group of model parts. The cost score balances three goals: keeping execution time low, reducing the cost of sending tensors between devices, and having an efficient number of convolutions in each group. By balancing between these factors, the cost helps us compare different groupings and guides the partitioning toward the best mix of speed, computation, and communication.

4.1 The Optimization Problem

Let a neural network be composed of n sequential model parts $P = \{p_1, p_2, \ldots, p_n\}$. The objective of partitioning is to divide these parts into k contiguous groups $G = \{G_1, G_2, \ldots, G_k\}$, where each group $G_i \subset P$ is a subsequence of model parts executed as a single unit on a device.

Performance Goal. Our goal should be to minimize the pipeline execution time when processing n concurrent requests through the distributed system:

Definition 4.1 (Optimization Objective)**.**

$$\textit{Objective:} \min_{G:|G|=k} Perf(G, n)$$

where $Perf(G, n)$ is the pipeline performance for handling n requests in parallel using the k-group split G. The goal is to minimize the total inference time in a distributed system.

The optimization is subject to the following constraints:

- The grouping G preserves the execution order of parts.
- Each model part belongs to exactly one group.
- Group formation defines tensor transfer boundaries.
- Pipeline execution follows dependency and resource constraints.

In computational complexity theory, problems are classified according to the resources required to solve them. NP-hard problems represent a class of computational problems that are at least as difficult as the hardest problems in the complexity class NP (non-deterministic polynomial time) (Duchnowski et al., [3]). Formally, a computational problem X is NP-hard if, for every problem P which can be solved in non-determinitstic polynomial-time there is a polynomial reduction from P to X, meaning that if we could solve P in 1 unit time we could use P's solution to solve also X (Garey et al., [2]). NP-hard problems represent the boundary basis in computational complexity. As Fortnow explains (Fortnow et al., [4]), the P versus NP problem is *"one of the most important questions in all of mathematics"* with profound implications for algorithm design. Most computer scientists believe that P $\neq$ NP, which suggests that NP-hard problems cannot be solved efficiently. If a problem is proven to be NP-hard, it means

it's usually better to use approximation or heuristic methods instead of trying to find exact solutions, especially for large cases (Fortnow et al., [4]).

Finding the best way to split model parts into k groups to minimize pipeline performance $P(G, n)$ is an NP-hard problem. This can be shown by relating it to several other well-known NP-hard problems.

- *Quadratic Assignment Problem (QAP)* [5]: QAP involves assigning facilities to locations while minimizing a cost function that depends quadratically on both the flows between facilities and the distances between locations. Our partitioning problem is similar to the Quadratic Assignment Problem (QAP) because pipeline performance depends on complex interactions between group execution times, transfer delays, and resource sharing. The final performance is influenced by how parts are assigned to groups and how their execution times relate to each other.
- *Resource-Constrained Project Scheduling (RCPSP)* [6]: RCPSP is about scheduling tasks that must follow a certain order while sharing limited resources, with the goal of finishing as quickly as possible. Our problem is similar: model parts must be run in a set sequence, groups decide how resources are used, and we aim to minimize total time to process n requests while following pipeline order and resource limits.
- *Multiprocessor Scheduling Problem* [2]: This classic NP-hard problem assigns jobs with set processing times to multiple processors to finish all work as quickly as possible. Our pipeline partitioning is similar but more complex: we must also respect the order of model parts, handle dependencies between groups, and account for communication costs between them.

Since our pipeline performance optimization problem is closely related to well-known NP-hard problems and adds extra challenges (like sequential dependencies, communication costs, and handling multiple requests at the same time), we can conclude it is also NP-hard. Simulating pipeline execution requires making complex scheduling decisions with tightly linked timing constraints. This makes finding an exact optimal solution impractical for large cases, which is why efficient heuristic methods are needed for real-world use.

4.2 Practical Optimization Approach.

From the above, directly optimizing $Perf(G, n)$ is computationally expensive and this problem belongs to the class of NP-Hard problems. Furthermore, it requires running simulations for the complete pipeline execution for every possible grouping.

That said, instead of optimizing $Perf(G, n)$ directly, our partitioning algorithm uses a computationally efficient alternative: the cost function. At each step, the algorithm used for model partitioning will compare the cost of splitting a group versus keeping it together, making local decisions that should lead to good overall pipeline performance.

Cost Function. Each model part p_j is associated with three scalar metrics:

- t_j: execution time
- s_j: data transfer size at the output

- c_j: number of convolutional operations

For each group G_i, we define aggregated metrics:

$$T_i = \sum_{p_j \in G_i} t_j, \quad C_i = \sum_{p_j \in G_i} c_j, \quad S_i = s_{\text{last}(G_i)}$$

where $s_{\text{last}(G_i)}$ is the transfer size of the last part in the group, approximating the cost of communication to the next group.

Let the total sums be:

$$T_{\text{total}} = \sum_{j=1}^{n} t_j, \quad C_{\text{total}} = \sum_{j=1}^{n} c_j, \quad S_{\text{total}} = \sum_{j=1}^{n} s_j$$

We define normalized group metrics:

$$T_i^{\text{norm}} = \frac{T_i}{T_{\text{total}}}, \quad S_i^{\text{norm}} = \frac{S_i}{S_{\text{total}}}$$

Convolution Count Score. To ensure the number of convolutions per group is important for the partitioning, we apply a bending downwards scoring function to the raw count ($\log(1 + C_i)$). This allow us to ensure that groups with more consecutive convolution layers return a better normalized score than splitting them. Such grouping aligns with compiler-level fusion optimizations (for example, Conv + BatchNorm + ReLU fused into a single kernel), which reduces memory traffic and kernel launch overhead and speeds up inference. This follows what Niu et al. (2021) [29] described in their work for accelerating DNN execution with operator fusion.

We define a normalized convolution score for each group G_i as:

Definition 4.2 (Normalized Convolution Score)**.**

$$\mathit{ConvScore}(G_i) = \frac{\log(1 + C_i)}{\log(1 + C_{total})}$$

where C_i is the number of convolution layers in group G_i, and C_{total} is the total number of convolutions in the model. This score grows with the number of convolutions, but sublinearly due to the logarithm, which prevents overly favoring large groups.

Group Cost Function. The cost of group G_i is computed using a weighted combination:

Definition 4.3 (Group Cost Function)**.**

$$\mathit{Cost}(G_i) = \alpha \cdot T_i^{norm} + \beta \cdot S_i^{norm} - \gamma \cdot \mathit{ConvScore}(G_i)$$

with weight constraints:

$$\alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \in [0, 1]$$

Local Optimization Strategy. As we have described above, the global objective (Definition 4.1) is too expensive to compute. Instead, we apply a local partitioning strategy, which will be explained in detail in Section 4.4. We begin with the whole model in one group and gradually split it into k groups.

For each candidate group G_i, we test whether splitting it into two subgroups $\text{Cost}(G_i^{\text{Left}})$ and $\text{Cost}(G_i^{\text{Right}})$ reduces the total pipeline cost. The reason we use cost reduction is because our main objective is to minimize the total execution cost of the full pipeline. By comparing the cost of one large group to the cost of its two subgroups, we can check if the split moves us closer to that goal.

$$\Delta\text{Cost}(G_i) = (\text{Cost}(G_i^{\text{Left}}) + \text{Cost}(G_i^{\text{Right}})) - \text{Cost}(G_i)$$

Here:

- $\text{Cost}(G_i^{\text{Left}})$ and $\text{Cost}(G_i^{\text{Right}})$ are the two new groups created from splitting G.
- $\Delta\text{Cost}(G_i)$ measures how much the suggested splitting improves cost.
- A negative value means the split reduces the pipeline cost.

However, cost reduction alone is not enough. If one subgroup becomes much slower than the others, the whole system will be limited by that bottleneck. This is why we make use of the $BalanceFactor(G)$ function we have defined in Definition 4.5.

This penalty is added because an unbalanced split may reduce cost locally but in the long run affect the overall performance. The system runs at the pace of the slowest group, so balance must always be considered.

The final score for the candidate split of G_i is then:

Definition 4.4 (Final Cost Reduction)**.**

$$\text{FinalCostReduction}(G_i, G) = \Delta\, Cost(G_i) + \text{BalanceFactor}(G)$$

- $\Delta\text{Cost}(G_i)$ measures the local benefit of splitting G_i.
- $\text{BalanceFactor}(G)$ measures the negative impact of creating unbalanced group execution times across the whole partitioning, as stated below at Definition 4.5.

A split is accepted if and only if $\text{FinalCostReduction}(G_i, G) < 0$. This ensures that we only apply splits that both improve local cost and maintain balanced workloads across the entire pipeline.

Balance Factor. To prevent bottlenecks where one group takes much longer than others, we add a balance penalty (calculated using the formula in Definition 4.5) that grows when groups have uneven execution times. Let k denote the total number of groups in the current partitioning:

Definition 4.5 (Balance Factor)**.**

$$BalanceFactor(G) = \delta \cdot \left(\frac{\max(\{T_1, T_2, \ldots, T_k\})}{\text{mean}(\{T_1, T_2, \ldots, T_k\})} - 1 \right)$$

where $\delta \in [0, 1]$ is a adjustable weight controlling the strength of the penalty. This penalty encourages balanced workload distribution across groups and is subtracted from the cost reduction when evaluating potential splits.

Design Reasoning. The cost function is designed as a heuristic that should lead to good pipeline performance. It balances three competing factors:

- Minimizing execution time (T_i^{norm}) to reduce computational overhead.
- Minimizing transfer size (S_i^{norm}) to reduce communication costs.
- Maximizing the convolution score ($ConvScore(G_i)$) to reward groups that contain more consecutive convolution layers. This encourages groupings where compiler-level fusion is more effective, improving computational density and reducing memory transfers.

Fine-Tuning. By adjusting α, β, γ, and δ, one can configure the cost function for specific deployment constraints. For example, settings with limited bandwidth may assign higher β, while environments with many GPUs might emphasize compute density with higher γ, and systems with strict latency requirements may increase δ to enforce balanced group execution times. This flexibility makes the function adaptable to a wide range of distributed inference scenarios.

4.3 Partitioning vs Splitting Algorithms

Now that we have formally defined the optimization problem (Definition 4.1), the key question turns into: *how should we group the p model parts into k partitions to minimize inference cost?*

- The merging strategy starts with each part in its own group. In each iteration, it merges the pair of groups that results in the smallest cost increase (or largest reduction). This bottom-up approach focuses on local decisions that improve cost at each merge step.
- On the other hand, the *top-down approach* begins with the entire model treated as a single group. It then splits recursively this group by checking all possible split points and selecting the one that reduces most the cost. This approach is close to decision-tree recursive partitioning, and was inspired in part by works like MONNET (Osterwind et al, [22]), which use per-layer cost prediction to guide division.

During our research both methods aim to optimize a multi-objective cost function that considers (i) execution time, (ii) communication overhead, and (iii) computational density. As explained their main difference is in the order of decisions: one builds up from minimal groups, while the other breaks down from a monolithic whole.

4.4 Balanced Number Partitioning Algorithm

The Balanced Number Partitioning Algorithm is the main algorithm we use in our experiments. Algorithm 1 shows a specific way to group model parts and tries to balance the work across groups while keeping costs low.

Balanced number partitioning tries to split model parts into k groups so that each group takes about the same time to run. This idea comes from classic computer science problems like the multi-way number partitioning problem, which is known to be NP-hard (Garey et al, [2]). For model parallelism, balanced number partitioning helps spread the work evenly across different devices, which reduces bottlenecks and makes the pipeline run faster.

Unlike exhaustive search that checks every possible way to split the parts, this algorithm uses a greedy approach. It starts with all parts in one big group, then keeps splitting groups to lower

costs while keeping the workload balanced. This method is much faster than exhaustive search when we want both good performance and balanced work distribution.

Algorithm 1 Balanced Number Partitioning via Cost-Balanced Splitting

Require: $P = \{p_1, p_2, \ldots, p_n\}$, k, M_{total}, α, β, γ
Ensure: Partition $G = \{G_1, G_2, \ldots, G_k\}$

```
 1: function BalancedNumberPartitioning(P, k, M_total, α, β, γ)
 2:     groups ← {P}                                   ▷ Start with all parts in one group
 3:     while |groups| < k do
 4:         best_reduction ← −∞, best_group_idx ← −1, best_split_pos ← −1
 5:         for group_idx ← 0 to |groups| − 1 do
 6:             group ← groups[group_idx]
 7:             current_cost ← Cost(group, α, β, γ, M_total)
 8:             for split_pos ← 1 to |group| − 1 do
 9:                 left_group ← group[0 : split_pos]
10:                 right_group ← group[split_pos : |group|]
11:                 left_cost ← Cost(left_group, α, β, γ, M_total)
12:                 right_cost ← Cost(right_group, α, β, γ, M_total)
13:                 base_cost_reduction ← current_cost − (left_cost + right_cost)
14:                 new_groups ← groups[0 : group_idx] ∪ {left_group, right_group} ∪
    groups[group_idx + 1 : |groups|]
15:                 exec_times ← [∑_{p∈g} t_p for g ∈ new_groups]
16:                 avg_time ← ∑(exec_times) / |new_groups|
17:                 balance_factor ← max(exec_times) / avg_time
18:                 cost_reduction ← base_cost_reduction − 0.5 × (balance_factor − 1)
19:                 if cost_reduction > best_reduction then
20:                     best_reduction ← cost_reduction
21:                     best_group_idx ← group_idx
22:                     best_split_pos ← split_pos
23:                 end if
24:             end for
25:         end for
26:         if best_group_idx ≠ −1 then
27:             group_to_split ← groups[best_group_idx]
28:             left_group ← group_to_split[0 : best_split_pos]
29:             right_group ← group_to_split[best_split_pos : |group_to_split|]
30:             groups ← groups[0 : best_group_idx] ∪ {left_group, right_group} ∪
    groups[best_group_idx + 1 : |groups|]
31:         end if
32:     end while
33:     return groups
34: end function
```

This splitting method works differently from exhaustive search. Instead of trying every possible combination of splits, we start with the whole set of model parts as one group and split it step by step into smaller groups. At each step, it picks the split that gives the biggest cost reduction (line 13) while keeping groups balanced (line 17). The algorithm uses the same cost function as stated in Definition 4.3.

This continues until we have k groups or no more good splits can be found. By always picking the split that saves the most cost while staying balanced, the algorithm finds groupings that work well for pipeline performance with even workload distribution.

Compared to exhaustive search, which checks all possible ways to split the parts, this method is much faster because it makes decisions one step at a time based on the current state. This makes it better for cases where the model parts have very different costs or when we need fast results without checking every possibility.

Definition of Cost. The function $\text{Cost}(G_i, \alpha, \beta, \gamma, M_{total})$ calculates the cost of a group G_i following the definition 4.3:

Algorithm 2 Cost Calculation

Require: G_i (candidate group), M_{total} (metrics of entire model), α, β, γ
Ensure: $\text{Cost}(G_i)$

1: $norm_exec_time \leftarrow \frac{\sum_{p \in G_i} t_p}{M^{\text{exec}}_{\text{total}}}$
2: $norm_transfer_size \leftarrow \frac{s_{G_i}}{M^{\text{transfer}}_{\text{total}}}$
3: $group_convs \leftarrow \sum_{p \in G_i} c_p$
4: $total_convs \leftarrow C_{\text{total}}$
5: $log_conv_score \leftarrow \frac{\log(1 + group_convs)}{\log(1 + total_convs)}$
6: **return** $\alpha \cdot norm_exec_time \; + \; \beta \cdot norm_transfer_size \; - \; \gamma \cdot log_conv_score$

Balance Factor. To make sure the groups stay balanced, the algorithm adds a penalty when splits create uneven execution times. The balance factor (Definition 4.5) measures how unbalanced the groups are by comparing the slowest group to the average group time. When groups have similar execution times, the balance factor is close to 1.0 and no penalty is added. But when one group takes much longer than others, the balance factor grows larger and a penalty is applied. This penalty reduces the effect of splits that create bottlenecks. Without this penalty, the algorithm might create one very slow group that would hold up the entire pipeline, since all requests must pass through every group in order. By penalizing unbalanced splits, we ensure that the workload stays distributed across groups, which keeps the pipeline running smoothly.

4.5 Exhaustive Search Algorithm

The Exhaustive Search Algorithm is the algorithm we use to compare the results of the Balanced Numbers Partioning Algorithm 1. Algorithm 3 shows a method that checks every possible way to split the model parts to find the best answer.

Exhaustive search tries every way we can split the model parts into k groups. Since model parts must stay in the same order, this means trying all possible places where we can put dividers between groups. For n parts split into k groups, we need to pick $k-1$ divider spots out of $n-1$ possible spots. This gives us $\binom{n-1}{k-1}$ different ways to check.

The big advantage of exhaustive search is that it always finds the best possible answer for our cost function. Since it checks every option, we know the result is the best one possible. But the number of ways to split grows very fast as we add more parts, making this approach slow for big models.

Algorithm 3 Exhaustive Search for Best Model Splitting

Require: Model parts $P = \{p_1, p_2, \ldots, p_n\}$, number of groups k, total metrics M_{total}, weights α, β, γ
Ensure: Partition $G = \{G_1, G_2, \ldots, G_k\}$

```
 1: function ExhaustiveSearch(P, k, M_total, α, β, γ)
 2:     best_cost ← ∞
 3:     best_grouping ← None
 4:     combinations ← all ways to pick k − 1 spots from {1, 2, . . . , n − 1}
 5:     for each combination in combinations do
 6:         groups ← {}
 7:         start ← 0
 8:         for each divider in combination do
 9:             groups ← groups ∪ {P[start : divider]}
10:             start ← divider
11:         end for
12:         groups ← groups ∪ {P[start : n]}
13:         total_cost ← 0
14:         for each group in groups do
15:             group_cost ← Cost(group, α, β, γ, M_total)
16:             total_cost ← total_cost + group_cost
17:         end for
18:         balance_penalty ← GetBalancePenalty(groups)
19:         total_cost ← total_cost + 0.5 × balance_penalty
20:         if total_cost < best_cost then
21:             best_cost ← total_cost
22:             best_grouping ← groups
23:         end if
24:     end for
25:     return best_grouping
26: end function
```

The algorithm works by trying every possible way to split the model parts. For each way to split, it makes the groups by putting dividers at the chosen spots. Then it finds the total cost by adding up the cost of each group using the same cost function as defined in 4.3.

Balance Factor. Just like the balanced number partitioning algorithm, exhaustive search also adds a penalty to avoid making uneven groups. The balance factor (Definition 4.5) looks at how different the execution times are across groups. This penalty gets bigger when groups have very different execution times, making the algorithm prefer more balanced splits. Without this penalty, the algorithm might find ways to split that have low costs but create slow groups that hold up the whole pipeline.

The algorithm keeps track of the best split found so far and updates it whenever it finds a better one. After checking all possible ways to split, it returns the one with the lowest total cost.

5 Results

5.1 Methodology

To test how well our model partitioning algorithms work, we ran experiments on five ResNet versions: *ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152.*

For each model, we used a batch of 16 images from the *ImageNet* dataset as input for runtime evaluation.

Our goal was to measure the balance between optimal pipeline execution performance and the total execution time for two different partitioning strategies:

- *Balanced Numbers Partitioning.*
- *Exhaustive Search.*

Each algorithm's goal is to divide a given ResNet model into k sequential groups, with k ranging from 1 to 8.

For each k and algorithm:

- Execution time was measured as the duration required to compute the k-group partition.
- Partition quality was measured using the pipeline execution performance metric $P(G, n)$. This metric simulates running n requests at the same time through the k-group pipeline, taking into account both computation time and communication costs.

All layers in each model were split into single model parts, each being a sequence of operations with one input and one output (as defined in the Background section). These same parts were used for both algorithms.

We repeated the evaluation for each ResNet variant and showed the performance values and execution costs for each k value in grouped bar charts.

All experiments were run in a controlled Python environment in CPU-only mode (no GPU) to keep timing results consistent.

5.2 Pipeline Execution Performance Metric

To better measure the success of each partioning algorithm, we introduced the pipeline execution performance function $Perf(G, n)$ that attempts to simulate the concurrent execution of n requests through a k-group pipeline, accounting for both computational and communication costs. This function applies the realistic scenario where multiple requests are processed in parallel through a distributed inference pipeline.

Group Metrics Definition. For a grouping $G = \{G_1, G_2, \ldots, G_k\}$, we define:

- Group execution time: $T_i = \sum_{p_j \in G_i} t_j$
- Group transfer time: $D_i = \frac{s_{\text{last}(G_i)}}{B}$

where $s_{\text{last}(G_i)}$ is the output tensor size of the last part in group G_i, and B represents the bandwidth between devices (set to 25×1024 bytes/ms in the implementation).

Sequential Case. For the simplest case where $k = 1$ (no partitioning), the performance is:

$$Perf(G, n) = n \cdot T_1$$

Concurrent Pipeline Execution. For $k > 1$, we model the pipeline execution as follows. Let A_i^r denote the availability time of group G_i after processing request r, and let S_i^r be the start time of request r on group G_i.

The pipeline execution performance formula is based on the following principles:

1. **Group Availability:** A group cannot start processing a new request until it finishes the current one:
$$S_i^r \geq A_i^{r-1}$$
for all groups i and requests r.
2. **Pipeline Dependency:** A request cannot start on group G_i until it completes processing on the previous group G_{i-1}:
$$S_i^r \geq A_{i-1}^r$$
for all groups $i > 1$ and requests r.
3. **Initial Condition:** The first group can start processing the first request immediately:
$$S_1^1 = 0$$

The start time for request r on group G_i is determined by:

$$S_i^r = \begin{cases} A_i^{r-1} & \text{if } i = 1 \\ \max(A_{i-1}^r, A_i^{r-1}) & \text{if } i > 1 \end{cases}$$

The availability time after processing request r on group G_i is:

$$A_i^r = S_i^r + T_i + D_i$$

Performance Metric. The total pipeline execution time is the completion time of the last request through the final group:

$$Perf(G, n) = A_k^n = S_k^n + T_k + D_k$$

This performance metric captures the key aspects of distributed inference:

- **Pipeline Parallelism:** Multiple groups can process different requests simultaneously.
- **Resource Limitations:** Each group can only process one request at a time.
- **Communication Overhead:** Transfer delays between groups affect overall throughput.

This metric gives a clear picture of how different ways of splitting the model affect the total inference time in a distributed setup, taking into account both the computing time and the communication limits.

Default coefficients. Unless otherwise stated, we fixed the weight values used in all experiments to $\alpha = 0.3$, $\beta = 0.4$, and $\gamma = 0.3$. These values sum to one and were kept constant across all ResNet variants and all values of k.

5.3 ResNet-18 Results

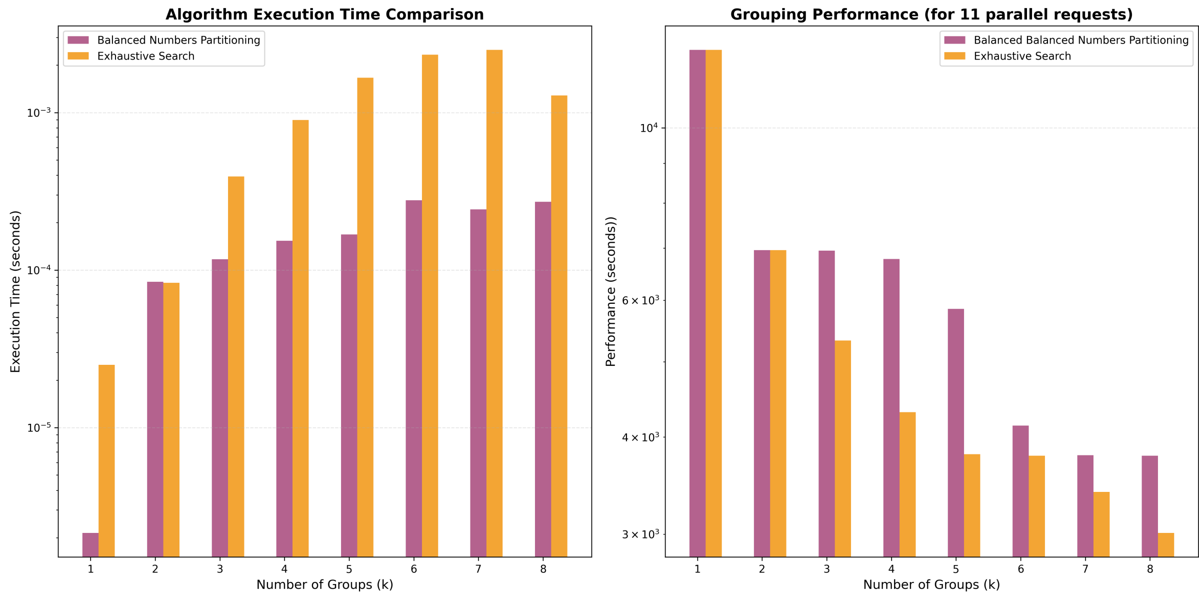

Figure 5: Execution time (left) and pipeline execution perfomance (right) for ResNet-18 across different group counts k.

The execution time plot (Figure 5 on the left) shows that Exhaustive Search becomes much slower as the number of groups k increases, while Balanced Numbers Partioning stays consistently faster across all values of k. Balanced Numbers Partioning keeps execution times well below 10^{-3} seconds, whereas Exhaustive Search takes almost ten times longer for higher k.

The grouping performance plot (Figure 5 on the right), measured for 11 parallel requests, shows that Exhaustive Search is slightly better for small group counts ($k = 2$ and $k = 3$). However, Balanced Numbers Partitioning stays close in performance while running much faster. Overall, it offers the best balance between speed and performance.

5.4 ResNet-34 Results

The execution time plot (Figure 6 on the left) shows that Exhaustive Search becomes much slower as the number of groups k increases, exceeding 10^{-1} seconds for larger k values. Balanced Numbers Partioning stays consistently faster, keeping execution times under 10^{-3} seconds across all k.

The grouping performance plot (Figure 6 on the right) shows that Exhaustive Search achieves the best performance for small group counts ($k = 2$ and $k = 3$), but Balanced Numbers Partitioning stays close while being far quicker to run. For higher k, it delivers strong performance with minimal runtime cost, making it a more practical choice.

5.5 ResNet-50 Results

For ResNet50, the execution time results (Figure 7 on the left) show a clear gap between the two methods. Exhaustive Search slows down sharply as k grows, passing 10^{-1} seconds at $k = 8$. Balanced Numbers Partitioning remains fast, as it stays under 10^{-3} seconds for all group counts.

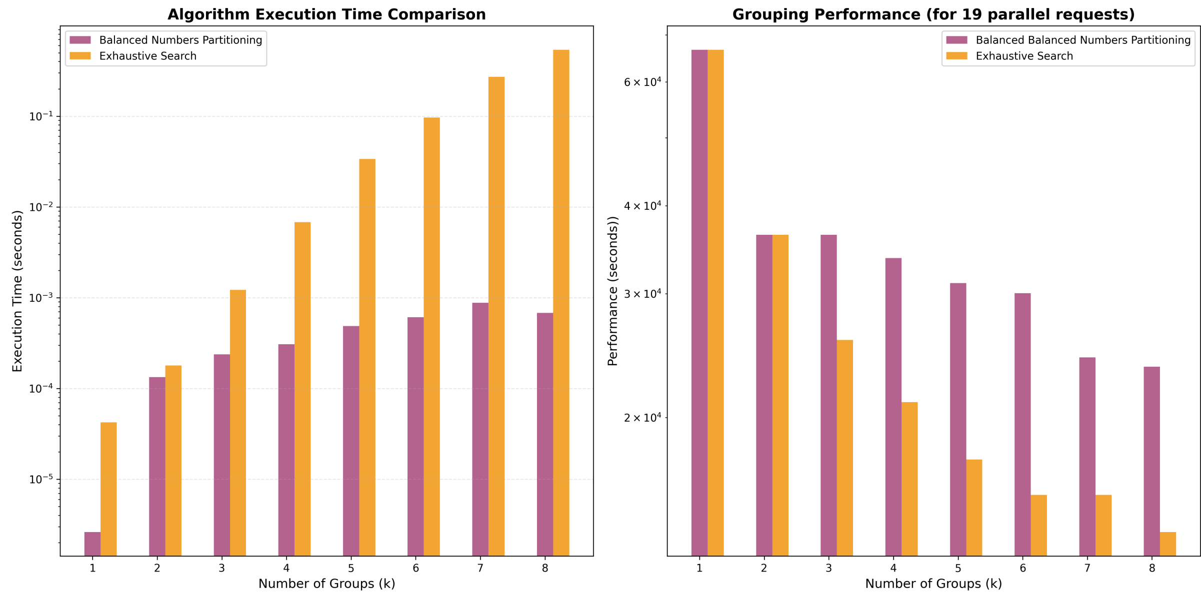

Figure 6: Execution time (left) and pipeline execution perfomance (right) for ResNet-34 across different group counts k.

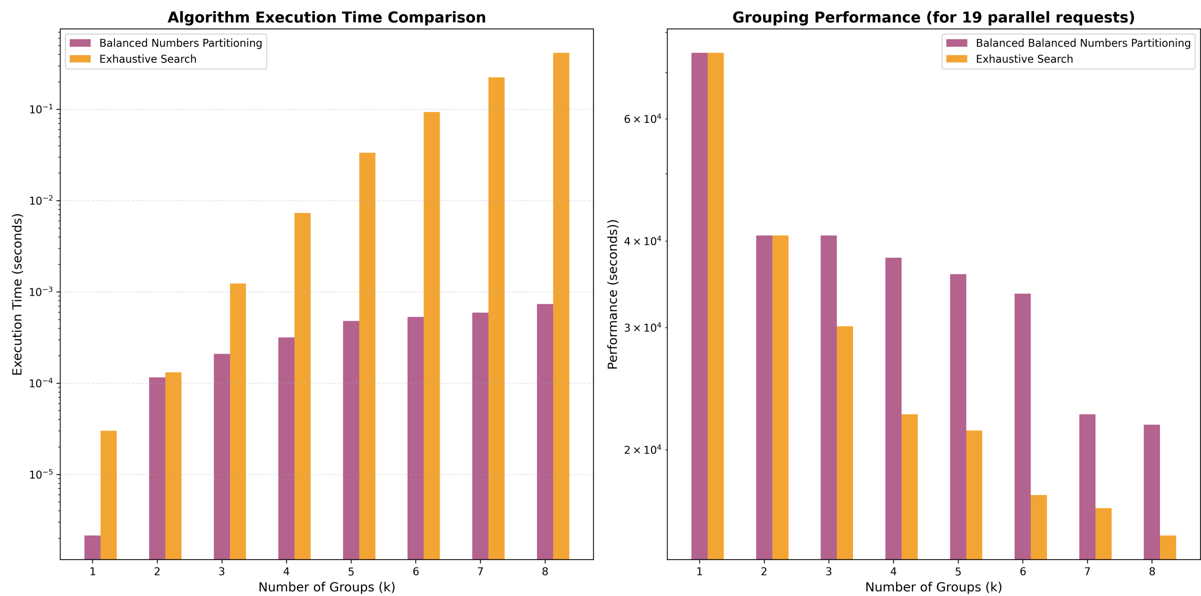

Figure 7: Execution time (left) and pipeline execution perfomance (right) for ResNet-50 across different group counts k.

Looking at performance (Figure 7 on the right), Exhaustive Search delivers the best results for smaller group counts, but the margin is small. Balanced Numbers Partitioning stays competitive across all k values and achieves this with a fraction of the runtime. This makes it a strong choice for cases where speed is critical and the absolute lowest cost is not the only priority.

5.6 ResNet-101 Results

ResNet101 was the main focus in our experiments, and all testing scripts were based on this model variant.

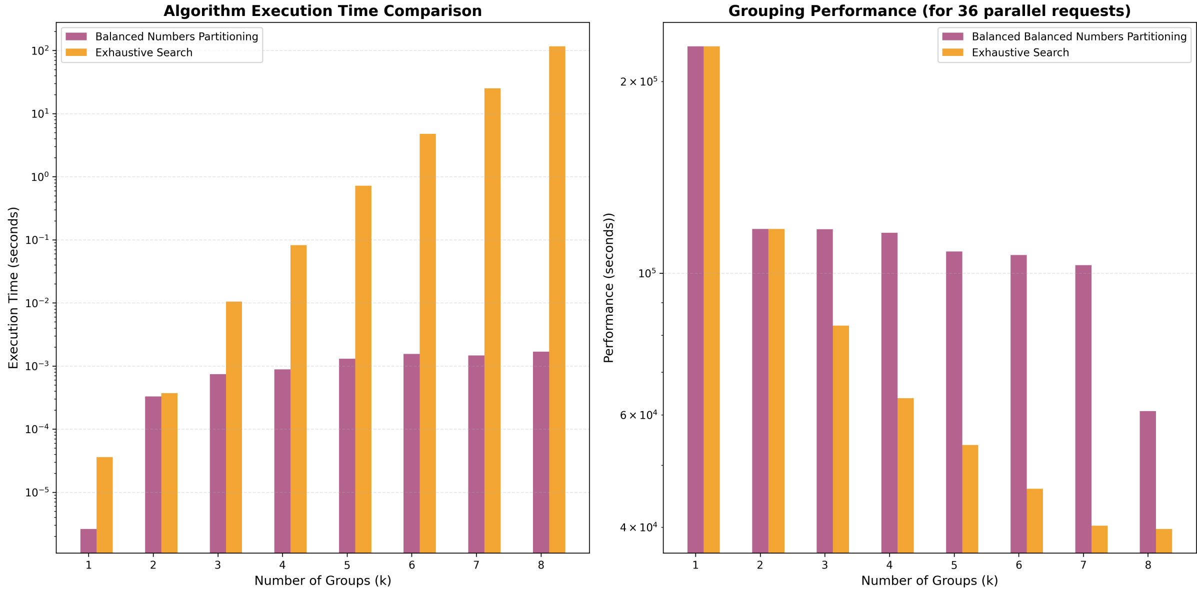

Figure 8: Execution time (left) and pipeline execution perfomance (right) for ResNet-101 across different group counts k.

From the execution time results (Figure 8 on the left), Exhaustive Search becomes much slower as k increases, with runtime growing exponentially. For k values above 8, running it locally is no longer practical. Balanced Numbers Partitioning, on the other hand, stays very fast and stable across all group counts.

In terms of performance (Figure 8 on the right), Exhaustive Search produces the lowest performance for most k values, but Balanced Numbers Partioning remains close to optimal while being far quicker to compute.

5.7 ResNet-152 Results

For ResNet-152, the execution time plot (Figure 9 on the left) shows that Exhaustive Search becomes extremely slow as k increases, surpassing 300 seconds at $k = 8$. Balanced Numbers Partitioning remains much faster, with execution times consistently below 10^{-3} seconds across all group counts.

Looking at the performance results (Figure 9 on the right), Exhaustive Search delivers the lowest cost for most k values, especially when $k \geq 3$. However, Balanced Numbers Partitioning

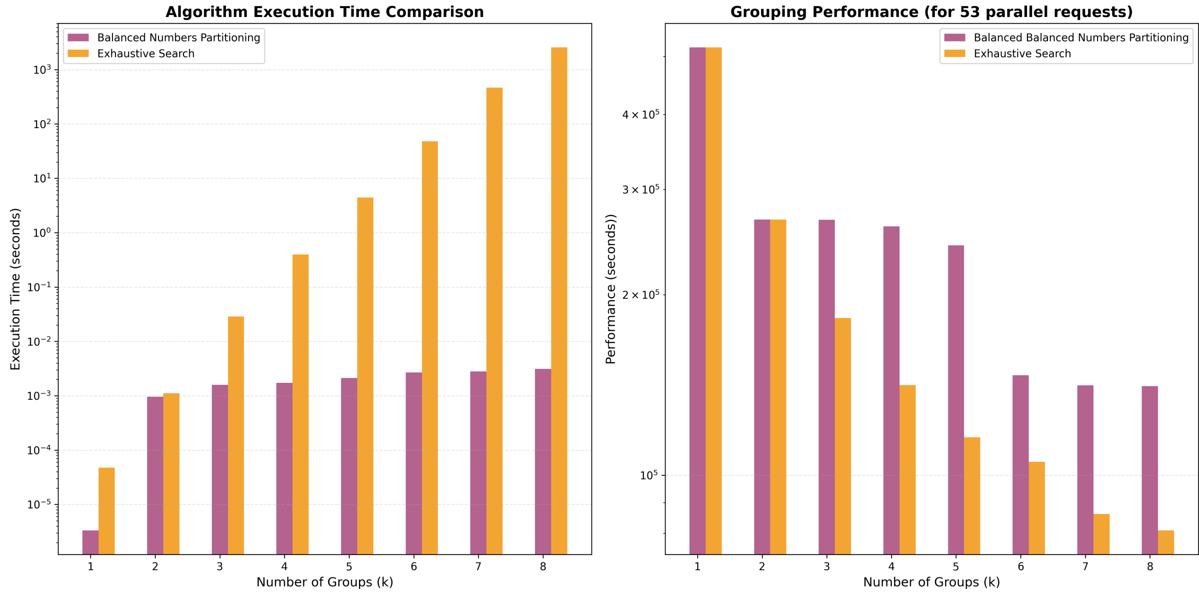

Figure 9: Execution time (left) and pipeline execution perfomance (right) for ResNet-152 across different group counts k.

stays relatively close to these results while being far more efficient to compute, making it more practical for large-scale scenarios.

5.8 Impact of Group Execution Time on Partitioning Performance

The α weights the execution-time term T_i of each potential group in the scoring function. We apply $\alpha \in [0, 1]$ on ResNet101 (other weights are adjusted so each time their sum is 1) and compute the pipeline performance $Perf(G, n)$ for the best partition returned by the search.

The performance is stable and low until $\alpha \leq 0.5$ (small variance around the baseline). A slight bump appears at $\alpha \approx 0.6$, followed by a mild improvement for $\alpha \in [0.7, 0.8]$. When $\alpha \geq 0.9$, performance degrades sharply, with a pronounced peak at $\alpha = 1.0$. This indicates that over-emphasizing compute time alone leads to unbalanced groups (compute-heavy stages dominate), hurting end-to-end latency.

5.9 Impact of Group Transfer Size on Partitioning Performance

The β weights the transfer-size term D_i, capturing communication between sequential groups. We apply $\beta \in [0, 1]$ on ResNet101 so we can evaluate $Perf(G, n)$.

Until $\beta \leq 0.5$, the performance remains close to the baseline. Starting around $\beta \approx 0.6$, performance worsens and stays high for $\beta \in [0.6, 0.9]$. At $\beta = 1.0$, the metric has a peak, indicating that focusing almost exclusively on minimizing transfer size returns partitions with poor compute balance (many small cuts that reduce communication but inflate stage runtimes).

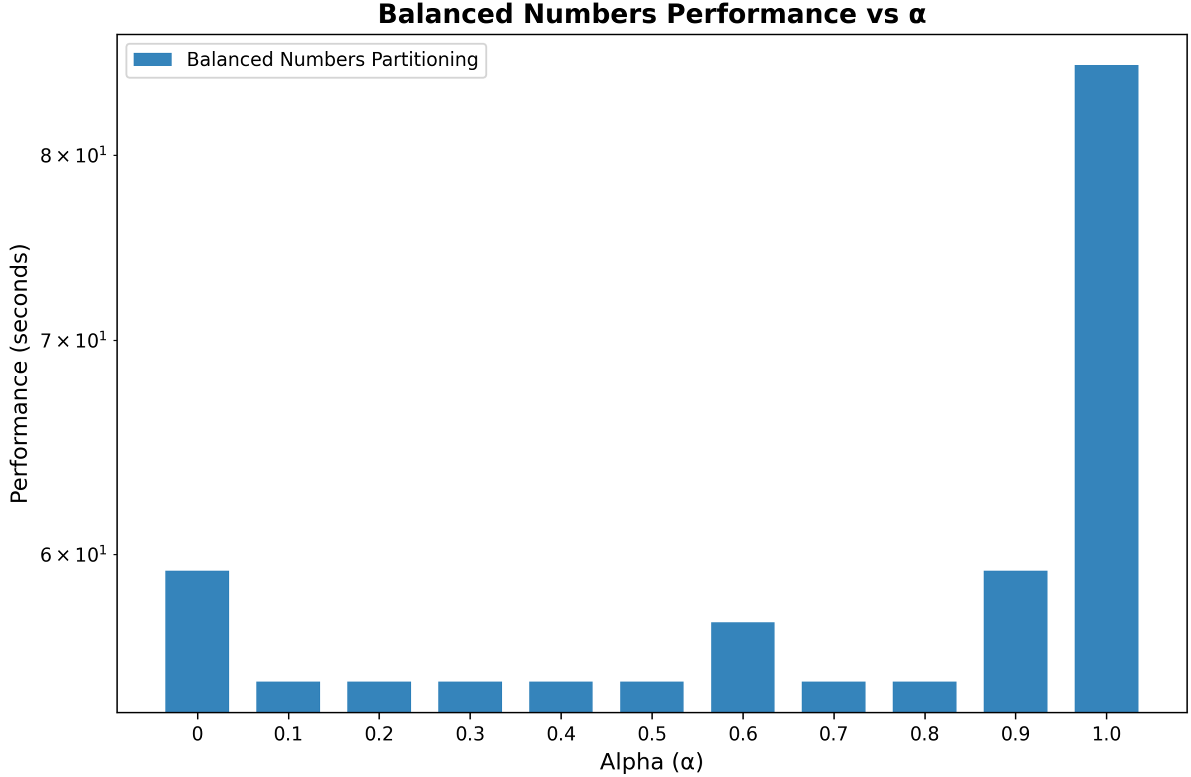

Figure 10: Evolution of pipeline performance as α varies (ResNet101). After a specific point (0.9) we can see that the overall performance is lower.

5.10 Impact of the Convolution Score on Partitioning Performance

Another goal for our research experiment, was justify the importance of the number of convolutions objective and why it is considered an equally important objective like the group execution time or the group transfer size.

Experiment Setup. The setup of the experiment was:

- We varied γ from 0 to 1, in steps of 0.1.
- For each value, we applied the balanced numbers partitioning algorithm on ResNet-152.
- The final metric for each different gamma γ value we have measured the pipeline performance.

As the Figure 12 shows us the worst scenario (e.g. the worst pipeline performance) is reported at $\gamma = 0$, proving that the absence of the convolution score has a negative impact on the final grouping's performance.

One the other hand we can note that the best range of γ is between 0.1 - 0.3 which reduce a lot the pipeline performance time, with the best performance at $\gamma = 0.3$.

Another important point is that beyond 0.3, the performance gradually worsens, which helps us justify that too high weight values on convolution score leads to suboptimal splits.

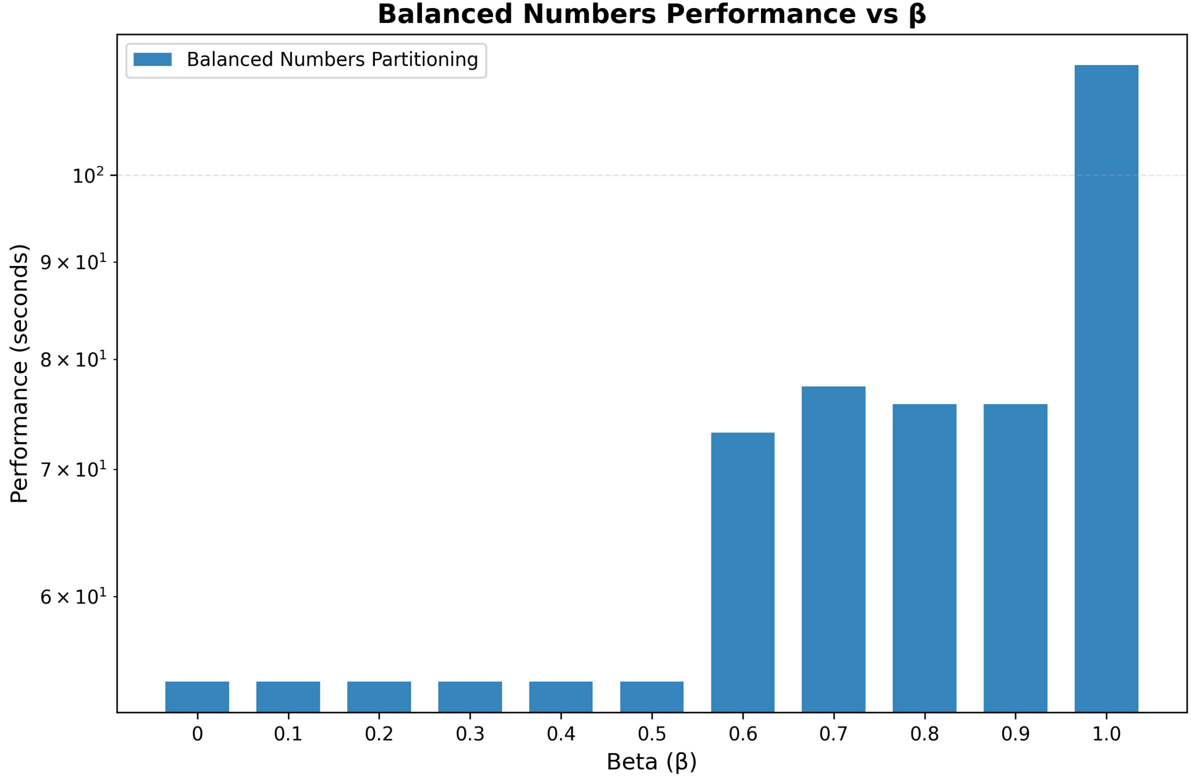

Figure 11: Evolution of pipeline performance as β varies (ResNet101). The performance looks stable until $\beta < 0.6$
, however after this point it gets worse.

Distribution of Convolutions per Group at Optimal γ. The histogram in Figure 13 shows how many convolution layers each group contains when using the optimal γ value 0.3. The k for this use case is 6 and the ResNet variation used is *ResNet-152*.

Key observations from Figure 13:

- We can see that the majority of groups are clustering around 25 convolutions per group.
- We can notice two more groups that contain a bigger number of convolutons (35 and 45).
- Finally there is an outlier group that has 55 convolutions.

Figure 13 proves that if we take into account the balance on computation intensity across the different groups the final result of the pipeline execution performance is better. The histogram shows us that the algorithm has ensured that the convolutions are spread accross groups.

6 Discussion and Future Work

Our experiments on different ResNet variants and architectures gave us important results for the strengths and trade-offs of our proposed cost formula and algorithms. More specifically, the *Balanced Numbers Paritioning Algorithm* 1 reported acceptable levels of performance just in a

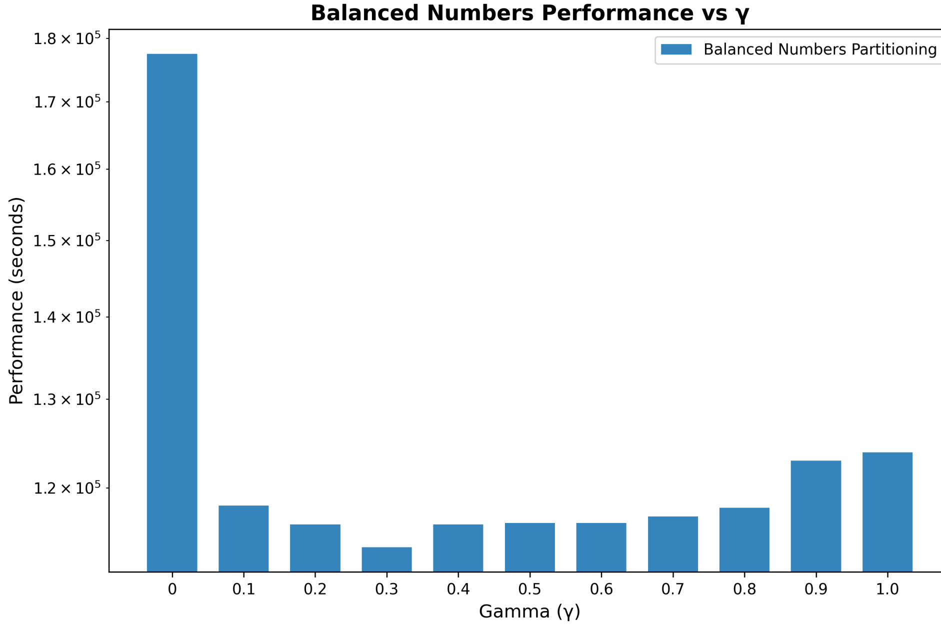

Figure 12: Partitioning Performance vs. the evolution of γ. The absence of γ weight adds a negative impact on the pipeline perfomance time, with the 0.3 being the most suitable value of γ. Finally, as γ grows we can see that the performance worsens gradually.

fraction of the time required by the *Exhaustive Search* 3. This confirms our original hypothesis: that cost, when defined to balance execution time, communication overhead, and computational density, serves an effective criteria for distributed model partitioning.

Specifically, the *Balanced Numbers Partitioning* approach was the most robust for all ResNet. It achieved close-to-optimal performance metrics values while keeping the total algorithm execution time low. Its ability to make split decisions early in the process, without relying on costly iterations.

While the *exhaustive search* strategy confirmed that has the lower performance values, especially as k increases, it's execution time makes it impractical for deployment scenarios (especially for bigger k values). Thus, our algorithms maintain balance between the most optimal and the most scalable solution, which is important for edge deployments where compute time and resources are limited.

Limitations

Our evaluation framework is currently limited to ResNet-based architectures and CPU-only runtime measurements. While this provides a controlled environment for performance comparisons, it does not include hardware-specific behaviors when using GPUs, FPGAs, or heterogeneous edge platforms. Additionally, transfer size estimations are not based on real network throughput, which may introduce approximation errors in dynamic environments.

Furthermore, on our experiments we focus only on sequential partitioning, however many real-world networks are based on branching structures, like attention layers or even parallel convolutional paths. These topologies were not addressed in the context of our research and present their own challenges for cost modeling and partitioning.

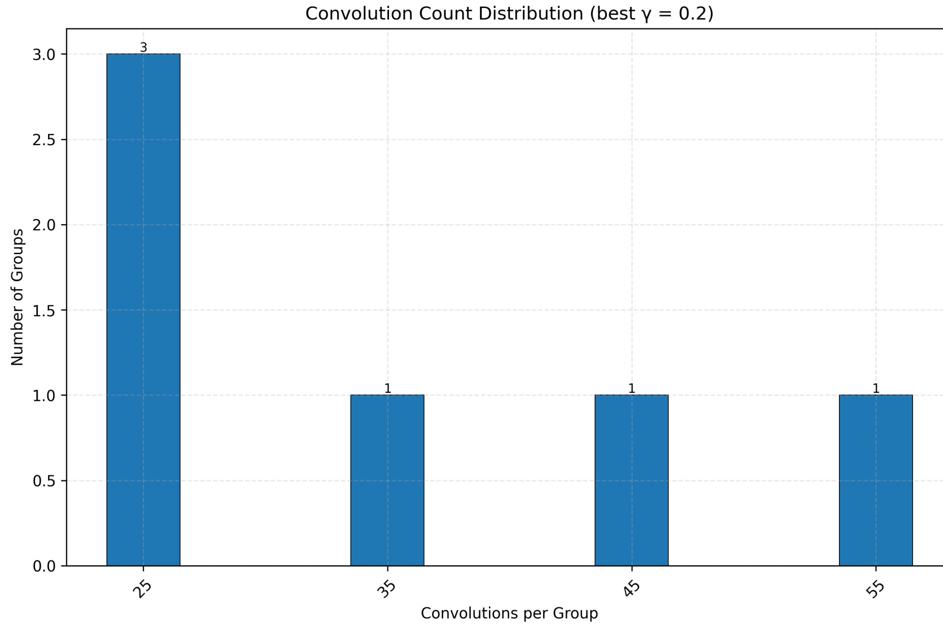

Figure 13: Distribution of convolution counts per group at the optimal γ=0.3 when attempting to split the *ResNet-152* into k=6 groups.

Future Work

Several research paths remain open for extending this work:

- **Work on Non-Sequential Models:** Future work should extend the framework to handle non-linear and graph-based architectures like Inception, MobileNetV3, or Transformer-based models, where parts cannot be easily defined as simple contiguous layer blocks.
- **Integration with Hardware Profilers:** Adding real latency profiles from edge hardware (e.g., Jetson Nano, Coral TPU) can improve the merging cost estimation accuracy. This would add support for hardware-aware partitioning and more realistic deployment use cases.
- **Dynamic Runtime Partitioning:** We could introduce support for dynamic partitioning at runtime, where decisions are adjusted based on system load, network latency, or even energy constraints. This would make the research more flexible and allow us to expand in volatile edge-cloud environments.
- **Multi-Objective Optimization Extensions:** While our merging cost combines three core metrics, future versions could explore options for new merging cost objectives. A suggested approach could be the Pareto front approximation (Deb et al., [23]) that is oftenly used in engineering for multi-objective optimizations.

Ultimately, this thesis creates the foundation for automated, cost-aware partitioning of DNNs. By moving towards more flexible architectures and deployment-aware optimizations, our goal is to create a toolkit for real-world distributed inference systems.

7 Conclusion

In this work, we experiment on a cost-driven partitioning approach for deep neural network models that tries to group a model into k sequential groups of model parts. Our method is designed to minimize the cost of forming groups and takes into account the execution time, data transfer size, and the number of convolutions, scoping on efficient model parallelism across homogenous hardware environments.

We investigated and compared two partitioning strategies: Exhaustive Search 3 and Balanced Number Partitioning 1. Through experiments on ResNet variants (*ResNet-18*, *ResNet-34*, *ResNet-50*, *ResNet-101*, and *ResNet-152*), we have shown that the Balanced Number Partioning approach, while with close-to-optimal performance results - allow us to tackle the exponential growth on the algorithm total execution time that the Exhaustive Search reports.

Our experiments confirm that intelligent grouping of model parts (each defined as a self-contained unit with a single input and output) can result in better workload balance and reduced overhead in model serving or distributed execution scenarios. Our analysis shows the trade-off between group quality and computation time alters according to the number of desired partitions and model depth.

This work creates the basis for further improvements in model partitioning, particularly in scenarios where resource constraints or latency considerations are critical.

References

[1] A. Parthasarathy, B. Krishnamachari, *Partitioning and Placement of Deep Neural Networks on Distributed Edge Devices to Maximize Inference Throughput*, IEEE, 2022.

[2] M.Garey, D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman San Fansisco, 1979.

[3] A. Duchnowski, E. Pavlick, A. Koller, *EHOP: A Dataset of Everyday NP-Hard Optimization Problems*, Saarland University, 2025.

[4] L. Fortnow, *The Status of the P versus NP Problem*, Communications of the ACM vol. 52, 2009.

[5] E. Loiola, N. Abreu, P. Boaventura-Netto, P. Hahn, T. Querido, *A survey for the quadratic assignment problem*, European Journal of Operational Research vol 176, 2007.

[6] S. Hartmann, D. Briskorn, *A survey of variants and extensions of the resource-constrained project scheduling problem*, European Journal of Operational Research vol. 207, 2010.

[7] A. Buluc, H. Meyerhenke, I. Safro, C. Schulz, *Recent Advances in Graph Partitioning*, Lecture Notes in Computer Science, 2016.

[8] D. Xu, Z. Wang, H. Chen, K. Li, *A Survey on Deep Neural Network Partition over Cloud, Edge and End Devices*, Harbin Institute of Technology, 2023.

[9] F. Kreß, M. Gottschalk, C. Hakert, K. Morik, M. Hübner, *Automated Deep Neural Network Inference Partitioning for Distributed Embedded Systems*, Computer Society Annual Symposium on VLSI, 2024.

[10] I. Rodríguez, E. Moguel, F. Sánchez-Figueroa, J. C. Preciado, *Horizontally Distributed Inference of Deep Neural Networks for AI-Enabled IoT*, Sensors, vol. 23, 2023.

[11] R. Mayer, H.-A. Jacobsen, *Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools*, ACM Computing Surveys, vol. 53, 2020.

[12] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, J. S. Rellermeyer, *A Survey on Distributed Machine Learning*, arXiv preprint arXiv:1912.09789, 2019.

[13] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, J. Zhang, Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing, Proceedings of the IEEE, 2019

[14] S. Teerapittayanon, B. McDanel, H. T. Kung, *Distributed Deep Neural Networks over the Cloud, the Edge and End Devices*, IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017.

[15] Y. J. Ku, S. Baidya, S. Dey, *Adaptive Computation Partitioning and Offloading in Real-Time Sustainable Vehicular Edge Computing*, IEEE Transactions on Vehicular Technology, 2021.

[16] N. Tho , N. Dac Hieu, N. Quoc-Thông, D. Kim, P. Kim, *Human-Centered Edge AI and Wearable Technology for Workplace Health and Safety in Industry 5.0*, Artificial Intelligence for Safety and Reliability Engineering, 2024.

[17] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, L. Tang, *Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge*, ACM SIGARCH Computer Architecture News, 2017.

[18] B. Stahl, T. Rausch, S. Dustdar, Edge AI: Deploying Deep Neural Networks on Distributed Heterogeneous Devices, IEEE Internet of Things Journal, 2021.

[19] Y. Li, T. Zhang, Y. Liu, *EdgeAI: Intelligent Edge Computing Framework for Hierarchical Deep Learning Model Partitioning*, Proceedings of the ACM/IEEE Symposium on Edge Computing (SEC), 2018.

[20] F. Ji, W. P. Tay, A. Ortega, *The faces of Convolution: from the Fourier theory to algebraic signal processing*, arXiv preprint, 2023.

[21] T. Wess, S. U. Stich, A. Kulkarni, R. Ernst, *ANNETTE: Accurate Neural Network Execution Time Estimation with Stacked Models*, IEEE Access, 2021.

[22] A. Osterwind, J. Droste-Rehling, M. R. Vemparala, D. Helms, *Hardware Execution Time Prediction for Neural Network Layers*, in Workshop of ECML-PKDD, Springer, 2022.

[23] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, *A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation, vol. 6, 2002.

[24] H. J. Jeong, H. J. Lee, C. H. Shin, and S. M. Moon, *Incremental Offloading of Neural Network Computations (IONN)*, Proceedings of ACM Symposium on Cloud Computing (SoCC), 2018.

[25] Y. Huang, D. Cheng, D. Guo, M. Shoeybi, B. Catanzaro, *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*, NeurIPS, 2019.

[26] X. Zhao, *A review of convolutional neural networks in computer vision*, Applied Intelligence, 2024.

[27] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition*, in Proc. CVPR, 2016.

[28] R. Wightman, *ResNet Strikes Back: An Improved Training Procedure in TorchVision*, arXiv:2110.00476, 2021.

[29] W.Niu, J.Guan, Y.Wang, G.Agrawal, and B.Ren, *DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion*, arXiv:2108.13342, 2021.