DTU

DTU Electro
Department of Electrical and Photonics Engineering

Study of Early Exit Neural Networks for Computer Vision

Jonathan Mikler A.

Study of Early Exit Neural Networks for Computer Vision

M.Sc Thesis
March 2025

By
Jonathan Mikler A.

Cover photo: AI image. Generated by author

Published by: DTU, Department of Electrical and Photonics Engineering, Ørsteds Plads, Building 326, 2800 Kgs. Lyngby, Denmark

https://electro.dtu.dk/

Approval

This thesis has been prepared over five months at the Department of Electrical and Photonics Engineering, at the Technical University of Denmark, DTU, in partial fulfillment for the degree Master of Science in Engineering, MSc Eng.

Disclaimer

In the spirit of academic transparency, I hereby disclose the use of Generative Artificial Intelligence tools throughout the preparation of this thesis. These tools were employed primarily to enhance textual clarity, assist in document structuring, and provide support in code development and implementation. These tools were employed primarily to enhance textual clarity, assist in document structuring, and provide support in code development and implementation. All external sources and references are appropriately cited in accordance with academic citation standards. This declaration aims to maintain full intellectual honesty and transparency regarding the research and writing process.

Jonathan Mikler A.
Student Number: S222962

Date: 20 August 2025

Acknowledgements

As this is the final day of my thesis, and after working non-stop for too long, I'll keep my gratitude brief.

Firstly I am most grateful to my supervisor Prof. Lazaros Nalpantidis, for the guidance and the insights along this work. I owe an equal amount of gratitude to Fabio Montello, my co-supervisor. This project demanded a lot of turns and jumps and his experience kept the ship well oriented.

This project would not have been possible without Open-Source software. The power of openly sharing has positive consequences more significant than we might think. I am grateful to those who help and advocate for this, as I have seen here its benefits first hand.

To my family; to all of you, in every corner of the planet.

Abstract

This thesis investigates deployment challenges and performance characteristics of early-exit neural networks for computer vision, focusing on the Lightweight Vision Transformer with Early Exit (LGViT) model.

The research addresses two aims: making early-exit models production-ready and expanding understanding of their performance. The work presents a redesigned implementation (EEVIT) optimized for ONNX export using PyTorch's dynamic behavior capturing features.

Four studies are conducted: profiling analysis identifying bottlenecks, evaluation of confidence threshold effects, platform benchmarking across computing environments, and class-wise performance analysis revealing relationships between visual features and exit behavior.

Results show EEVIT achieves a 2.08× speedup with 87.31% accuracy on CIFAR100. The study reveals insights about implementation penalties, optimal threshold configurations, class-specific patterns, and platform-dependent behavior for deploying early-exit architectures in resource-constrained environments.

Table Of Contents

1 Introduction

Deep learning models have enjoyed widespread success in computer vision tasks due to their impressive performance(Azizov et al., 2024). However, this success has been accompanied by an increase in size and computational complexity, leading to ever-growing computational requirements.

On one side, a clear trend in size increase can be seen in the increase in layer that State of the Art CNN models, growing from 8 layers to 19 layers to 152 layers (AlexNet, VGGNet and ResNet respectively) (Teerapittayanon et al., 2016). A more recent concern, related to the widespread adoption of the transformer model (Azizov et al., 2024), is that the computational complexity of the self-attention mechanism (the heart of the transformer) scales quadratically with sequence length, expressed as $O(n^2)$ (Keles et al., 2022)

This trend makes powerful models less suitable for environments with limited computing resources and low inference latency needs—precisely the conditions found in edge computing devices used in robotics applications and other domains(Teerapittayanon et al., 2016).

Different solutions have been proposed to render more efficient the ViTs. Among them, pruning and quantization. Pruning aims to identify and remove less important computations while quantization reduces the representation precision of the model's parameters. Rethinking the attention function has been also proposed, with the goal of reducing its computational complexity – Among them the Linformer (Wang et al., 2020) and the SWIN transformer (Liu et al., 2021).

Parallel to these approaches, other techniques break the assumption of a fixed forward computation graph. Collectively, the networks produced by these methods can be called *Dynamic Neural Networks*. These models selectively activate only relevant sub-graphs of the architecture based on input complexity, potentially offering more substantial efficiency gains for varied inputs(Montello et al., 2025).

Neural network models often encounter significant variations in task complexity across different dataset samples. Traditional neural networks aim to generalize well over the entire dataset, but this can result in over-parameterization for a large subset of samples that require less computation (Scardapane et al., 2020).

Research has shown that Convolutional Neural Networks (CNNs) can extract meaningful features in their early layers, which are often sufficient for confident classification ((Panda et al., 2015)). However, unnecessary processing of sample information can lead to *overthinking*, where initially useful representations become corrupted due to excessive computation, resulting in reduced confidence or incorrect predictions ((Kaya & Dumitras, 2018)).

Early-exit Neural Networks represent a promising model design approach that aims to improve latency while minimizing accuracy compromises. Conceptually, neural network models are computational graphs; the early-exit approach attaches additional computational steps to the base graph to produce **early** outputs with lower latency than the unmodified model. Then, if the *confidence* in the output is satisfactory, it is taken as the model's output, else the model continues its computation onto the next exit. A generic schematic description is presented in Figure 1.

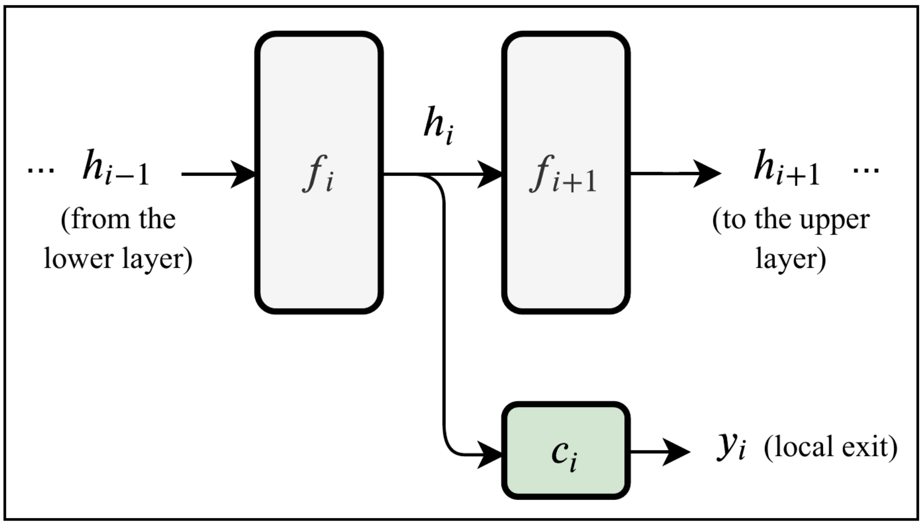

Figure 1: Graphical depiction of a generic early-exit in neural network architectures. taken from (Scardapane et al., 2020)

Despite their potential, adoption of early-exit models in production environments has been challenging due to the dynamic nature of their computational graph. Production environments, especially in edge computing scenarios, typically require minimal dependencies—both to simplify monitoring and updates, and to meet specific domain requirements[1].

Furthermore, production environments benefit from remaining agnostic to the development frameworks used to build the models they employ. This framework-agnosticism allows for seamless replacement of older models with newer, better-performing ones. The ONNX (Open Neural Network eXchange) standard addresses this need by providing a framework-independent representation for neural networks.

While neural networks are typically built using powerful frameworks like PyTorch or TensorFlow that come with a cornucopia of development tools, these same frameworks represent undesirable dependencies in production. Converting models created with any framework to ONNX format simplifies their adoption in framework-agnostic production environments.

This context highlights the adoption challenges for early-exit models: until recently, capturing their dynamic nature in ONNX was very cumbersome, when it was not impossible. For PyTorch, for instance, flow control operations were only

[1]Think for example of use cases were safety is of critical consideration, and some libraries used for the business logic do meet the performance requirements

added to exportable ONNX operations earlier this year[2]. Consequently, industries that could benefit from early-exit models' performance breakthroughs have been missing out the fruits of their work.

This thesis investigates one of such models, *Lightweight Vision Transformer with Early Exit (LGViT)* (Xu et al., 2023), one of the latest high-performing early-exit models, and the efforts required to make it production-ready. This works sets two goals: (1) To assess and implement the necessary efforts to port developed early-exit models into a production environment and (2) To widen the understanding of the performance nature of these models, with the intention of gaining insights in how they can be extended to other computer vision problems.

It begins by dissecting the logic implemented in the original work and presents a redesign optimized for ONNX exportability. Following this, a new implementation built from scratch is presented, compatible with the recently released dynamic behavior capturing features in PyTorch. With this newly exportable model, four performance studies are conducted:

1. A profiling study to identify performance bottlenecks, yielding insights for implementation decisions
2. An short analysis of the confidence threshold to better understand its role in performance.
3. A platform performance benchmark in a production environment (using an Nvidia Jetson Orin computer) and how it compares to a consumer computer.
4. A class performance analysis using the CIFAR100 dataset, providing insights into which input types yield the best results and what can be inferred about their visual features

The document is structures as follows: Section 2 presents the algorithms upon which the model leverages, followed by Section 3 with an overview of the most relevant commercial technologies. Section 4 presented both the recount of the related work done in the realm of early-exit for computer vision, and delves into the logic implementation of the original LGViT work, followed by highlighting its setbacks for porting. Section 5 presents the new implementation design for LGViT, where dynamic behavior export to ONNX is possible, leveraging on recently released dynamic behavior capturing features in PyTorch. Following this, the different benchmark studies are presented, whose discussions are then addressed in Section 6. Finally, some ideas based on the results of the work are presented for future consideration in Section 7

[2]PyTorch version 2.6.0 was officially released on January 29, 2025

2 Background

2.1 Transformers in Computer Vision

The transformer model was originally conceived for sequence-to-sequence tasks, such as translating sentences from one language to another. To effectively process sequences, understanding the relationships between sequence elements, called *tokens* can be helpful. This understanding is achieved through a process known as (self-)attention. Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence (Vaswani et al., 2017).

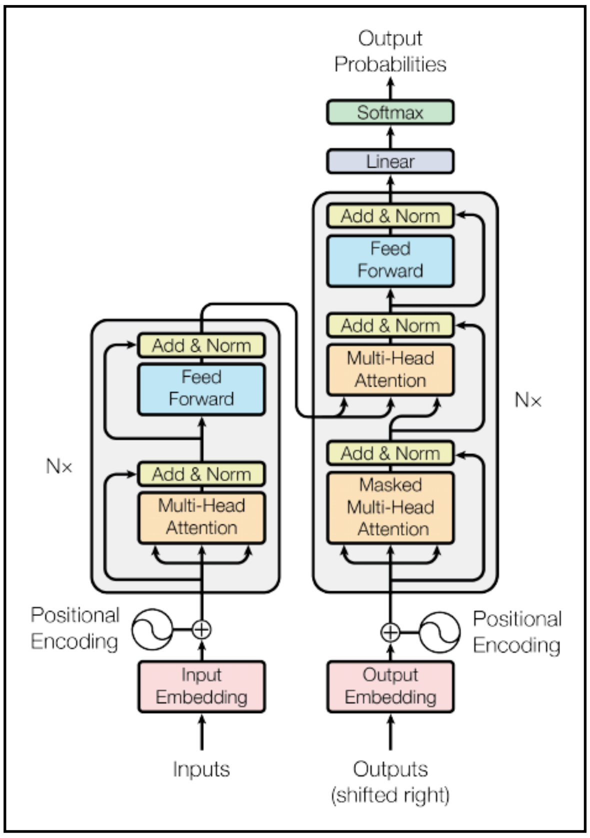

Figure 2: Original transformer architecture as proposed in (Vaswani et al., 2017).

The transformer, proposed in (Vaswani et al., 2017), follows an established encoder-decoder structure. It core module present in both encoder and decoder can is presented in Figure 2. The encoder maps input sequence tokens ($\boldsymbol{x} = x_1, ..., x_n$) into abstract representations ($\boldsymbol{x} = z_1, ..., z_n$), which are then used by the decoder to produce an output sequence ($\boldsymbol{y} = y_1, ..., y_m$), typically generating one element at a time. The decoder is *auto-regressive* or *recurrent*; together with the abstract representations, the output is fed back to the decoder, to produce the next element in it. The key element in the transformer is its novel attention mechanism, which is used to produce both the intermediate representation $\boldsymbol{z}$ and the final output $\boldsymbol{y}$.

What distinguished the transformer model and made it so performant was its innovative approach to attention. Previous attention methodologies utilized convolution to relate tokens in a sequence, with computational complexity increasing based on the distance between tokens, as seen in sequence processing models like ConvS2S or ByteNet (Rush, 2018).

The transformer introduced *Scaled Dot-Product Attention.* For each input token x_i, three vectors are computed: *query*, *key*, and *value* (q_i, k_i, v_i respectively). All vectors are obtained from multiplying the input tokens by three different learned weight matrices (W_Q, W_K, W_V respectively).

$$q = xW_Q$$
$$k = xW_K$$
$$v = xW_V$$

The attention mechanism is computed as follows:

$$\text{Attention}(q, k, v) = \text{softmax}\left(\frac{q \cdot k^t}{\sqrt{d_k}}\right)v$$

It calculates a weighted sum of the *value* vectors, where the weight assigned to each is determined by the scaled and softmax-transformed dot product of the query and key vectors. The resulting vector is called the *attention* vector. Each input sequence token produces its own attention vector. In practice, attention vectors for all input tokens are computed simultaneously by stacking all input tokens into a matrix X and multiplying by the respective matrices.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q \cdot K^t}{\sqrt{d_k}}\right)V$$

Some details are of noteworthy importance:

- For large values of d_k, the dot products can grow excessively large in magnitude, pushing the softmax function into regions with extremely small gradients. To counteract this effect, the dot products are scaled by $\frac{1}{\sqrt{d_k}}$ (Vaswani et al., 2017)
- Already in the original transformer paper further, the authors further improve self-attention by applying it in parallel, introducing *Multi-Headed Attention.* This approach learns h sets of matrices W_Q, W_K, W_V, enabling "multiple representation spaces" (Vaswani et al., 2017). The h attention output matrices $Z_1, ..., Z_h$ are concatenated and linearly projected into a single attention matrix Z: $\text{MultiHead}(Q, K, V) = \text{Concat}(Z_1, ..., Z_h) \cdot W_O$
- Unlike recurrent or convolutional neural networks, the transformer architecture contains no inherent notion of token order. Without adding positional information, the model would be invariant to token reordering, treating the sequence as a set rather than an ordered list. The original transformer paper introduced

sinusoidal positional encodings, which use sine and cosine functions of different frequencies:

$$\mathrm{PE}_{\mathrm{pos},2i} = \sin\left(\frac{\mathrm{pos}}{10000^{2\frac{i}{d_{\mathrm{model}}}}}\right)$$

$$\mathrm{PE}_{\mathrm{pos},2i+1} = \cos\left(\frac{\mathrm{pos}}{10000^{2\frac{i}{d_{\mathrm{model}}}}}\right)$$

Later architectures have explored learned positional embeddings and other variants, but the core idea remains the same: explicitly encode position information into token representations to enable the self-attention mechanism to consider sequence order.

2.1.1 Vision Transformers

Vision Transformers (ViTs) (Dosovitskiy et al., 2021) represent a paradigm shift in computer vision, applying the transformer architecture—originally designed for natural language processing—to image recognition tasks. Prior to 2021, convolutional neural networks (CNNs) dominated computer vision. Although the self-attention mechanism had been introduced in 2017 (Vaswani et al., 2017), the ViT was the first to successfully leverage this approach at scale for vision tasks, by avoiding applying complex engineering modification to the self attention mechanism and instead leveraging on slicing the image into patches of a determined size and applying attention onto their a latent space projection of them.

The core innovation of ViTs lies in their approach to images: rather than processing them through convolution layers, they divide images into fixed-size patches and treat these patches as tokens (analogous to words in NLP). This allows the standard Transformer architecture to be applied to images with minimal modifications.

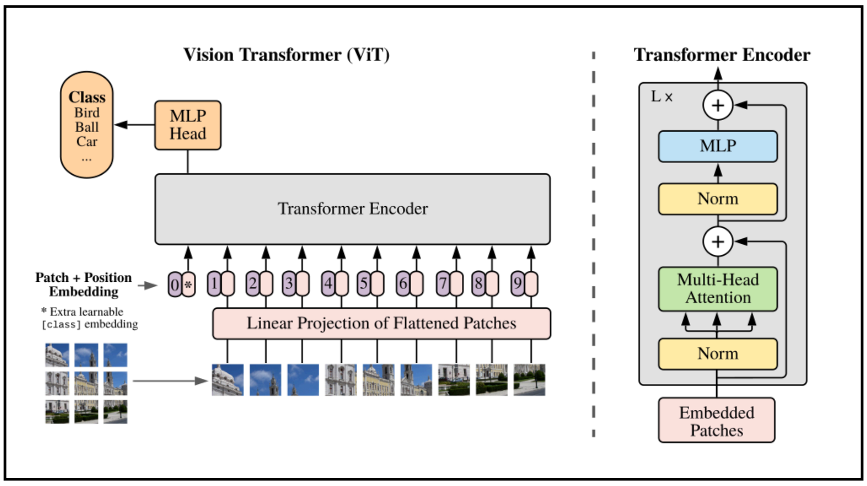

Figure 3: Diagram from the original ViT paper (Dosovitskiy et al., 2021)

Image Representation

The input image, $x \in \mathbb{R}^{H \times W \times C}$ is divided into a sequence of N flattened 2D patches,$r \in \mathbb{R}^{P \times P}$, resulting in a sequence of vectors with dimension $\mathbb{R}^{N \times (P^2 \cdot C)}$, where $N = \frac{\mathrm{HW}}{P^2}$ represents the effective sequence length. These patches are then linearly projected to a latent space of dimension D using a learnable projection matrix. At this point, the projected patches are referred to as “tokens,” following transformer terminology.

Before processing through attention blocks, positional information must be added to preserve spatial relationships that would otherwise be lost when treating the image as a sequence. The authors used learnable 1D position embeddings, noting that 2D-aware embeddings did not significantly improve performance.

Following established practice from BERT (Devlin et al., 2018), a special classification token (`[CLS]`) is prepended to the sequence. After processing through the transformer, the state of this token serves as the image representation for classification tasks, which is then processed by a simple MLP head.

Architecture

The ViT architecture closely follows the original transformer design. Each block consists of Multi-headed Self-Attention (MSA) followed by a Multi-Layer Perceptron (MLP), with layer normalization (LN) applied before each block and residual connections around each block, as shown in Figure 3.

Unlike the original transformer, ViT employs only the encoder portion since image classification doesn’t require the sequential generation provided by the decoder. The MLP contains two layers with a GELU non-linearity.

Less inductive bias than CNNs: A key difference between ViTs and CNNs is their inductive bias. While CNNs have strong spatial inductive biases built into every layer through local convolutions and pooling operations, ViTs have substantially less image-specific inductive bias. In ViTs, only the initial patch extraction and positional embeddings incorporate knowledge about 2D image structure. The self-attention layers are global, allowing any patch to attend to any other patch from the first layer onward, which enables the model to capture long-range dependencies more easily than CNNs (Dosovitskiy et al., 2021).

Model Variations

The original ViT paper experimented with three model sizes:

Model	**Layers**	**Hidden Size**	**MLP Size**	**Attention Heads**	**Parameters**
ViT-Base (B)	12	768	3072	12	86M
ViT-Large (L)	24	1024	4096	16	307M
ViT-Huge (H)	32	1280	5120	16	632M

Table 1: Model variations and their parameters from the original ViT paper (Dosovitskiy et al., 2021)

Training and Results

The original ViT models were pre-trained on large datasets like ImageNet, ImageNet-21k, or JFT-300M before fine-tuning on specific tasks. Of particular interest for this work is the performance of the ViT-B/16 configuration (Base model with 16×16 patch size) for the fine-tuned case of CIFAR100:

Model	**Accuracy [%]**
ResNet 50×1	97.67
ResNet 152×2	92.05
ResNet 200×3	93.53
ViT-B/16	91.87

Table 2: Performance comparison on CIFAR100 selected results from (Dosovitskiy et al., 2021)

These results demonstrated that the ViT approach could match or exceed state-of-the-art CNN models when pre-trained on sufficient data, despite having less image-specific inductive bias.

2.2 Early Exits in Deep Learning for Computer Vision

The early-exit approach proposes to provide additional computational flows to that of a given model, with the intention of producing outputs faster than the main model would have produced them (also called *Backbone*). Then, discriminating with some heuristic, such *early* output is potentially taken as the model's output.

The design and placement of this *early exits* of the network are guided by the guiding principle that the time spent in the early exit computation should be lower than the time it would have taken to process the complete backbone flow of the model. Figure 4, taken from (Bakhtiarnia et al., 2021) is a conceptual diagram of the idea.

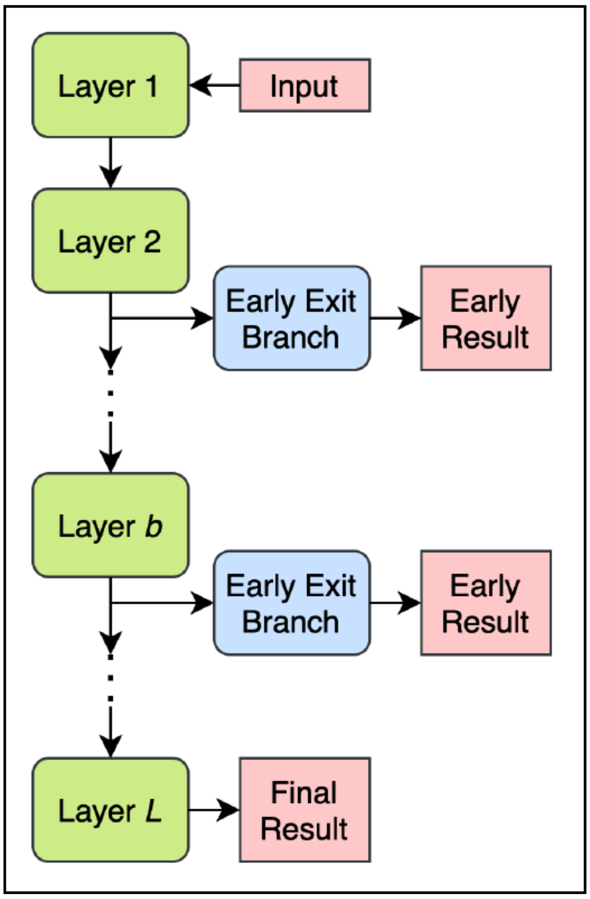

Figure 4: Schematic representation of early exits in a neural network model (Bakhtiarnia et al., 2021).

The early exit approach presents some questions: What should be the architecture of the branches? Where along the main computation flow should they be placed? How should they be trained and what should be the heuristic for deciding on the early predictions? All these are design question without a clear cut answer and research has been done to explore possible solutions to this challenges. A short overview of the guiding principles is presented hereunder. For a more detailed explanation, refer to (Scardapane et al., 2020), and for a comprehensive survey of the models proposed in the last five years, refer to (Montello et al., 2025).

2.2.1 Design and Placement of Early Exits branches

Processing partially processed data of a model with the intent of early inference is obviously dependant on the dimensionality of this data. For example, early layers in most CNNs have a very high dimensionality, which would result in extremely high-dimensional classifiers. In the case of having a transformer as the main model, the branch would have as domain the hidden states dimensionality.

As for the placement, procedures typically compare the computational *gain* of exiting at a given location, i.e. the computation cost *saved* from not continuing down the main computation flow. (Scardapane et al., 2020) present an approach were a relation between the computation delta of the main computation flow and that of a potential early exit are used to determine the early exit positions. Alternatively, (Bakhtiarnia et al., 2021) argues for a fine-train exit coverage, basically attaching exit to every step on their backbone, arguing that if earlier exits are more accurate than later ones, the network will use the optimal amount of resources at every inference. To illustrate a third approach to placement, (Li et al., 2023) implement a low-cost exit prediction, to determine with high confidence where will the model exit, and place a unified early exit at that location to carry out the exit processing and decision.

2.2.2 Training of Early Exit Networks

Just as with the design and placement of the early exit branches, the training strategies are as well a design challenge. however, some guiding principles have been developed along the years. (Scardapane et al., 2020) mentions that three broad training strategies:

- **Joint training**: The overall architecture can be trained jointly by defining a single optimization problem that embraces all intermediate exits.
- **Layer-wise**: At each iteration a single auxiliary classifier is trained together with the backbone layers preceding it.
- **Classifier-wise**: The backbone network is trained first, and then separately train the auxiliary classifiers on top of it [72]. This can be interpreted as a very primitive form of knowledge distillation.

Needless to say, there are many variations to this strategies, as researchers explore the one that yields the best results for their design.

2.2.3 Inference in Early Exit Networks

The decision to take an intermediate - *early* - output is determined by the desire to select the prediction with the highest confidence for the given input. This *optimal* prediction would correspond to an optimal exit, early or not. This related to the phenomenon referred to as *over-thinking* (Kaya & Dumitras, 2018), in which later predictions on an input are of lower confidence than earlier ones ones, or even wrongly classified. The process by which a early prediction is selected is called the *exit strategy* or *exit policy.* The survey work done by (Rahmath P et al., 2024) classifies them into rule-based and learnable. Common rule-based strategies are:

- **Max-Softmax**: Softmax is applied to a vector of logits and the probability of the most likely element is compared to a pre-determined threshold value. This value can be different for each early exit or not (Scardapane et al., 2020).
- **Entropy**: Predictions are considered confident if the entropy value falls below a predefined threshold, prompting the side branch to classify the input (Rahmath P et al., 2024).
- **Patience**: Early predictions whose entropy is lower than a predefined threshold are tracked, and exit occurs after the same prediction has been done across a number of exits (Xu et al., 2023).

More robust exit strategies have been explored beyond rule-base heuristics. (Rahmath P et al., 2024) mentions some models using Markov decision Processes as policy generators, allowing to take environment factor into consideration, by training a *Deep Query Network* (DQN) as the policy. Other models learn the policy together with the model's parameters (Bolukbasi et al., 2017). Finally, although not directly relevant to this work, an interesting alternative to the exit strategy design involves distinguishing different kinds of uncertainty, referred as *aleatoric* and *model* uncertainties in (Xia, 2024), and using a Dirichlet framework to directly measure model uncertainty.

"[Aleatoric uncertainty] usually refers to the irreducible uncertainty which arises from the natural complexity of data, such as class overlap and label noise. [model uncertainty] on the other hand, is the uncertainty caused by a lack of training data or insufficient model complexity." (Xia, 2024)

3 Tools of the Trade

3.1 ONNX

ONNX is an open format designed to represent machine learning models by providing a common set of data types and operations for extensible graph definitions that represent computational graphs.

The need for ONNX in this work stems from the desire to asses how straightforwards or easy is it to deploy early exit models. The deployment of a machine-learned model into production usually requires replicating the entire ecosystem used to train the model, most of the time with a docker. Once a model is converted into ONNX, the production environment only needs a runtime to execute the graph defined with ONNX operators. This runtime can be developed in any language suitable for the production application (ONNX Community, 2025)

Deep learning with neural networks is accomplished through computation over dataflow graphs. Some frameworks (such as CNTK, Caffe2, Theano, and TensorFlow) make use of static graphs, while others (such as PyTorch and Chainer) use dynamic graphs. The graph serves as an Intermediate Representation (IR) that captures the specific intent of the developer's source code, and is conducive for optimization and translation to run on specific devices (CPU, GPU, FPGA, etc.). By providing a common representation of the computation graph, ONNX helps developers choose the right framework for their task (ONNX Project Contributors, 2023).

The ONNX framework consists of three main components: (1) a definition of an extensible computation graph model, (2) definitions of standard data types, and (3) definitions of built-in operators (the "operator set"). The first two elements constitute the ONNX intermediate representation specification, while the third provides the available operations and functions used with the IR specification to compose an intermediate representation of a model. A key feature of the ONNX operator set is its extensibility, allowing implementations to add operators with semantics beyond the standard set that all implementations must support.

In ONNX, the top-level construct is a `Model`, which contains all necessary metadata to run the model, including the opset version and the graph itself. The computation dataflow graph is structured as a topologically sorted list of nodes forming a cycle-free graph, with each node representing an operator call. Every node has zero or more inputs and one or more outputs.

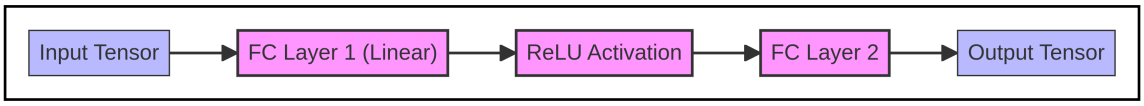

Figure 5: Two layer MLP forward pass

A basic MLP design is presented in Figure 5, and Figure 6 (left) visualizes an ONNX graph representing the model's computations. Notably, matrix multiplication and bias addition are combined into a single General Matrix Multiplication operation. Figure 6 (right) is a more explicit representation, illustrating how different ONNX implementations or framework backends can produce different IRs for semantically equivalent models.

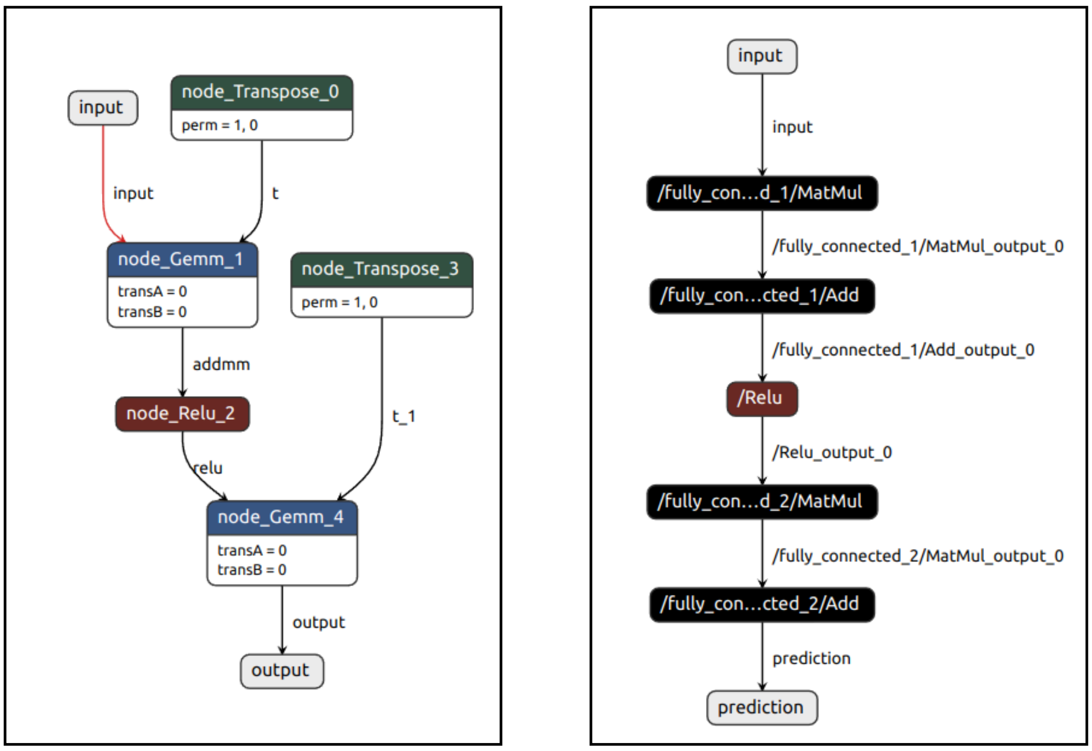

Figure 6: Visualizations of the Two layer MLP computational graph. Left: Standard representation. Right: Explicit intermediate representation.

3.2 PyTorch

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. It was created by the Meta AI research lab in 2016 and has enjoyed widespread adoption[3]. PyTorch is written in C++ and Python. It's ease of use lies on several aspects. Firstly, PyTorch's basic data structures, the `Tensor` and `Module`, make it easy for anyone with some coding and linear algebra experience to implement neural network architectures that enjoy some very complex features.

A key feature of PyTorch is that by default, it uses *eager* execution: the computational graph is is defined at runtime. This allows for on-the-spot graph manipulations as well as use of standard debugging tools, which contribute to a smoother development experience (Mario, 2021).

3.2.1 PyTorch meets ONNX

While it's main focus is neural networks modelling, PyTorch has a plethora of utilities and tools for different purposes. One of those modules which played a

[3]The success of PyTorch can be partly attributed to its collection of modules, but not exclusively. Other reasons are Meta's backup, strong academic adoption and being open source. This last one practically made this project possible

critical part in the development in this work is `torch.onnx`. PyTorch's ONNX module captures the computation graph from a native PyTorch `Module` model and converts it into an ONNX graph, which then can be consumed by any of the many runtimes that support ONNX.

It is important to note that there are two *flavors* of ONNX API in `torch.onnx`, which vary on the compilation backend used: The established `TorchScript` and the newer `TorchDynamo`. Both are compilation engines part of the `torch.compile` toolset.

These differ in how the computational graph is created. While `TorchScript` uses traditional tracing tools to capture the computation instructions and compile them, Dynamo uses Python's frame evaluation API (Foundation, 2016) to capture the computation instructions. This difference allows for Dynamo to capture flow-control behavior; computations whose execution is data dependant (like for example parts of the graph to be executed only under certain conditions), such features were not entirely possible for the `TorchScript` engine[4].

Flow Control

Dynamo produces a graph break when it sees data-dependent control flow (Team, 2025a). Concretely, it is unable to trace the following into a full graph:

```python
@torch.compile(backend="eager", fullgraph=True)
def f(x):
    if x > 0:
        return x.sin()
    else:
        return x.cos()

x = torch.tensor(1.)
f(x)
```

This is because Dynamo traces a function by running through it with FakeTensors (tensors without storage), and so it is unable to determine if x (as a FakeTensor) is greater than 0 and it ends up falling back to eager-mode PyTorch (PyTorch Team, 2023).

PyTorch offers `torch.cond`, a custom version of the if/else statement as part of their 'High Order Operators' which allows for the *true* and *false* execution graphs to be traced. Semantically, torch.cond performs the following:

```python
def cond(pred, true_fn, false_fn, args):
    if pred:
```

[4]These modules work deep inside the torch ecosystem and can be quite complex to grasp, for which reason I refer interested readers to the torch documentation on the matter

```python
        return true_fn(*args)
    else:
        return false_fn(*args)
```

When Dynamo sees a torch.cond, it will:

- Trace both the true_fn and false_fn and turn them into subgraphs
- emit a call to `torch.ops.higher_order.cond(pred, true_fn_graph, false_fn_graph, args)` into the graph.

Only recently[5] the operator set of ONNX used by PyTorch was extended to include `torch.cond`. Nothing better than a practical example to cement understanding. The following is a simple conditional model, implemented in PyTorch, for which an ONNX model is presented in Figure 7.

```python
class CondModel(torch.nn.Module):
    def __init__(self): super().__init__()

    def forward(self, x):
        def true_fn(x, z): x = x + 1.0; return x
        def false_fn(x, z): x = x - 1.0; return x

        x = torch.cond(pred=x.sum() > 0, true_fn=true_fn,
        false_fn=false_fn, operands=(x,))
        return x
```

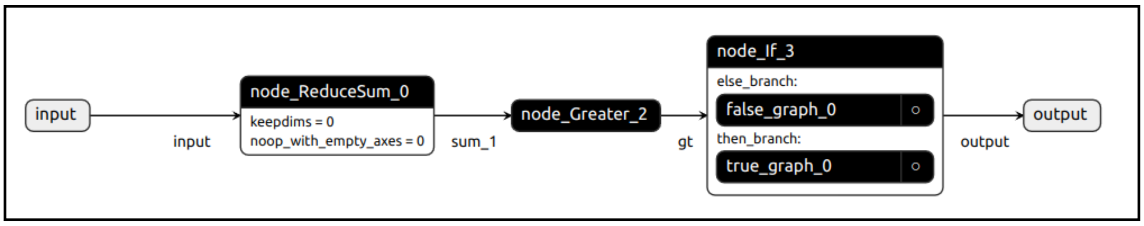

Figure 7: ONNX model for a simple conditional neural network

[5]With the release of torch 2.6.0

4 Related Work

4.1 Early Models

The concept of early exits has been explored for several years, surfacing probably in 2015. Since then, research interest in conditional computation and dynamic inference has grown steadily (Montello et al., 2025).

Within the field of neural networks for computer vision, early exits came to surface in 2015 with proposal of *conditional deep learning* (Panda et al., 2015). They attached iteratively linear classifiers between convolutional starting with the first layer, and monitoring the output to decide whether a sample can be exited early. the following year, the BranchyNet model came out (Teerapittayanon et al., 2016). They extend the work of (Panda et al., 2015), by attaching entire branch networks of more than a single layer at determined locations in the model.

BranchyNet expanded this concept by implementing more complex branch networks at strategic locations. Their work addressed several practical aspects:

- placement of branch points
- structure of branches
- exit point classifier design
- exit criteria and computational costs
- training methods for all classifiers

BranchyNet addressed the correlation between prediction accuracy and model size by adding side branches to the main network backbone. For training, they computed cross-entropy loss at each exit and combined them into a global loss function[6]. As for early exit criteria, the prediction entropy was used as exit strategy.

Their results showed minimal accuracy impact (less than 1% decrease) with substantial latency reductions across LeNet, AlexNet, and ResNet architectures on MNIST and CIFAR10. CPU inference times decreased significantly: LeNet from 3.37ms to 0.67ms, AlexNet from 9.56ms to 6.32ms, and ResNet from 137ms to 73ms.

4.2 Vision Transformers with Early Exit

Given the success the transformer architecture had on computer vision tasks, it was a matter of time until the early exit paradigm would be incorporated in them. Indeed, in 2021 (Bakhtiarnia et al., 2021) did just that. They applied early exit mechanisms to Vision Transformers and explored seven different exit designs based on the ViT-B/16 model:

- **MLP-EE**: A simple branch model consisting in an mlp processing the (early) classification token

[6]This corresponds to the *end-to-end* strategy mentioned in Section 2.2

- **CNN-X**: Three branch models with CNN architecture, motivated by contemporary research which focused greatly on this kind of architecture and the low parameter and computation overhead of the CNN algorithm
- **ViT-EE**, MLP-Mix-EE, ResMLP-EE: Three branch models with the intention of finding an alternative to the locality of the receptive field of CNNs models. They use both a short transformer encoder as exit branch (ViT-EE) as well as other algorithms of lower computational complexity, inspired by research on optimizing the transformer itself by replacing the attention with them.

They experimented on image classification with the CIFAR10, CIFAR100, and Fashion MNIST datasets for all seven proposed architectures. In their experiments they observed poor performance of the MLP-EE branch, as it does not contain enough parameters and layers to extract useful features. Regarding the CNN-based branches, they outperformed other types in the first locations, and the authors argue that it is likely because the fusion of convolutional layers that capture local interactions well, with the global attention of the backbone. Finally, they show the ResMLP-EE outperforming the ViT-EE for image classification.

4.3 The 'LGViT' model

4.3.1 Summary

The work conducted by (Xu et al., 2023) explores a novel modification to the traditional Vision Transformer (ViT) by implementing what they call "heterogeneous exiting heads" with the primary goal of improving inference latency while maintaining acceptable accuracy. As highlighted in their paper, they discovered that direct application of early exiting methods to Vision Transformers without careful consideration leads to substantial performance degradation. Through systematic investigation, the researchers identified two critical constraints affecting early exiting performance in ViTs: first, inadequate feature representations in shallow internal classifiers, and second, limited ability to capture target semantic information in deep internal classifiers. These observations emerged from a comprehensive probe study that evaluated the effectiveness of different early exit methods, including vanilla MLP-based approaches, convolution-based exits, and attention-based exits.

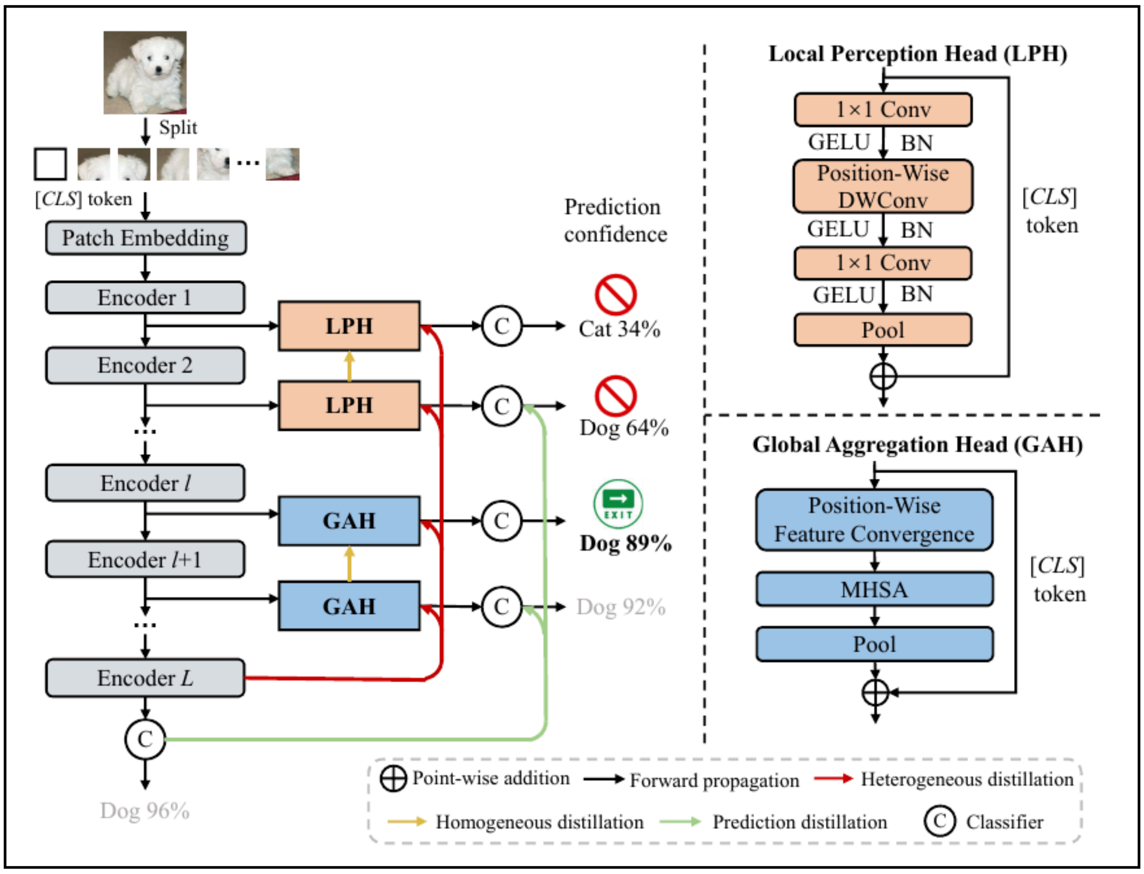

Figure 8: Overview of the proposed early-exiting ViT framework by (Xu et al., 2023)

To address these limitations, LGViT conducted a systematic study to determine the optimal location and type of intermediate heads within the ViT structure. The researchers implemented two distinct types of intermediate heads: For the first half of exiting points, Local Perception Heads (LPHs) –based on convolution– designed to enhance local information exploration, and Global Aggregation Heads (GAHs) –based on self-attention– for the second half, to augment global information acquisition. Both types can be seen in Figure 8.

The placement of each intermediate head followed an approximately equidistant computational distribution strategy, where exiting points were positioned to maintain consistent multiply-accumulate operations (MACs) between adjacent exits. This approach ensured balanced computational load while optimizing the effectiveness of different head types at their respective network depths. The final configuration for their published pre-trained weights positioned exits at layers 4 through 11, a distribution that emerged from both their ablation study on optimal head placement and adherence to the computational equidistance principle.

Intermediate Head types

<u>Local Perception Head</u>

The Local Perception Head focuses on capturing local information through convolutional operations. This design choice stems out of previous research on the efficacy of convolutional methods integrated with attention, demonstrating their effectiveness in early layers. To further support this choice, the authors compare the feature extraction capacity of convolutional and MLP based early exits and compared them to that of ResNet by means of Central Kernel Alignment. The study shows higher similarity for the convolutional variance. The LPH computation is the following:

$$\mathrm{LPH}(\boldsymbol{X}_{\mathrm{en}}, m) = \mathrm{Pool}(\mathrm{Conv}_{1\times 1}(G(\boldsymbol{X}_{\mathrm{en}}, m))) + \boldsymbol{X}_{\mathrm{CLS}}$$
$$G(\boldsymbol{X}_{\mathrm{en}}, m) = \mathrm{PDConv}(\mathrm{Conv}_{1\times 1}(\boldsymbol{X}_{\mathrm{en}}), m)$$

The LPH first employs a 1×1 convolution layer to expand dimensions. Subsequently, the expanded features are passed to a position-wise depth-wise convolution (PDConv) with $k \times k$ kernel size that depends on the exiting positions m, with smaller kernels for deeper layers. Then the features are projected back into the original patch space using a 1×1 convolution and then passed to an average pooling layer. The [CLS] token is then added to the pooled out-put to facilitate the fusion of global information from the original backbone and local information from the convolution. The output of LPH is then passed to the internal classifier.

Global Aggregation Head

The Global Aggregation Head, connected to deeper exiting points, is designed to extract global information. As per the authors: "*…integrates features from locally adjacent tokens and then compute self-attention for each subset to facilitate target semantic information exploitation*".

The authors first employ what they call a *position-wise feature convergence* (PFC) block to aggregate features from the exiting point. Analogous to PDConv, the window size s of PFC also depends on the exiting position m In the PFC block, down samples the embedded patches by average pooling them. Then the integrated features are passed through multi-head self-attention and a pool layer.

$$\boldsymbol{X}_{\mathrm{en}_{\mathrm{converged}}} = \mathrm{PFC}(\boldsymbol{X}_{\mathrm{en}}, m) = \mathrm{Pool}_m(\boldsymbol{X}_{\mathrm{en}})$$

$$\mathrm{GAH}(\boldsymbol{X}_{\mathrm{en}}, m) = \mathrm{Pool}\Big(\mathrm{MHSA}\Big(\boldsymbol{X}_{\mathrm{en}_{\mathrm{converged}}}\Big)\Big) + \boldsymbol{X}_{\mathrm{CLS}}$$

Training

Quite different from the pre-train, fine tuning training schema done in the original ViT work (Dosovitskiy et al., 2021), (Xu et al., 2023) implements a custom training strategy with the intention of tackling performance degradation that stems from common approaches, after comparing different training schemes (Dosovitskiy et al., 2021). It involves two stages. First, an end-to-end approach helps the backbone ViT achieve high accuracy. In the second stage, the parameters of the backbone are frozen, and only the exiting heads are updated through self-distillation techniques. This careful training procedure helps minimize information loss between the heterogeneous exit architectures and facilitates the transfer of knowledge from deeper to shallower exits (Xu et al., 2023)

Results

The researchers evaluated LGViT across multiple Vision Transformer architectures, including ViT, DeiT, and SWIN models. However, our focus here is primarily on their results with the ViT model for the CIFAR100 dataset. Their experimental results show a 1.87× speedup with only a minimal accuracy drop of around 2% compared to the original models (88.5% for LGViT and 90.8% for ViT-B/16).

Method	Parameters count	CIFAR100	Food-101	ImageNet-1K
ViT-B/16	86 M	90.8% | 1.00 X	89.6% | 1.00 X	81.38% | 1.00 X
LGViT	101 M	88.5% | 1.87 X	88.6% | 2.36 X	80.3% | 2.7 X

Table 3: Results of the LGViT model compared to the original ViT-B/16 on different datasets. The accuracy is reported as a percentage, and the speedup factor is shown in parenthesis.

4.3.2 LGViT implementation

Structure

The LGViT implementation maintains a structure similar to the original Vision Transformer (ViT) model architecture known as 'ViT-B.16' (Dosovitskiy et al., 2021), retaining the same embedding, transformer block, and final MLP head components. Figure 9 shows the class diagram of the hierarchical organization of the model components. The implementation unifies both ViT and DeiT models under a single codebase, with naming conventions primarily reflecting the DeiT terminology. This unified approach facilitated experimentation with different backbone architectures for the authors.

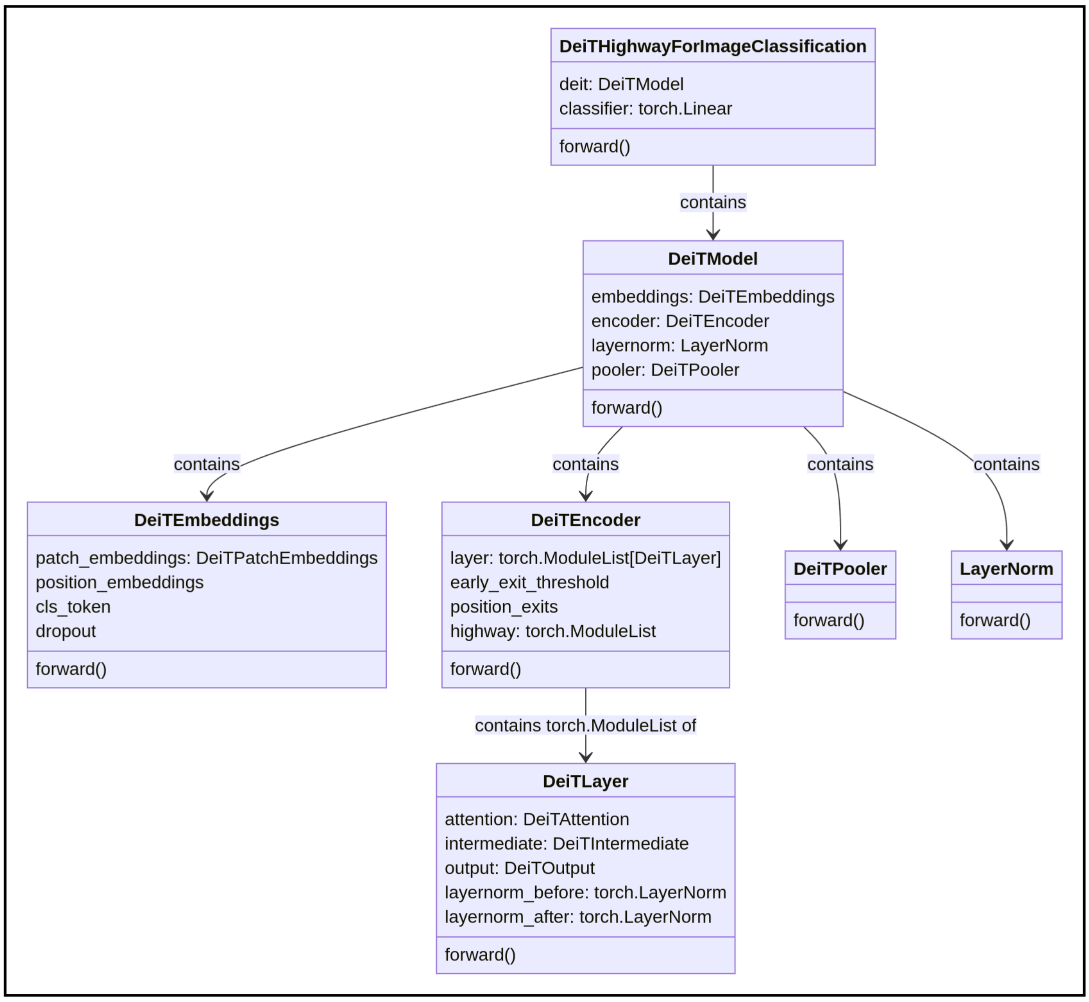

Figure 9: Class diagram for the LGViT model

The forward pass through the model follows a specific flow pattern, as depicted in Figure 10 and Figure 11. The first figure shows the high-level flow through the entire model, while the second illustrates the encoder's operation with early exit mechanisms. The implementation utilizes `try/catch` statements as the exit mechanism. When exit conditions were met, the determined `DeiTLayer` throws a `HighwayException`, which is then caught by the `DeiTEncoder`. This approach, while functional, created challenges for deployment since try/catch statements prevent the computation graph from being properly traced by PyTorch's ONNX export functionality.

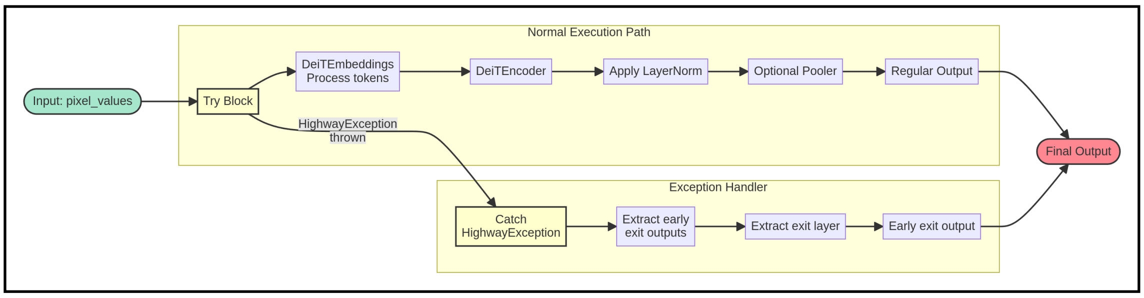

Figure 10: Overview of the forward pass of the DeiTModel from (Xu et al., 2023)

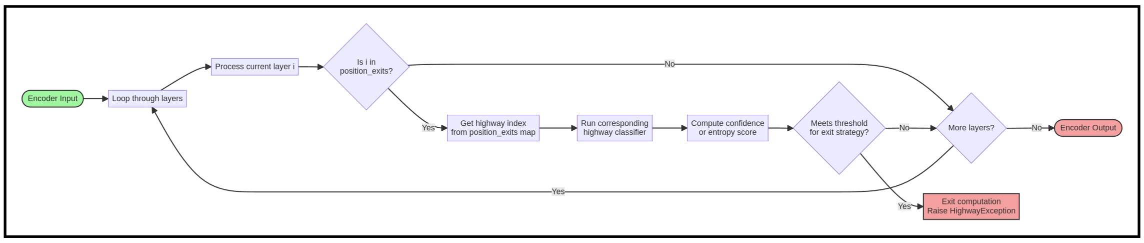
Encoder Input
Loop through layers
Process current layer i
Is i in position_exits?
No
Yes
Get highway index from position_exits map
Run corresponding highway classifier
Compute confidence or entropy score
Meets threshold for exit strategy?
No
More layers?
No
Encoder Output
Yes
Exit computation Raise HighwayException
Yes

Figure 11: Overview of the forward pass of the encoder component from (Xu et al., 2023)

Implementation problems

Working with the original LGViT codebase posed sever challenges. The code structure was based of the template of the models implemented in the 🤗 `Transformers`[7] (Wolf et al., 2020) model database, requiring both inheritance and composition from specific base classes.

Although useful for models in the 🤗 `Transformers` infrastructure, it led to significant code duplication throughout the repository, making it more difficult to analyze and understand the implementation. Additionally, the execution flow was primarily managed through bash scripts that passed configurations to high-level Python scripts, which obscured parameter settings and made replication challenging.

Certain design decisions further complicated understanding the codebase. For example, the retrieval of corresponding early exit heads by layer index and the frequent overwriting of output variables made the code flow difficult to follow.

A particularly complex example appears in the encoder's forward method (Listing 1), where the handling of early exits involves multiple nested conditions and state tracking. This particularly implementation detail, quite embedded in the code of LGViT, made the export to ONNX impossible, since the computation graph is broken when try/catch statements are in place.

[7] Yes, the official name includes the emoji[8]

[8] I find this annoying when reading, but that's their brand identity

```python
# Simplified exit strategy decision-making logic in the original implementation
for i, layer_module in enumerate(self.layer):
    # ... process current layer ...

    if i in self.position_exits:
        highway_exit = self.highway[self.position_exits[i]](current_outputs)

        # Inference stage exit decision logic
        if not self.training:
            highway_logits = highway_exit[0]

            # Entropy-based exit strategy
            if self.exit_strategy == 'entropy':
                highway_entropy = entropy(highway_logits)
                highway_exit = highway_exit + (highway_entropy,)
                all_highway_exits = all_highway_exits + (highway_exit,)

                if highway_entropy < self.early_exit_threshold[self.position_exits[i]]:
                    new_output = (highway_logits,) + current_outputs[1:] + ({"highway": all_highway_exits},)
                    raise HighwayException(new_output, i + 1)

            # Confidence-based exit strategy
            elif self.exit_strategy == 'confidence':
                highway_confidence = confidence(highway_logits)
                highway_exit = highway_exit + (highway_confidence,)
                all_highway_exits = all_highway_exits + (highway_exit,)

                if highway_confidence > self.early_exit_threshold[self.position_exits[i]]:
                    new_output = (highway_logits,) + current_outputs[1:] + ({"highway": all_highway_exits},)
                    raise HighwayException(new_output, i + 1)
            ...
```

Listing 1: “Code Snippet from the original LGViT implementation”

4.3.3 CIFAR-100 Dataset

The LGViT model was evaluated on three datasets in the original paper, with CIFAR-100 among them. The CIFAR-100 (Krizhevsky, 2009) dataset consists of 60,000 color images of size 32×32 pixels, divided into 50,000 training images and 10,000 testing images, with samples uniformly distributed across 100 distinct classes. Each class contains exactly 500 training images and 100 test images, providing a balanced dataset for classification tasks.

For their experiments, the LGViT authors applied standard data augmentation techniques to improve model generalization. The training data underwent random crops, random horizontal flips, and normalization, while the testing data was processed with center crops and normalization only. Both training and test samples were resized to 224×224 pixels

The CIFAR-100 dataset presents varying levels of visual complexity across its classes. Some classes exhibit highly consistent and distinctive visual features, as demonstrated by the keyboard samples in Figure 12. These classes typically present well-defined visual patterns that remain consistent across samples.

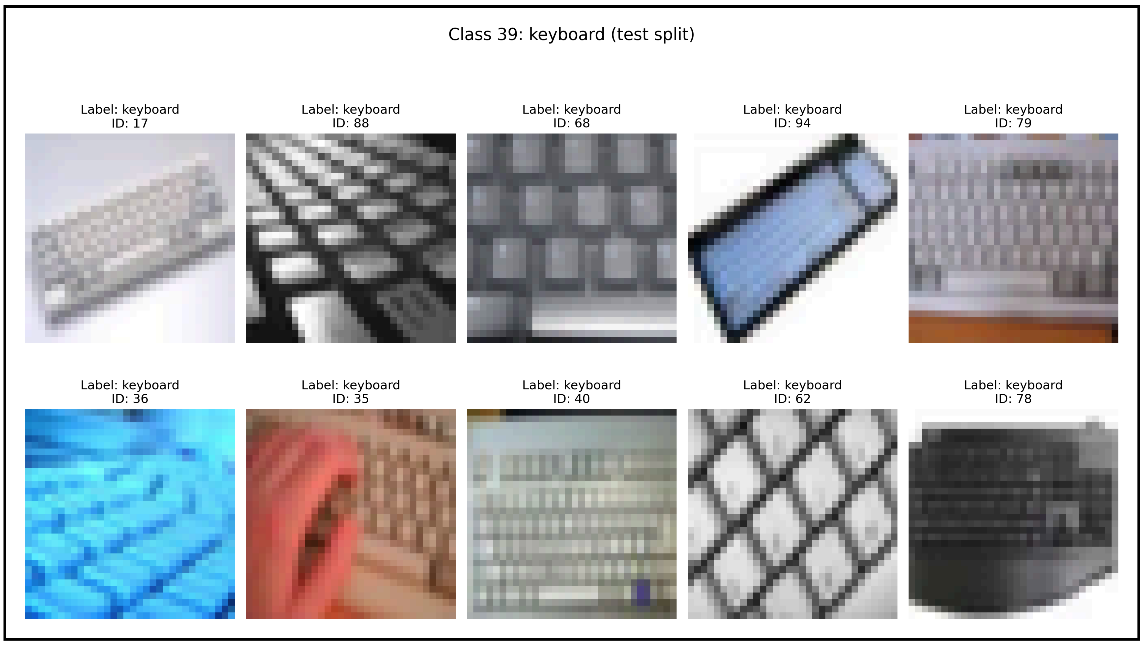

Figure 12: CIFAR-100 sampled from the 'keyboard' class.

Other classes share significant visual similarities. The examples in Figure 13 illustrate this challenge, where oak trees and willow trees share substantial structural similarities. Note how tree structures dominate most of the image area, making it difficult to distinguish between tree classes even for human observers without specialized knowledge.

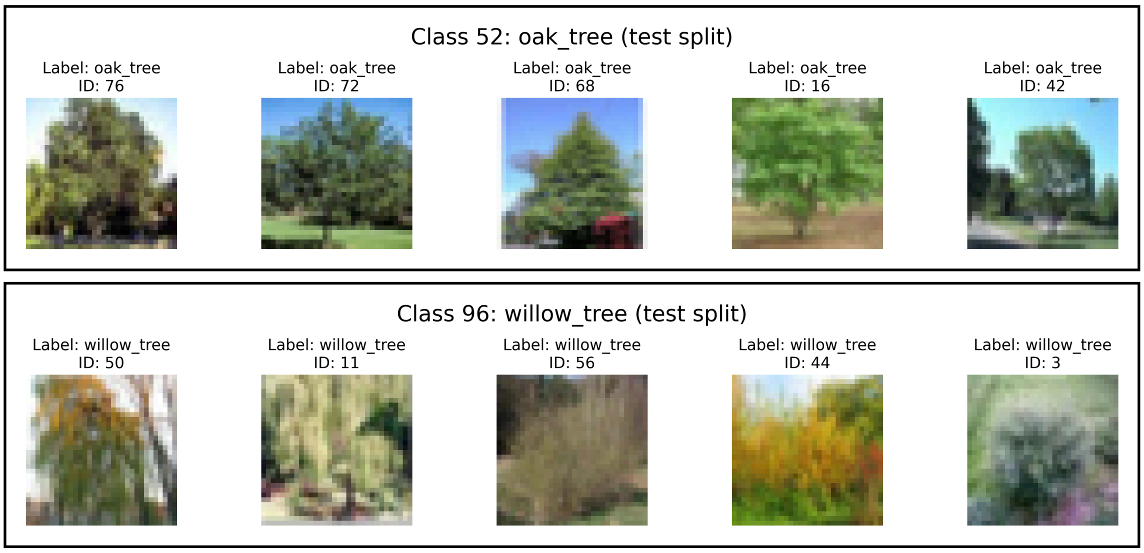
Class 52: oak_tree (test split)
Label: oak_tree ID: 76
Label: oak_tree ID: 72
Label: oak_tree ID: 68
Label: oak_tree ID: 16
Label: oak_tree ID: 42
Class 96: willow_tree (test split)
Label: willow_tree ID: 50
Label: willow_tree ID: 11
Label: willow_tree ID: 56
Label: willow_tree ID: 44
Label: willow_tree ID: 3

Figure 13: Sample CIFAR-100 images from different classes but similar visual features

Additionally, some classes display considerable intra-class variability. As can be appreciated in the class samples shown in Figure 14. These images vary significantly in pose, texture, background, and other attributes while belonging to the same semantic category.

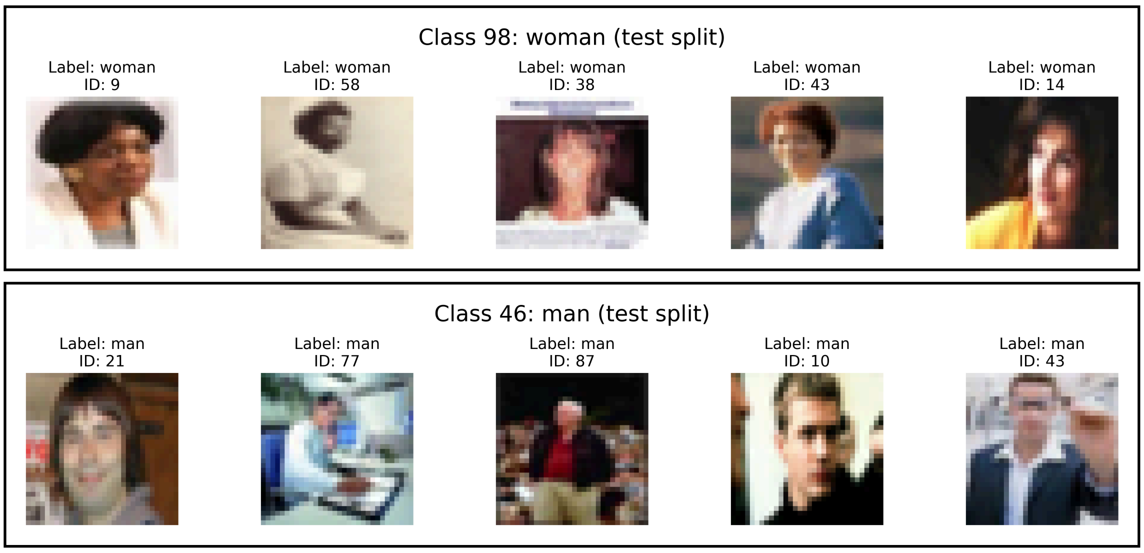

Figure 14: CIFAR-100 samples from the ‘woman’ and ‘man’ classes. Visual features both in the object of interest as well as the background are varied

5 Results

5.1 Terminology

The re-implementation of the LGVIiT done in this work is regarded hereinafter as *EEViT*[9]. When discussing its architecture, the following terminology is used:

- The names of the original ViT internal elements remain unchanged (patch embeddings, encoder, classifier).
- The encoder of the ViT is referred to as the "*Backbone*".
- Early exits are called "*Highways*" from the idea of taking a faster route to the destination.
- Within a highway, what the LGViT work calls "*Intermediate Heads*" (LPH or GAH) are referred here as the *neck* of the highway, which lead up to the highway's *head*.
- The classifier at the end of the highway is called the highway's *head*, following the common convention of naming the task-specific module (image classification in this case) as the "head" of the model.

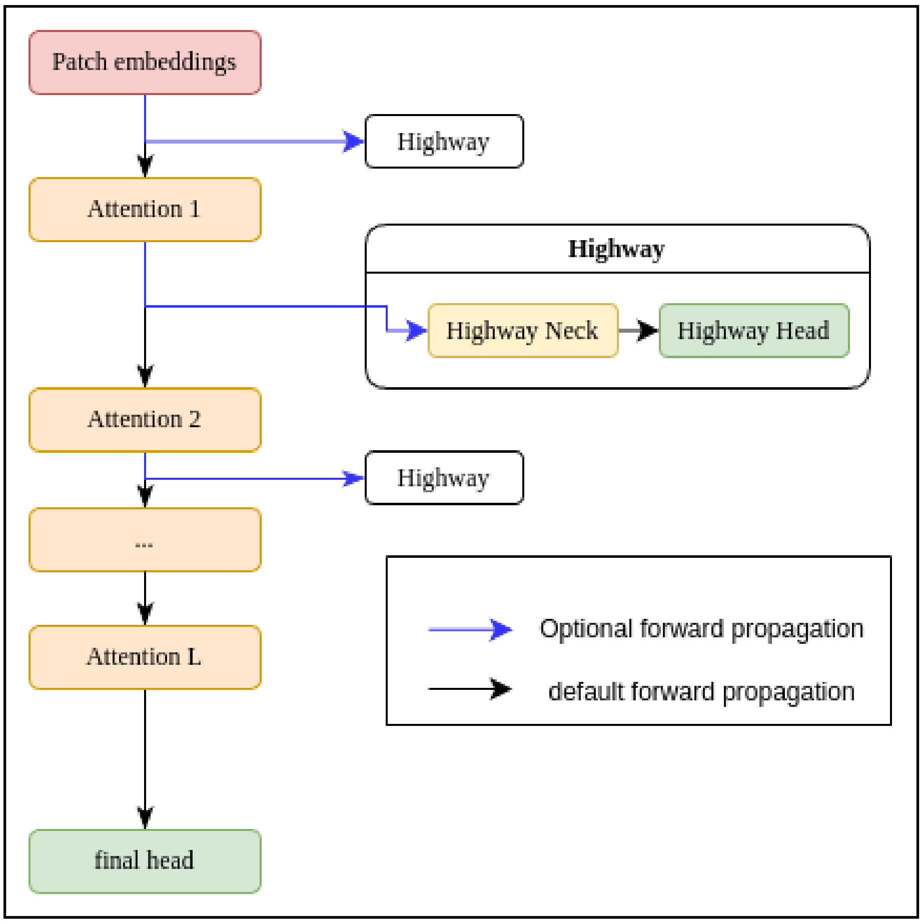

Figure 15: EEVIT's naming conventions

[9]The term EEVIT has different meanings in other publications, but here is the name given to the re-implementation of the LGViT codebase

5.2 Implementation results

5.2.1 Architecture

EEVIT follows the standard Vision Transformer (ViT) architecture. It takes as reference the implementation from the ViT-PyTorch repository (Wang, 2025) with modifications to support early exiting using one of the exit strategies detailed by the LGViT authors (Xu et al., 2023). Specifically, the *confidence* or *max-softmax* strategy. The reason for this is that the original authors published pre-trained weights trained on with strategy. The model's main components are presented hereunder, and the model's UML diagram is presented in the Annex section.

1. **Patch Embedding**: Converts the input image into a sequence of embeddings.
   - Projects image patches into the embedding space using a convolutional layer
   - Adds a learnable class token and positional embeddings
2. **Transformer Encoder**: Processes the embedded patches through multiple self-attention layers.
   - Contains multiple Attention layers, each with the potential for an early exit
   - Implements a "fast pass" mechanism to skip remaining layers once an exit condition is met
3. **Attention Blocks**: Standard self-attention mechanisms with potentially additional highway components.
4. **Highway Networks**: Early exit pathways attached to specific transformer layers.
   - Different types of highway necks (e.g., LPH, GAH)
   - Classifier to make predictions from intermediate features
   - Exit evaluator to determine if the confidence threshold is met
5. **Final Classifier**: The standard exit point for samples that don't meet early exit criteria.

It uses a hierarchical configuration system with 2 levels, Model and early-exit level. This allows flexibility in specifying different model variants while maintaining type safety. The configuration file allows specifying which highway type to use at each exit point:

```yaml
early_exit_config:
  exits:
    - [3, 'conv1_1']
    - [4, 'conv1_1']
    - [5, 'conv2_1']
    - [6, 'conv2_1']
    - [7, 'attention', {sr_ratio: 2}]
    - [8, 'attention', {sr_ratio: 2}]
    - [9, 'attention', {sr_ratio: 3}]
    - [10, 'attention', {sr_ratio: 3}]
```

Listing 2: excerpt from configuration file where the early exits are specified

Some design decisions were taken to simplify the implementation without altering the behavior of the model:

As mentioned in Section 4.3, LGViT specifies intermediate heads at modelling time[10]. Instead, in the new implementation, the network reads from a configuration file the location and type of early exit to be added to the network and appends it to the encoder layers. This allows for more flexibility in creation and placement of highways. The construction of the layers can be seen in version 1 of the Figure 16.

Initially, a conditional statement would be the mechanism to determine how to populate the fast-pass token as seen in version 1 of Figure 16, and although the export library supports tracing nested conditionals, it was decided to instead assign to the fast-pass token value, the output of the decision method. This can be done since they both the evaluation method's output and the fast-pass token have the same dimensionality, since it is an evaluation on each element of the attention score's output, the revision is presented in version 2 of Figure 16

The stacking of the attention and highway layer was further revised, since it made difficult the identification of the exit layer's index, since the list was shared by attention blocks and highways. Thus, the designed changed, making the highways an attribute of the attention block. Every attention block in the encoder will either get a exit head as specified or a generic `IdentityHighway`. The final design is version 3 of Figure 16.

At the end of the highway's forward pass, a classifier will evaluate the processed information and make a prediction, scored by a confidence level.

[10]The term *modelling time* refers to the time window in the execution of the program where the neural network is constructed. This is to make a distinction of the moment the model is used for inference form the moment it is constructed. Both happen within the traditionally used term *runtime*

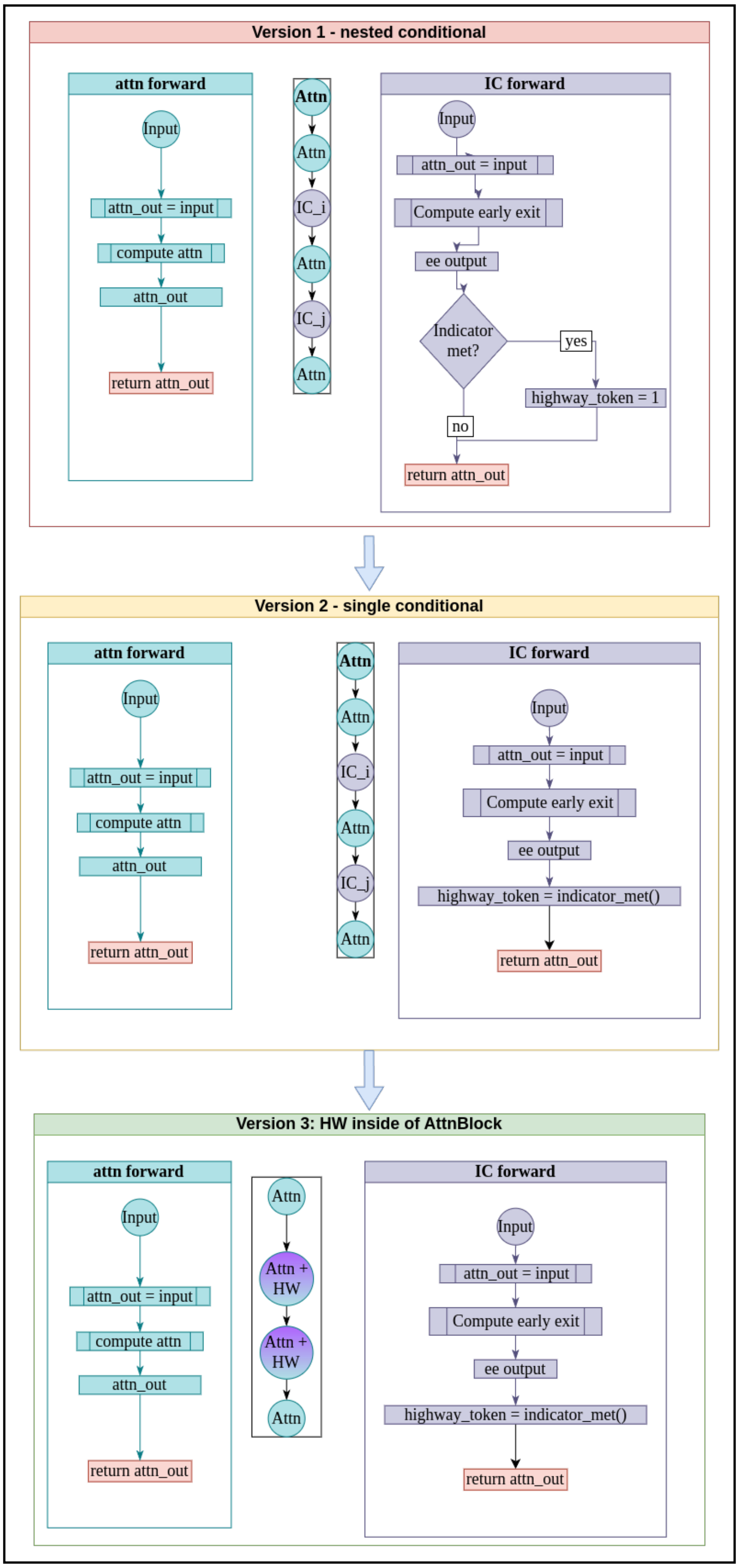

Figure 16: Encoder forward pass iterations

5.2.2 Early Exit Mechanism - Fast Pass Token

EEVIT implements a distinct mechanism to skip unnecessary computation. The exit mechanism design consists of adding a token to the patch sequence, termed the 'fast-pass' token. This token will act as signaling for the network to by-pass any further computations by reading it's values as truthy or falsy and acting accordingly. A diagram of the idea is presented in Figure 4.

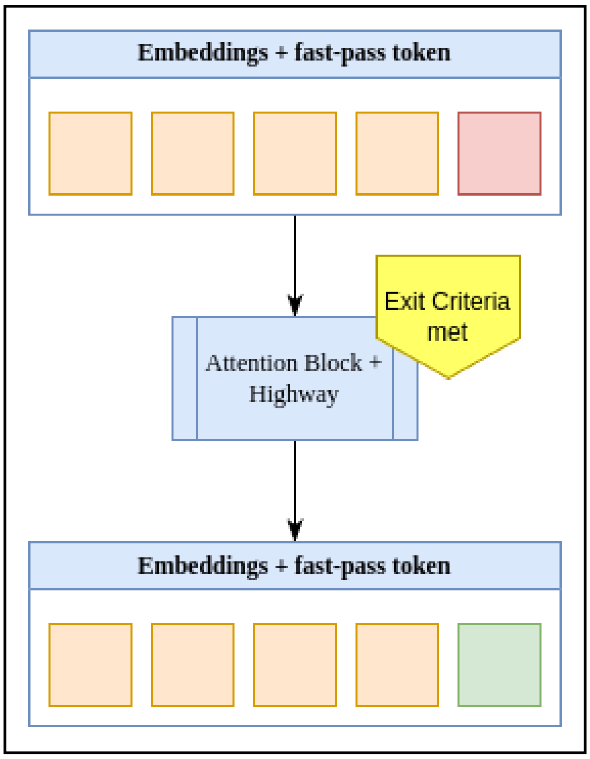
Embeddings + fast-pass token
Exit Criteria met
Attention Block + Highway
Embeddings + fast-pass token

Figure 17: Forward pass schematics of the encoder of the EEVIT model

With this mechanism the prediction from the early exit is preserved and passed through a fast-pass channel in the network up until the end. Figure 18 shows the internal mechanics of the attention block.

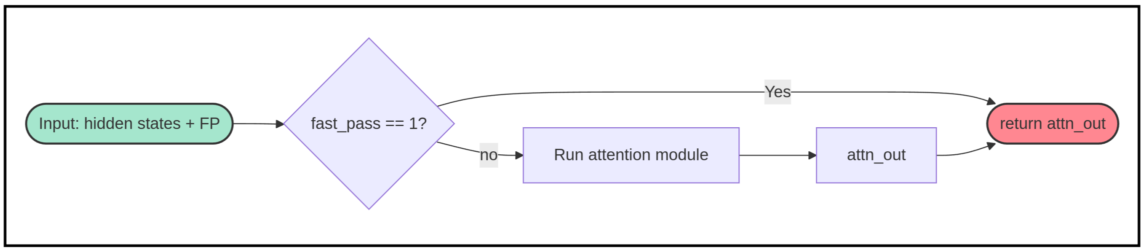

Figure 18: Internal logic of EEVIT's attention blocks

5.2.3 Highway Types

One very helpful feature of the new implementation, is the Highway factory. This is a feature that uses the factory method design pattern to create different types of `Highway` modules — The early exits — in the EEVIT architecture. The key components are:

- The `Highway` class: The main wrapper class that combines a highway's neck and head and exit evaluator — of classes `HighwayNeck`, `HighwayClassifier` and `ExitEvaluator` respectively —
- The `HighwayNeckFactory` class: A factory class that creates `HighwayNeck` network modules based on the specified type
- The `HighwayNeck` class: Different implementation classes for early exits, which are mapped to the highway type ni the following way:
  - LPH(s): `Conv1_x` with `x` corresponding to kernel size s
  - GAH(s): `attention(s)` with `s` corresponding the pooling window size s

This design offers several advantages. It provides modularity by using the factory pattern to separate the creation of highway networks from their usage, making it easy to add new types without altering the rest of the code. The configuration-driven approach allows highway networks to be created based on configuration values, enabling dynamic architecture selection without requiring code changes. Consistency is ensured as all highway heads follow the same interface, making them interchangeable and ensuring proper integration with the rest of the model. Additionally, reusability is achieved by sharing common components like the HighwayClassifier across different highway implementations. Finally, the declarative configuration enables early exits to be defined in the YAML configuration, simplifying experimentation with different early exit setups.

Implementing new Highways is now made significantly easier. The neck can be modelled as a normal PyTorch `Module`, with the following two requirements:

- It constructor function signature must have as arguments: `ee_config: EarlyExitsConfig, kwargs: dict`
- Its `forward` function signature must have as arguments `x: torch.Tensor, H: int, W: int)`

Then, when the new early exit model is ready to be included in the model, it can be added to the configuration:

```yaml
// yaml
early_exit_config:
  exits:
    - [5, 'my_new_head', {param1: value1, param2: value2}]
    - ...
```

5.2.4 Parameter Count and Distribution

Total parameter count is presented in Table 4. The parameter count of the implemented models encoder by layer is presented in Figure 19. As can be appreciated, the increase in parameters from layer without early exits to one with is about 30% or 2M parameters. As for the difference between Highway types, the parameter count doubles from LPH to GAH, as seen in Figure 20.

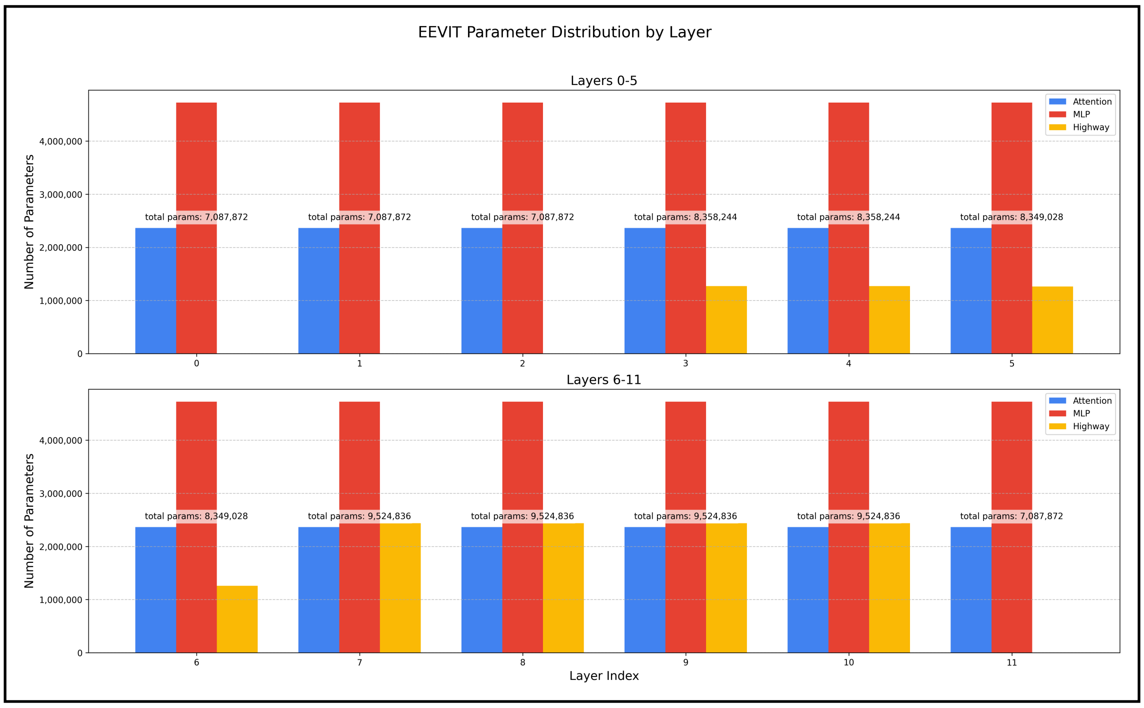

Figure 19: parameter count for the EEVIT encoder by layer

Component	**Parameter Count**	**Percentage**
Patch embeddings	742,656	0.74 %
Encoder	85,056,000	84.48 %
Early Exits	14,810,912	14.71 %
Final Classifier	76,900	0.08 %

Table 4: Parameter count and distribution for LGViT model, taken from own re-implementation

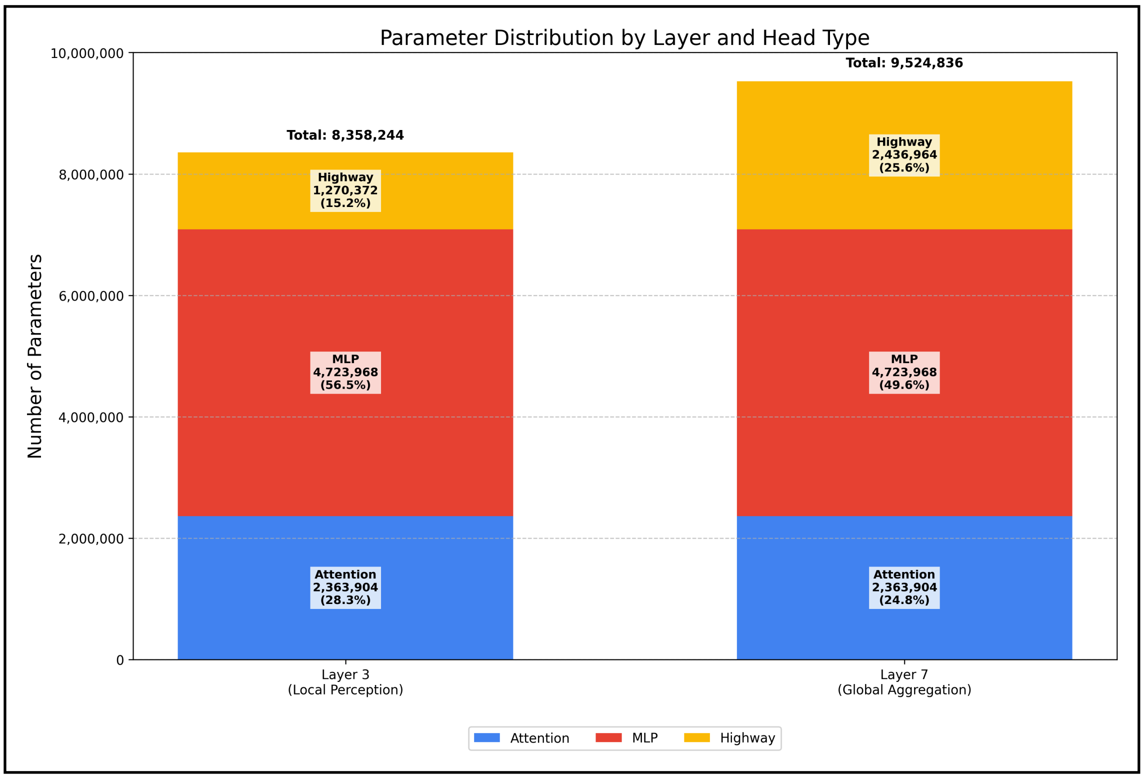

Figure 20: parameter count for encoder layer with Local Perception and global Aggregation necks

5.2.5 ONNX Export

The design presented in the last section was done taking into consideration that the model itself, although built using the PyTorch framework, needed to be exportable to ONNX. As can be seen in Figure 18, some form of flow-control is needed to guide the execution of the relevant nodes of the computation graph that is the model.

This flow-control need is the key reason behind the need for re-write. As mentioned in Section 3.2.1, it is not enough to replace the try/catch statements with an if/else statement for the exit decision, since the base python version brakes the computational graph of the model and is therefore untraceable.

Both the original implementation as well as this work's implementation are developed with PyTorch, and therefore the tool to create the ONNX model is `torch.onnx.export`. Thus, the star of the show in the new EEVIT implementation is `torch.cond`. It is important to highlight its use since without it it would have been impossible to test the model in the intended environment.

To illustrate how the feature was used, a simplified version of the encoder's forward pass logic, were the fast-pass mechanism is used is presented hereunder:

```python
1   """TransformerEncoder.forward method"""
2   def forward(self, x: torch.Tensor):
3     # code here skipped
4     for layer in self.layers:
5         fast_pass_layer = get_fast_pass(x_with_fastpass)
6
7         # torch.cond usage
8         x_with_fastpass, predictions_placeholder_tensor = torch.cond(
9             fast_pass_layer.any(),
10            self.fast_pass,
11            layer,
12            (x_with_fastpass, predictions_placeholder_tensor),
13        )
14        i += 1
15    fp = get_fast_pass(x_with_fastpass)
16    x_norm = self.norm_post_layers(remove_fast_pass(x_with_fastpass))
17    x_with_fastpass = set_fast_pass_token(add_fast_pass(x_norm), fp)
18
19    return x_with_fastpass, predictions_placeholder_tensor
```

Listing 3: Forward pass method of the `EEVIT` encoder

Although officially the support for torch.cond in torch.onnx.export was only release late january 25′, the feature luckily available beforehand in pre-released versions available already in Dec 24′, and the version used was `2.6.0.dev20241226`.

A final details equally important, is the fact that some data copying needed to be introduced, namely in two places. The code extract shown in Listing 3 shows in line 10 the conditional call to `self.fast_pass`. The code of this function in presented hereunder:

```python
def fast_pass(
        self,
        x_with_fastpass: torch.Tensor,
        predictions_placeholder_tensor: torch.Tensor,
    ):
        return x_with_fastpass.clone(), predictions_placeholder_tensor.clone()
```

Listing 4: fast pass method of the `EEVIT` encoder

Similarly, the highway module, in charge of setting the value of the fast-pass token carries the following logic in its forward pass method:

```python
"""TransformerEncoder.<layer_x>.highway.forward method"""
def forward(self, x_with_fastpass: torch.Tensor,
predictions_placeholder_tensor: torch.Tensor):

  # highway logic skipped for clarity

  x_with_fastpass = set_fast_pass_token(
      x_with_fastpass,
      value=self.exit_evaluator.decision_tensor(logits).to(
          dtype=x_with_fastpass.dtype
      ),
  )

  return x_with_fastpass, predictions_with_idx

# fast_pass_utils.py
def set_fast_pass_token(x_with_fastpass: torch.Tensor, value: float)
-> torch.Tensor:
  output = x_with_fastpass.clone()
  output[:, -1, :] = value
  return output
```

Listing 5: Forward pass method of the `Highway` class

We see the call to `set_fast_pass_token()`, and the data copy in its its implementation. While seemingly inefficient, this approach is required for export to ONNX. The compilation method needed for ONNX conversion does not support in-place value assignments — Also known as *aliasing* — (e.g., `x_with_fastpass[:, -1, :] = value` in Python), as such operations create issues during computation graph compilation.

Results from a test run with the same configuration as the reported by LGViT wa run and it's results are presented later on in Section 5.3.2 as part of the benchmark studies.

5.3 Benchmarking

5.3.1 Latency Profiling Study

To gain insight into the computational cost of the EEVIT model during inference, I conducted a comprehensive profiling study of the model, focusing on the operations in the encoder. The `torch.profiler` toolset from PyTorch provides detailed performance metrics, allowing us to identify computational bottlenecks and understand the latency contributions of different model components.

Methodology

The profiling was conducted using the following approach:

- The analysis focused on three distinct layer groups within the model:
  - Group 0 (No early-exit): Layers without early exits (layers 0-2, and 11)
  - Group 1 (LPH): Layers with Local Perception Head exits (layers 3-6)
  - Group 2 (GAH): Layers with Global Aggregation Head exits (layers 7-10)
- To capture the computational cost of all layers, the early exit mechanism was disabled, i.e. the computations were carried out but the decision to exit early was not taken.
- Model execution was captured using PyTorch's `Profile` context manager API, which records operator execution times, input shapes, stack traces, and device kernel activity
- A warmup phase with 200 iterations using randomized data was implemented to ensure stable performance measurements.
- Both CPU and GPU implementations were profiled to compare performance across devices.
- Due to the large size of profiling data, statistical analysis was performed on a random sample of 50 inference runs.

Initial Findings and Challenges

My initial profiling , done for the CPU case, gave results that presented some unexpected patterns, as shown in Table 5. Surprisingly, layer group 0 (without early exits) appeared to be slower than group 2 (with GAH exits), which contradicted our understanding of the model architecture since early exit layers contain additional computation.

Layer Group	**Avg (ms)**	**Std Dev**	**Min (ms)**	**Max (ms)**
No early-exits	17.324	16.728	11.183	165.115
LPH	18.784	5.925	13.966	75.426
GAH	16.865	3.591	13.057	28.778

Table 5: Initial statistics for the layer groups

To investigate this anomaly, I expanded the profiling to analyze each layer individually, as shown in Table 6. This revealed that layers 0 and 1 were significantly slower than other layers and exhibited much higher variability in execution time.

Layer	Avg (ms)	Std Dev	Min (ms)	Max (ms)
0	20.357	26.406	11.296	165.115
1	19.827	19.545	11.183	100.164
2	14.848	4.011	11.315	25.263
3	19.900	9.155	14.231	75.426
4	18.853	4.540	14.296	27.953
5	18.401	4.356	14.048	31.455
6	17.981	4.130	13.966	29.400
7	17.169	4.023	13.057	28.778
8	17.010	3.289	13.750	23.959
9	16.695	3.502	13.260	24.877
10	16.587	3.586	13.152	24.996
11	14.262	3.083	11.721	21.877

Table 6: Initial statistics for the all attention layers

Further investigation of the execution timeline for layers 0, 1, and 2 (shown in Figure 21) indicated that the initial runs showed greater variability, suggesting insufficient warmup.

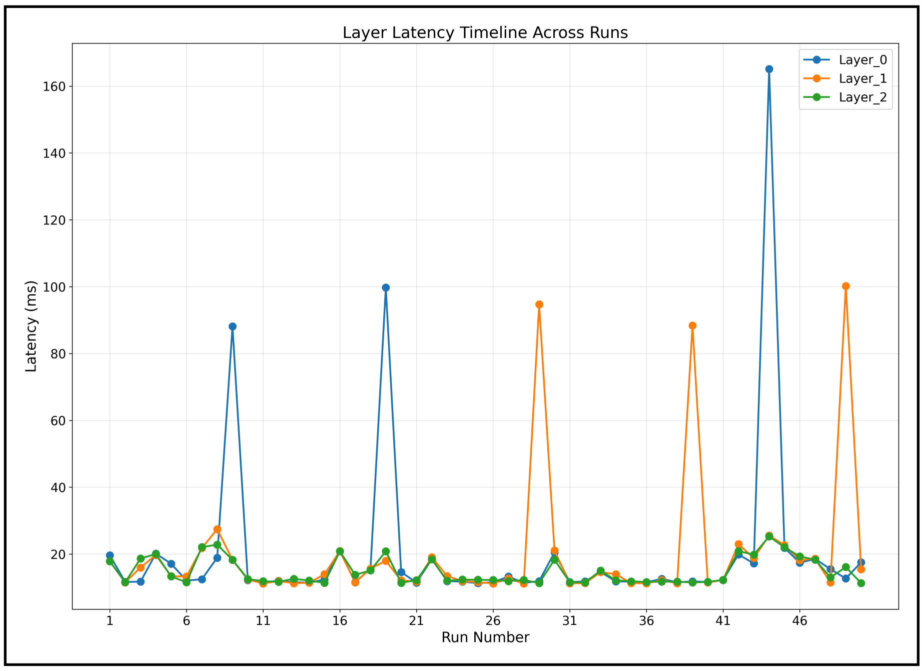

Figure 21: Latency for layers 0,1 and 2 across all profiling runs. 50 warm-up iterations.

Improved Methodology and Results

To address these issues, I increased the warmup phase from 20 to 100 iterations. This significantly improved the stability of the measurements, as shown in Figure 22. I then decided to increase the iteration number to 200, and again the stability improved, albeit slightly less significant, as can be seen in Figure 23. With the improved methodology, not only the timeline for layers 0, 1, and 2 showed more consistent behavior, but less high latency outliers run occur, as can be appreciated from contrasting Figure 22 and Figure 23. I decided then to set the iterations at 200.

Layer Group	Avg (ms)	Std Dev	Min (ms)	Max (ms)
No early-exit	14.777	11.378	11.148	98.959
LPH	16.763	4.921	13.944	77.302
GAH	15.609	2.424	13.614	23.913

Table 7: Layer groups profiling with **200** warmup iterations. We see the no early-exits layer group behaving more stable and slightly faster than the rest

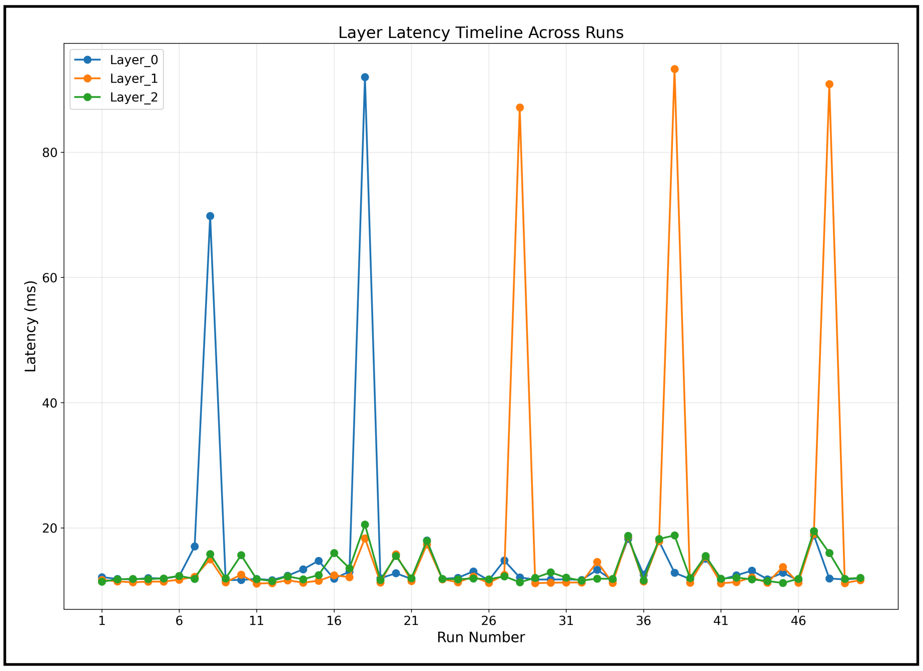

Figure 22: Latency after **100** warm up iterations, for layers 0,1 and 2 across all profiling runs

Why did this work?

The improvement in measurement stability following the increase in warmup iterations can be attributed to several factors:

Parallelism and multithreading:

PyTorch internal operations library `ATen` implements multithreading operations with the OpenMP library, and uses all the available CPU cores. This can lead to *oversubscription* to the memory caches, which The OS scheduler may take time to allocate threads optimally across CPU cores.(Team, 2025b)[11]

[11]The default library is OpenMP and the default number of threads is the number of CPU cores (Team, 2025b)

Cache Warming:
Local memory caches need time to be populated with frequently accessed data: The first few iterations may load data and instructions into CPU caches (L1, L2, L3). As the model runs longer, more computations fit into caches, reducing access to slower RAM.

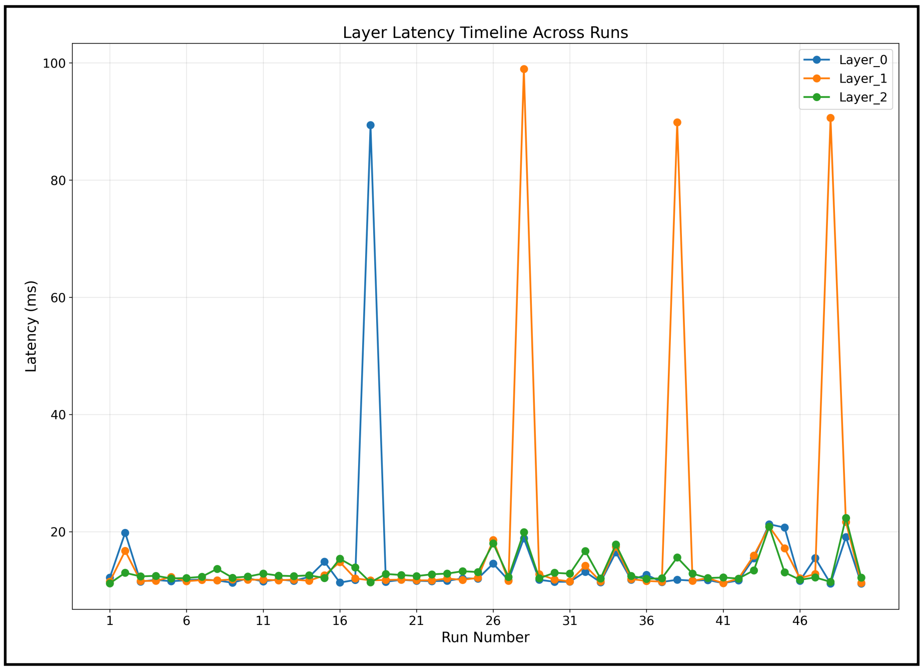

Figure 23: Latency after **200** warm up iterations, for layers 0,1 and 2 across all profiling runs

GPU Performance Analysis

After establishing a reliable profiling methodology, I conducted similar measurements on GPU hardware. The results are presented in Table 8.

Layer Group	Avg (ms)	Std Dev	Min (ms)	Max (ms)	Count
No early-exit	2.775	9.931	1.092	72.840	200
LPH	5.027	1.324	4.625	23.045	200
GAH	4.626	0.247	4.425	6.012	200

Table 8: Attention Latency by Layer Group. **GPU** case

Figure 24 shows the latency for representative layers from each group, highlighting the consistent pattern of increased latency in layers with early exits compared to those without.

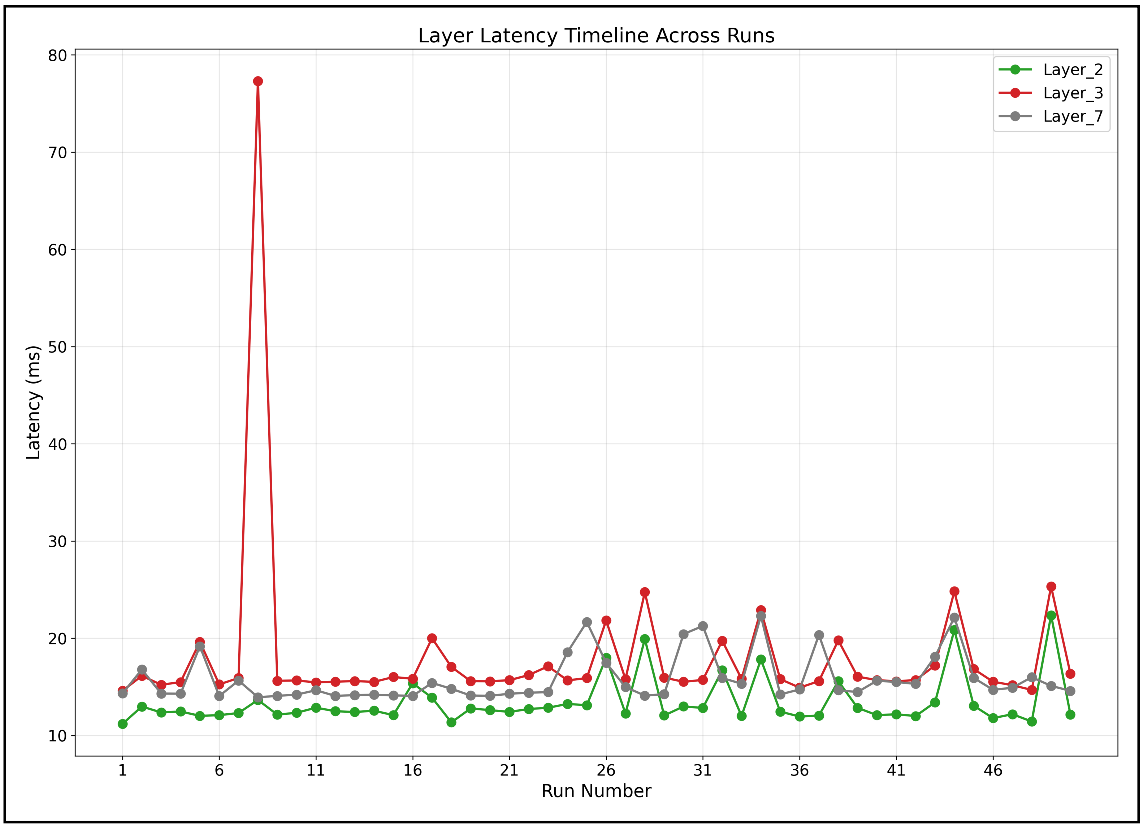

Figure 24: Latency for layers 2, 3, and 7 on GPU across all profiling runs

Key Findings

With the stabilized profiling methodology, we can now answer important questions about the computational cost of different components:

1. What is the latency of a standard attention block without early exits?
2. How much additional latency do different types of early exit mechanisms (LPH and GAH) introduce?

Table 9 summarizes these findings across both CPU and GPU implementations.

Device	**No early-exits [ms]**	**LPH Layer [ms]**	**GAH Layer [ms]**
CPU	14.777 ± 11.378	16.763 ± 4.921 (13%)	15.609 ± 2.424 (6%)
GPU	2.775 ± 9.931	5.027 ± 1.324 (81%)	4.626 ± 0.247 (67%)

Table 9: Latency comparison between layer types.
In parenthesis is the increment percentage w.r.t no early-exit layers

A striking observation is that the relative overhead of early exits differs dramatically between platforms—much higher on GPU (67-81%) compared to CPU (6-12%). This platform-dependent behavior warranted deeper investigation into the underlying causes.

For CPU execution, profiling visualization in Figure 25 reveal that the early exit computation time is comparable to standard self-attention operations. The segment labeled `eevit/vit_classes.py(171)` inside of the red square corresponds to the computation of the `Q, K`, and `V` matrices, while `AttentionMLPs` represents the second part of the attention block. In this environment, the Highway module accounts for less than a quarter of the total execution time, explaining the relatively modest overhead.

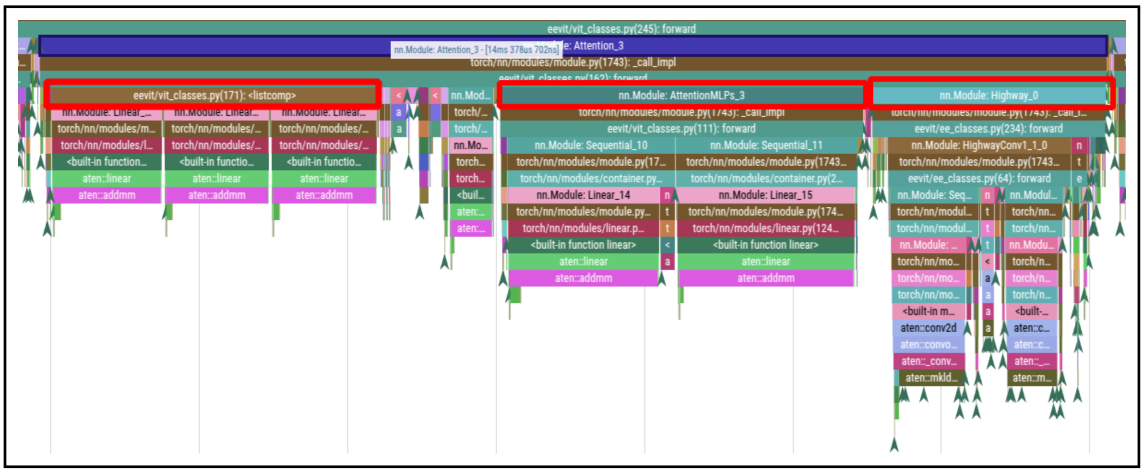
eevit/vit_classes.py(245): forward
nn.Module: Attention_3
torch/nn/modules/module.py(1743): _call_impl
eevit/vit_classes.py(171): <listcomp>
nn.Module: AttentionMLPs_3
nn.Module: Highway_0
aten::linear
aten::addmm
aten::conv2d

Figure 25: Profiling visualization for **CPU** execution of attention layer 3

GPU execution presents a different profile. For layers without early exits (layers 0 and 1 in Figure 26), we observe a significant latency bottleneck in the `cudaStreamSynchronize` operator within the `aten::is_nonzero operation`. In the example shown, approximately 4.1 ms is spent in `attention_0.forward` plus `aten::is_nonzero`, of which only 1.3 ms represents relevant computational logic.

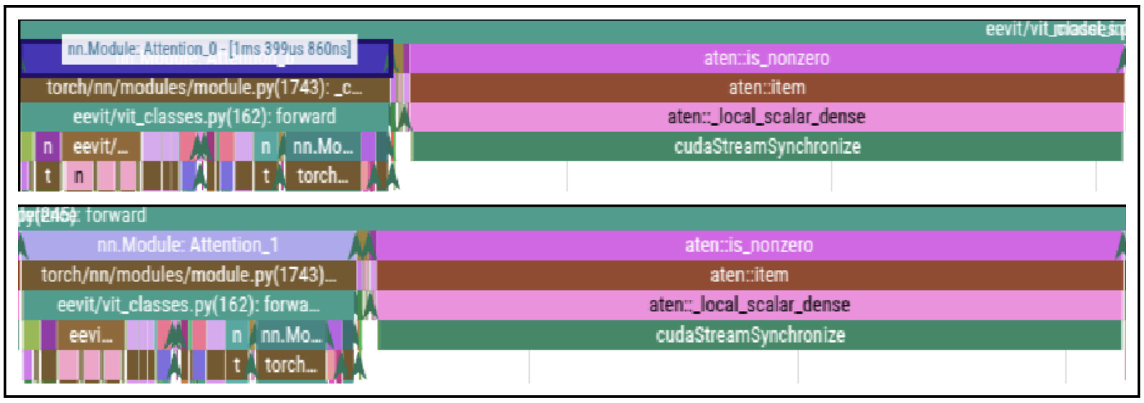
nn.Module: Attention_0 - [1ms 399us 860ns]
torch/nn/modules/module.py(1743): _c...
eevit/vit_classes.py(162): forward
aten::is_nonzero
aten::item
aten::_local_scalar_dense
cudaStreamSynchronize
nn.Module: Attention_1
torch/nn/modules/module.py(1743)...
eevit/vit_classes.py(162): forwa...

Figure 26: Profiling visualization for **GPU** execution of layers 0 and 1

This inefficiency stems from the fast-pass implementation as shown in Listing 3. The conditional statement on lines 8-13 check if the fast-pass token has a truthy value to determine whether to compute the next layer or skip further processing. This implementation forces a CUDA synchronization, as evaluating the tensor's truth value requires transferring data from GPU to CPU, creating a performance bottleneck. Interestingly, this synchronization overhead is substantially lower in layers with early exits, as demonstrated in Figure 27, though the exact reason for this difference remains unclear.

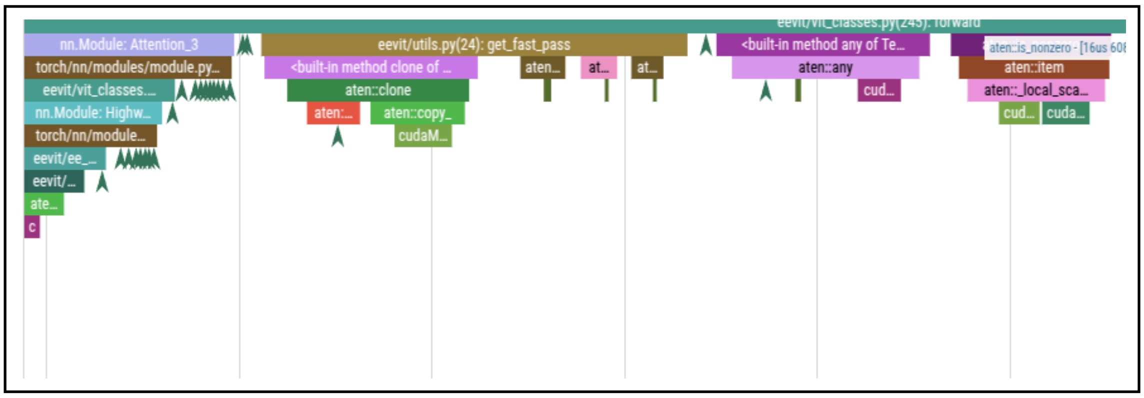

Figure 27: Profiling visualization for **GPU** execution of early exit layer3. The latency of its `aten:is_nonzero` is less than 0.02 ms

For layers with early exits, the dominant performance factor is the `aten::copy_` operation (Figure 28), which consumes more than half of the layer's execution time. When profiling without stack tracing—capturing only `aten` operations—computation time drops to around 1 ms, but the `aten::copy_` operation still requires approximately 2 ms (Figure 29).

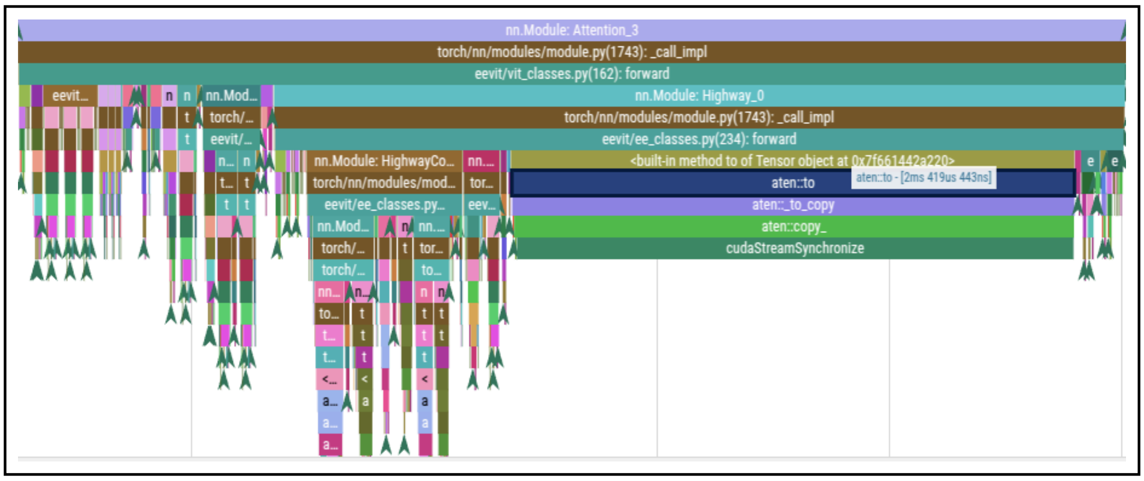
nn.Module: Attention_3
torch/nn/modules/module.py(1743): _call_impl
eevit/vit_classes.py(162): forward
nn.Module: Highway_0
torch/nn/modules/module.py(1743): _call_impl
eevit/ee_classes.py(234): forward
nn.Module: HighwayCo...
torch/nn/modules/mod...
eevit/ee_classes.py...
<built-in method to of Tensor object at 0x7f661442a220>
aten::to
aten::to - [2ms 419us 443ns]
aten::_to_copy
aten::copy_
cudaStreamSynchronize

Figure 28: Complete **GPU** execution of early exit layer3. Visualizing also the call stack

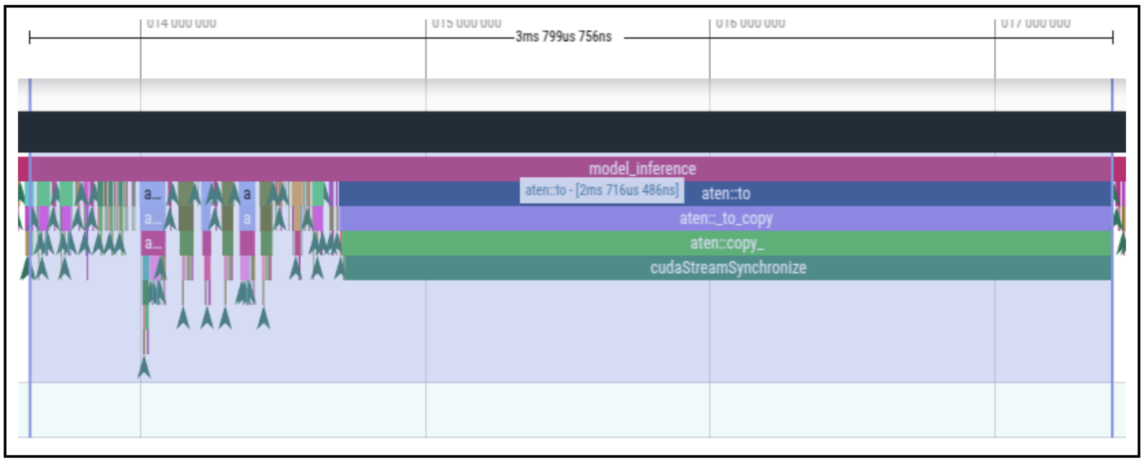

Figure 29: Complete **GPU** execution of early exit layer3

This overhead is directly attributable to the fast-pass token implementation in the Highway class (Listing 5), whose drawback and benefits are mentioned in Section 5.2.5

5.3.2 Threshold Study

To establish a baseline for my implementation and evaluate the impact of early exit configurations, I conducted a comprehensive threshold study based on the original LGViT model. The authors of LGViT configured their model with a confidence threshold of 80% for the highway classifiers, which serves as our reference point. To validate my implementation and understand the effects of early exits, I replicated the benchmark experiment on the CIFAR100 dataset's training set under identical conditions, particularly maintaining the 80% confidence threshold for highway classifiers. The speedup is computed using the same formula reported in the LGViT work:

$$\text{Speed-up} = \frac{\sum_{i=1}^{L} L \times m^i}{\sum_{i=1}^{L} i \times m^i}$$

Where L represents the total number of layers and m^i denotes the number of samples exiting at layer i. This formula effectively compares the computational cost of processing all samples through the entire network versus the reduced cost achieved through early exits.

To have a reference baseline for the coming experiments, two reference base experiments were ran, and their results are presented in Table 10, which shows a comparison between a model with no early exits and the original LGViT configuration. Detailed information about the exit distribution, accuracy, and average speed per exit for the 80% threshold is shown in Figure 30.

Configuration	**Accuracy Avg. [%]**	**Speed-Up**
No Early-Exits	90.24	x1.0
Original LGViT configuration	87.31%	x2.08

Table 10: baseline results for measurement of accuracy and latency for the CIFAR100 dataset

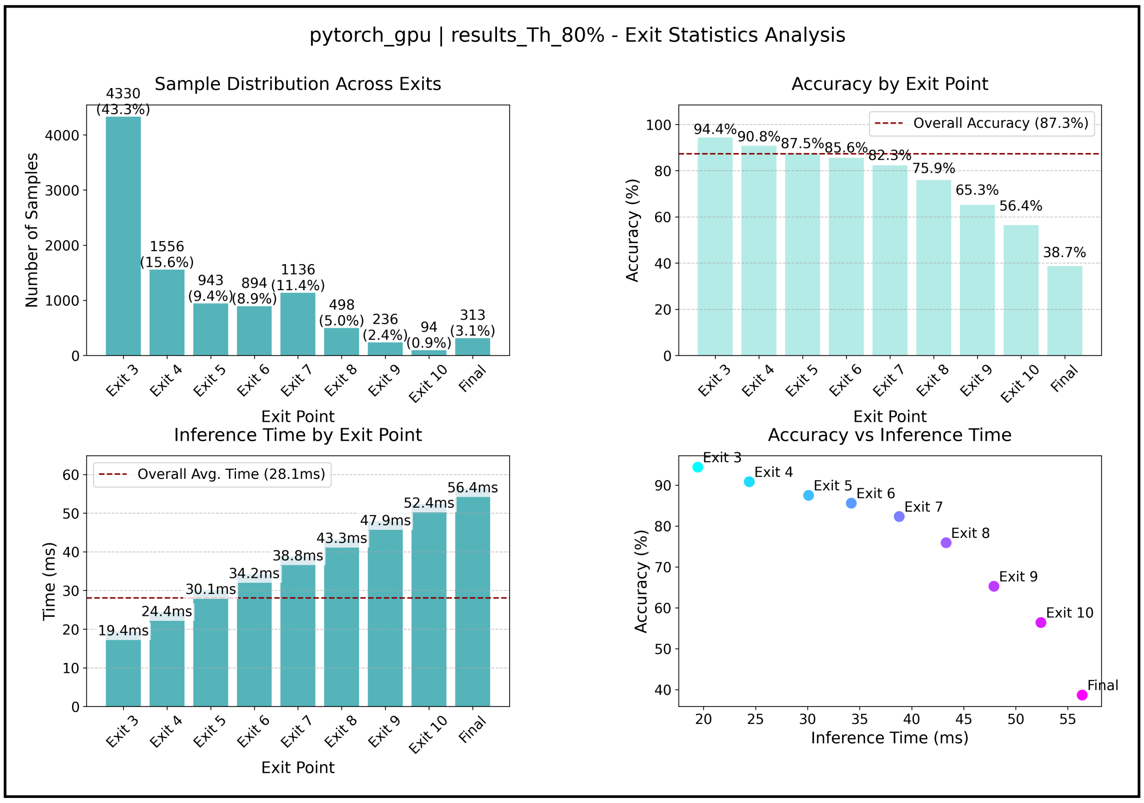

Figure 30: exits statistics with confidence threshold of 80%

The hypothesis that the model was optimized for the specific 80% confidence threshold during training was considered. Thus, to verify whether altering this threshold without retraining would incur costs in accuracy, latency, or both, additional experiments were conducted with elevated thresholds of 90% and 99%. The exit statistical results for the latter one are presented in Figure 32

As shown in Figure 31, increasing the confidence threshold pushes samples to exit at deeper network layers. At 80% threshold, layer 3 processes most samples, becoming the most computation-saving layer. At 90%, exits become more evenly distributed across layers, while at 99%, layer 7 becomes the predominant exit point.

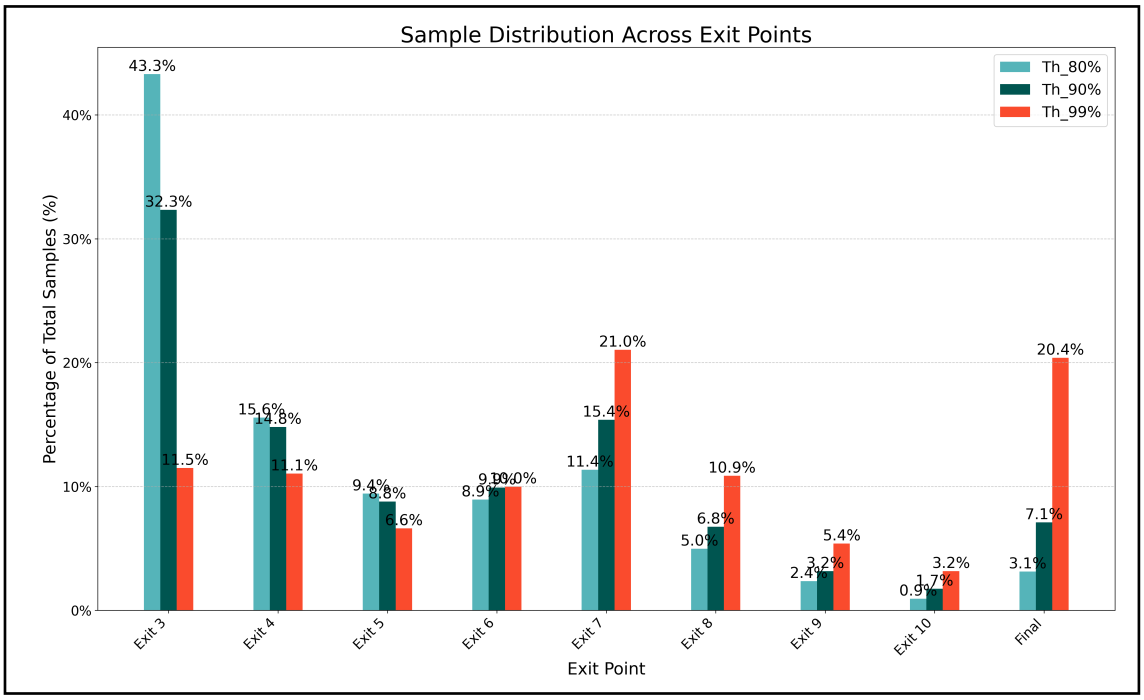

Figure 31: sample distribution for thresholds 0.8, 0.9 and 0.99

Threshold	**Accuracy [%]**	**speed-up**
0.80 (LGViT config)	87.31%	x2.08
0.9	89.41%	x1.86
0.99	90.36%	x1.48

Table 11: Changes in accuracy and speed-up w.r.t the original LGViT configuration

From Figure 33, we can observe that accuracy increases proportionally with threshold values at each exit point, while latency remains relatively unchanged, as expected. The overall average accuracy and speedup for each threshold configuration is summarized in Table 11. The results reveal an interesting trade-off: higher confidence thresholds improve accuracy but reduce computational efficiency. The 99% threshold nearly matches the accuracy of the non-early-exit model (90.36% vs. 90.24%) while still providing a significant x1.48 speedup. Meanwhile, the original 80% threshold configuration offers the highest speedup (x2.08) but at the cost of approximately 3% lower accuracy compared to the baseline.

Interestingly, my implementation achieved a speedup of x2.08, which is slightly higher than the x1.87 reported in the original LGViT paper. This discrepancy persisted even when running benchmarks with their implementation.

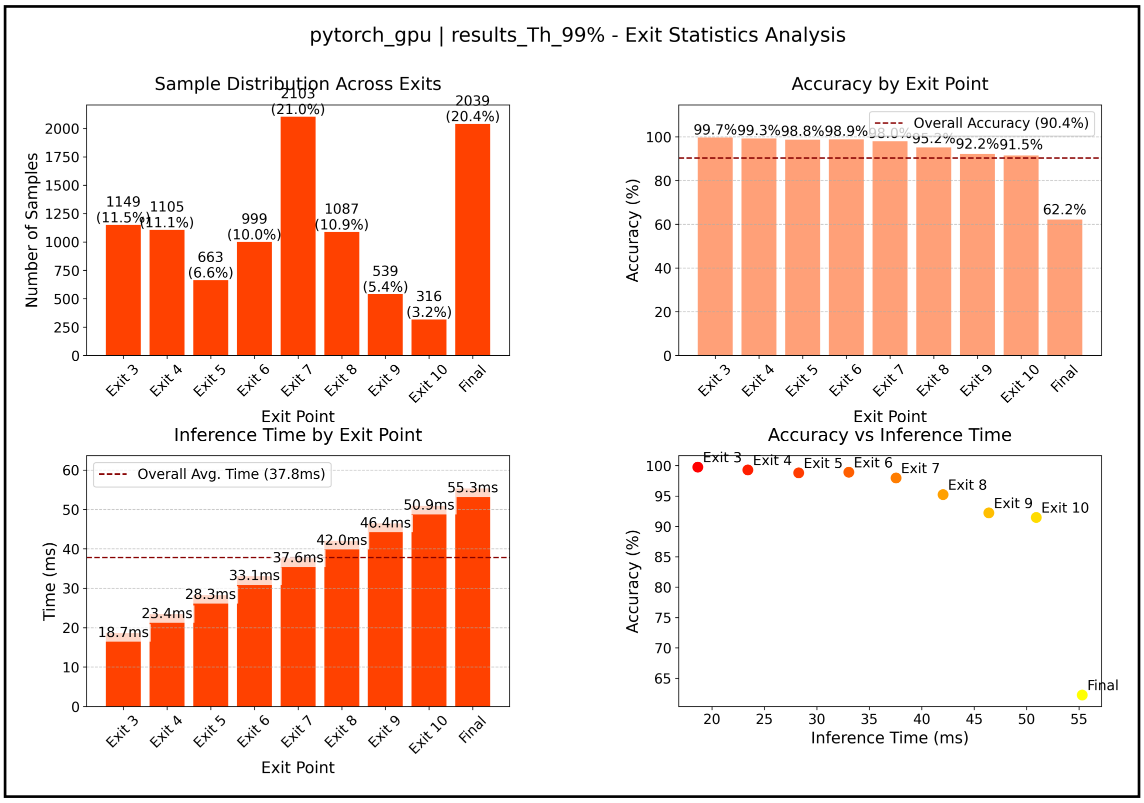

Figure 32: exits statistics with confidence threshold of 90%

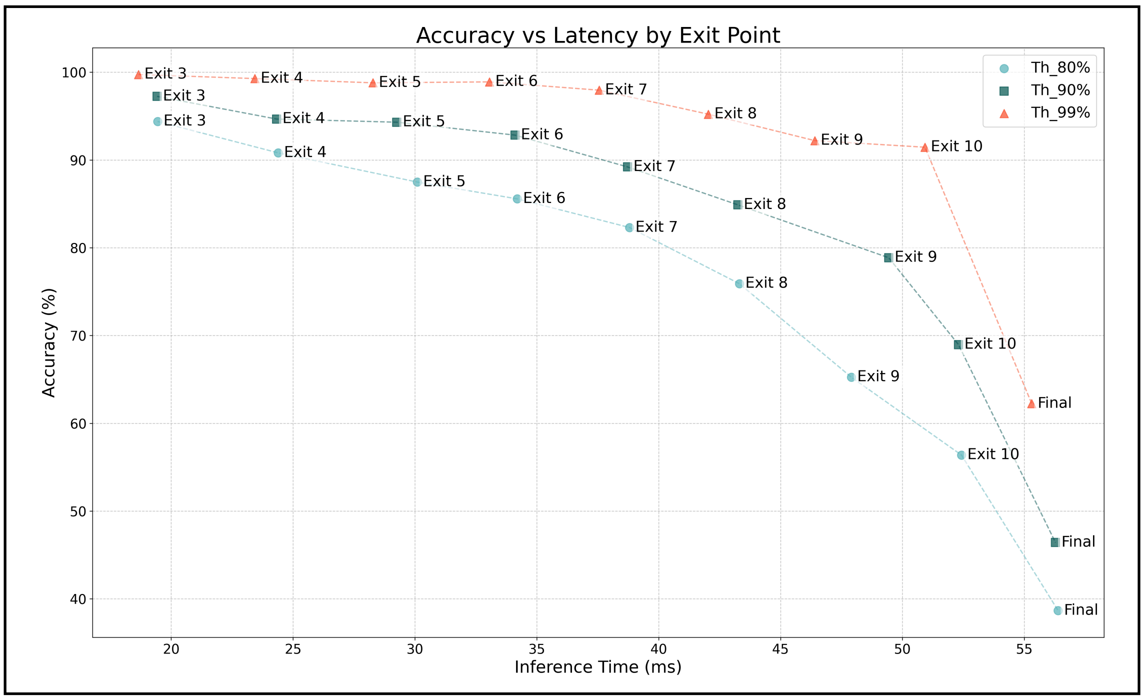

Figure 33: Accuracy vs Latency comparison for thresholds 0.8, 0.9 and 0.99

5.3.3 Platforms Performance Benchmark (working title)

To evaluate the performance of my model under production-like environment conditions, tests were conducted using the ONNX model only, on both a standard PC and a Jetson AGX Orin, which acts as the edge computing platform.

The PC testing environment featured an Intel Core i7-9750H CPU (6 cores, 12 threads) with 12 MB of L3 cache, along with an NVIDIA GeForce GTX 1650 GPU with 4GB of VRAM. For the edge computing environment, the NVIDIA Jetson AGX Orin was used, which is equipped with a 12-core Arm Cortex-A78AE CPU with 6MB of L3 cache and 2048-core NVIDIA Ampere architecture GPU. The benchmark methodology used is the same as in Section 5.3.2, namely using the test set of CIFAR100.

The results, presented in Figure 34, reveal several interesting patterns. Firstly, the accuracy values were consistently maintained across all platforms for each exit point and for the overall model performance, confirming that the model's behavior was preserved by the ONNX export faithfully.

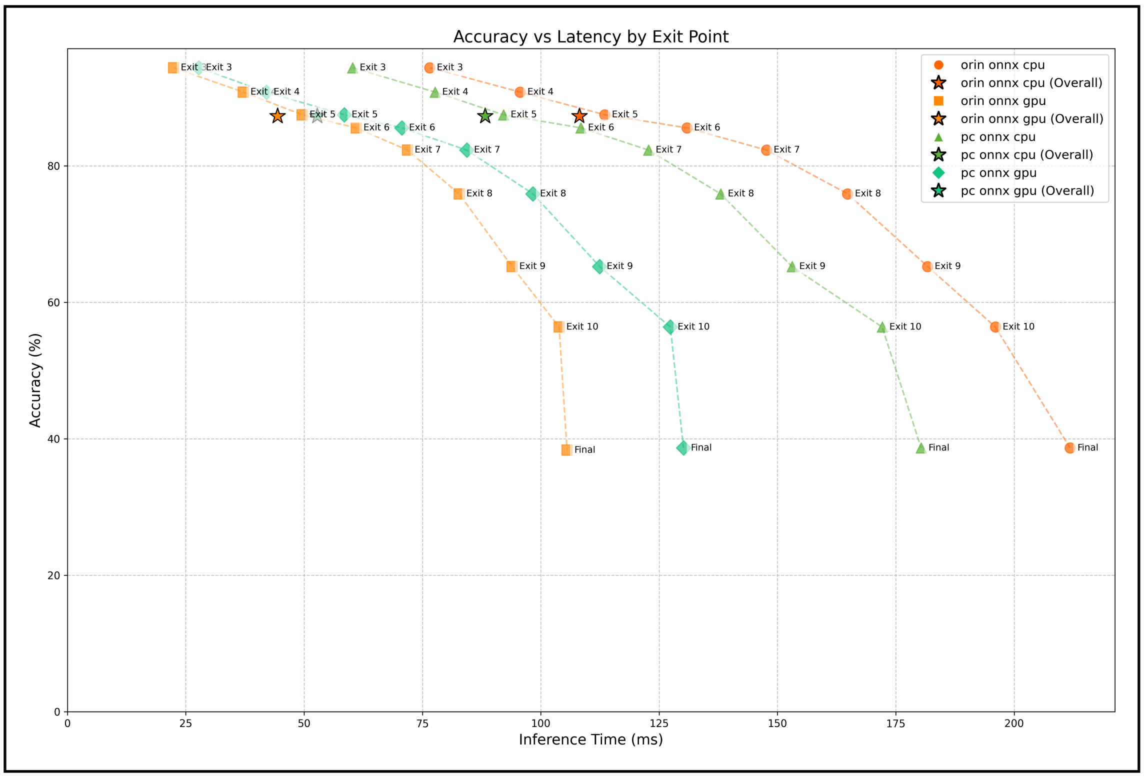

Figure 34: latency-accuracy tradeoff in different hardware targets

From a latency perspective, the slowest configuration is given by the Orin's CPU, followed by the PC's CPU. GPU acceleration provided significant speedups, with the PC's GPU performing better than these CPU implementations, while the Orin's GPU delivered the fastest overall performance.

Interestingly, I observed a counterintuitive result: while the PC outperformed the Orin in CPU-based inference, the situation was reversed for GPU acceleration, with the Orin's GPU achieving better performance than the PC's discrete graphics card. This may be attributed to the specialized architecture of the Orin's GPU, which is optimized for machine learning workloads.

5.3.4 Performance on CIFAR100

To better understand how the early exit architecture performs across different visual categories, I conducted a benchmark study[12] examining the model's behavior with respect to individual classes in the CIFAR100 dataset. This analysis aimed to uncover patterns in how different visual categories interact with the early exit mechanism, potentially revealing insights about which types of images benefit most from this approach.

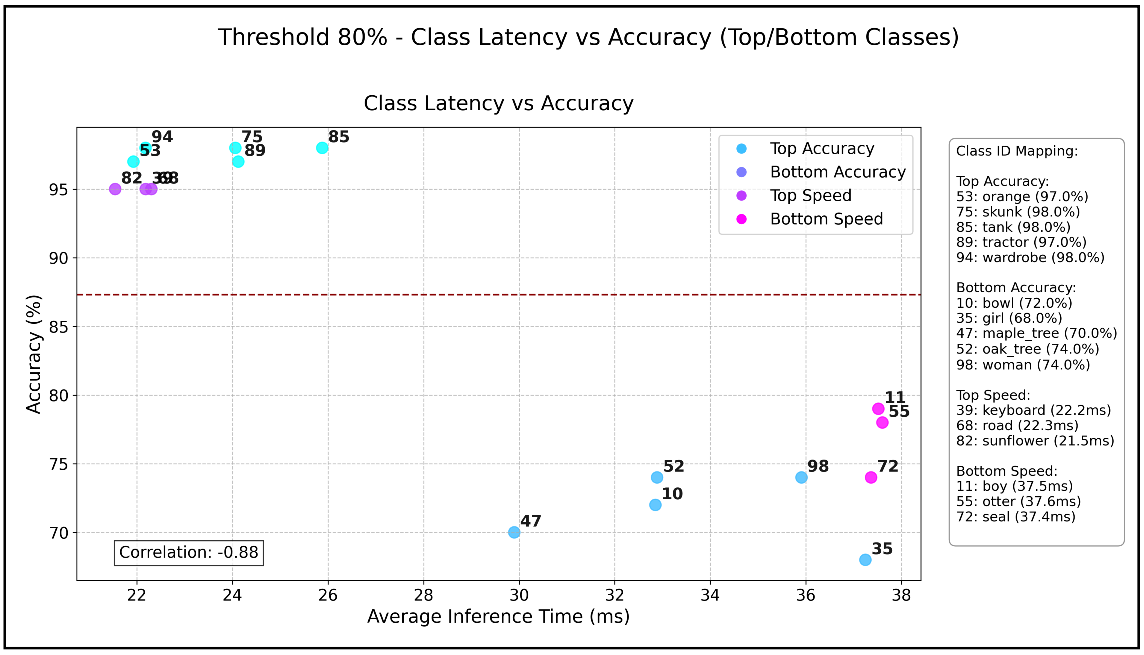

Figure 35: Accuracy-latency for top/bottom 5 classes of CIFAR100

Figure 35 presents the performance metrics for the top and bottom five classes in terms of both accuracy and inference speed. A clear pattern emerged when examining the exit point distribution for these classes, as shown in Figure 36. Notably, the most accurately classified and fastest processed classes predominantly exited at layer 3 - the earliest point in the network.

[12]with the confidence threshold of 0.8

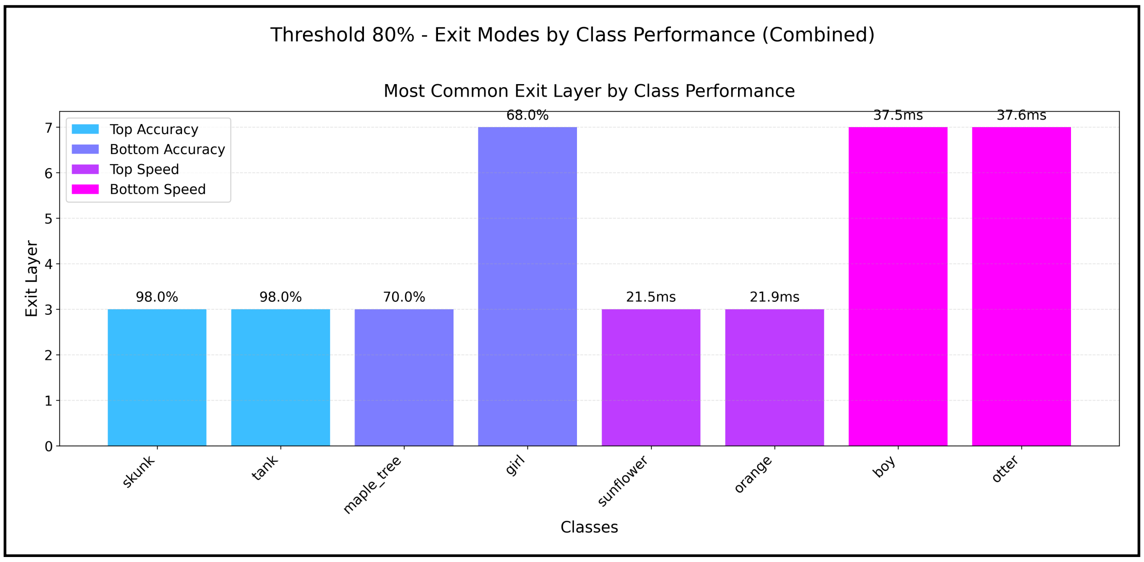

Figure 36: Most common exit for top/bottom most accurate classes

To visually investigate the characteristics of these classes, I examined samples from four representative categories highlighted in the exit distribution analysis. The detailed exit distribution for these standout classes is further illustrated in Figure 37. In it we see that the model specializes for the best-performing classes, exiting quite early for a large proportion of their samples.

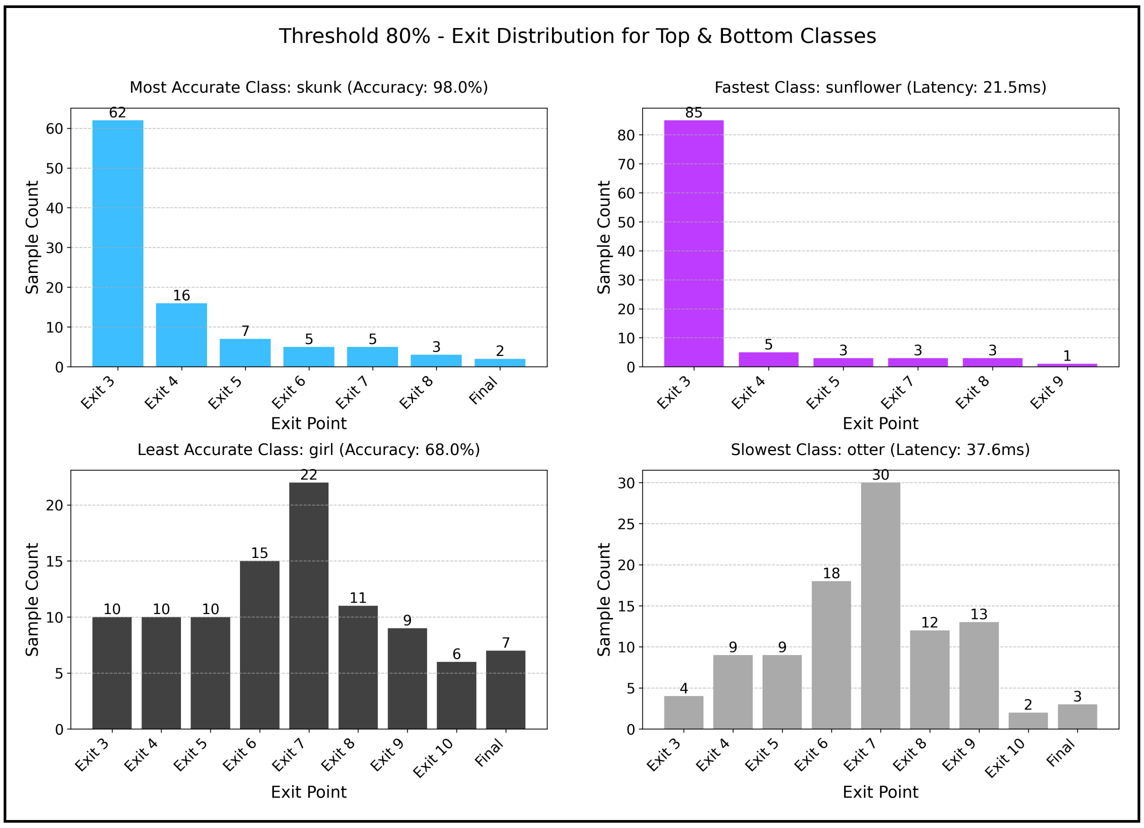

Figure 37: Exit distribution for top/bottom most accurate and fastest classes

Figures 39, 40, 41, and 38 each display 15 sample images from their respective classes. A pattern becomes apparent: classes “skunk” and “sunflower” show remarkably consistent visual features across samples. In contrast, “girl” and “seal” images exhibit much greater variation in appearance, pose, and background.

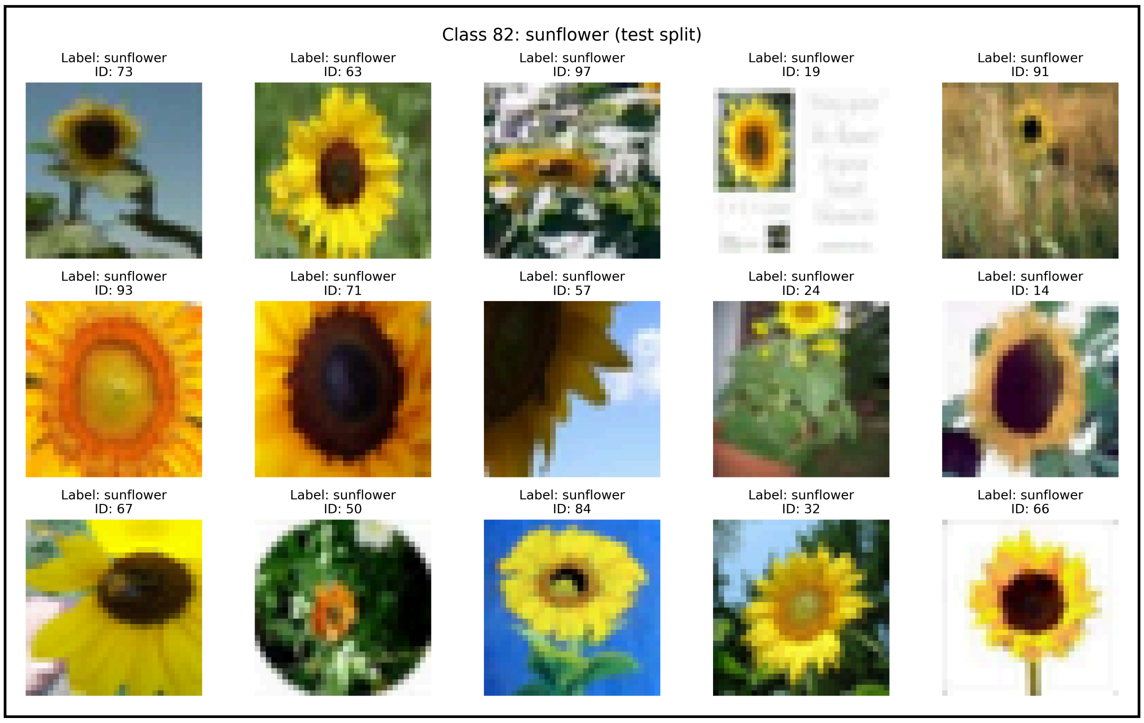

Figure 38: Sample image of Sunflower class

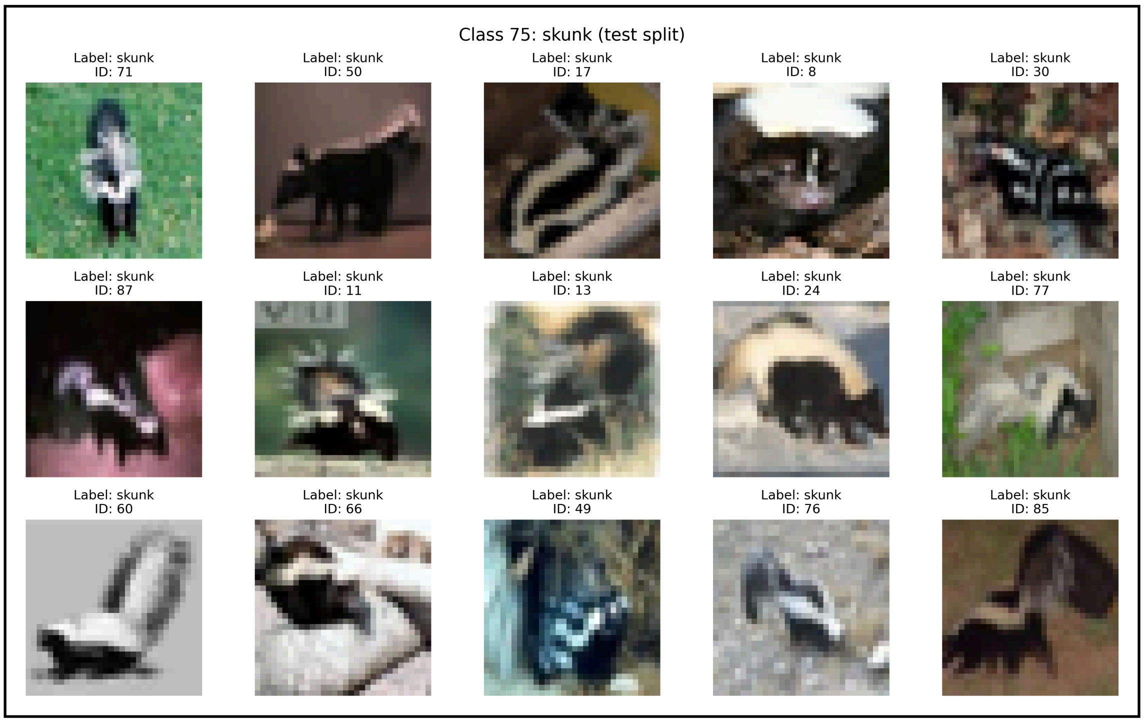

Figure 39: Sample image of Skunk class

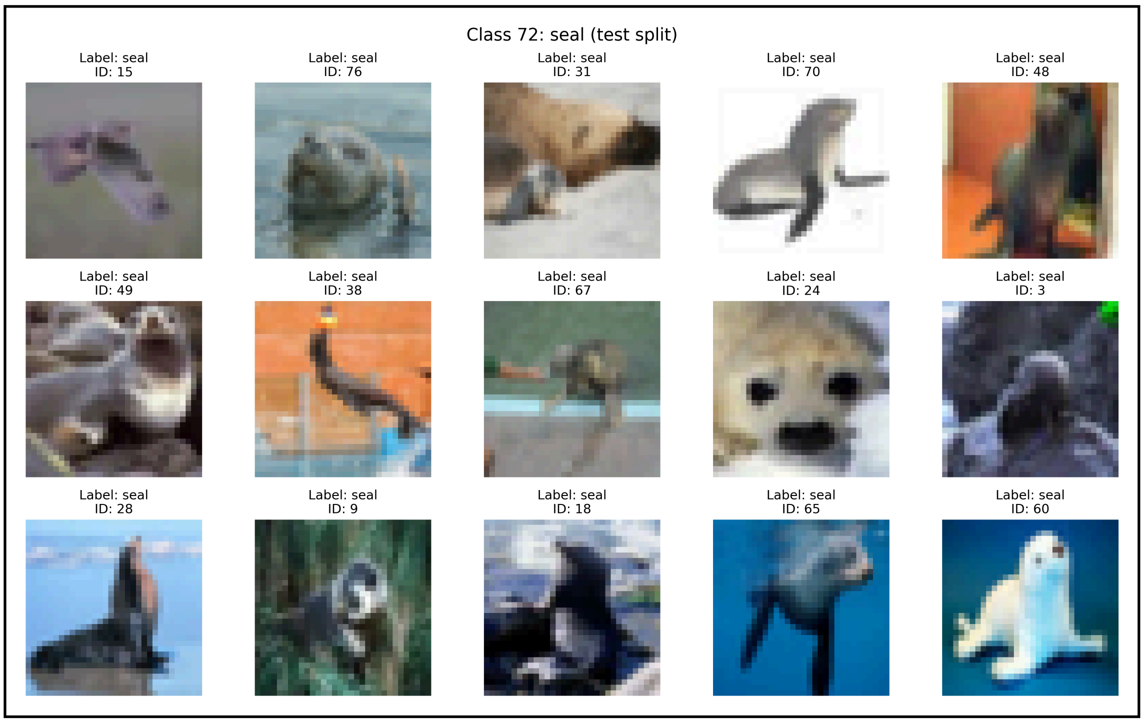

Figure 40: Sample image of Seal class

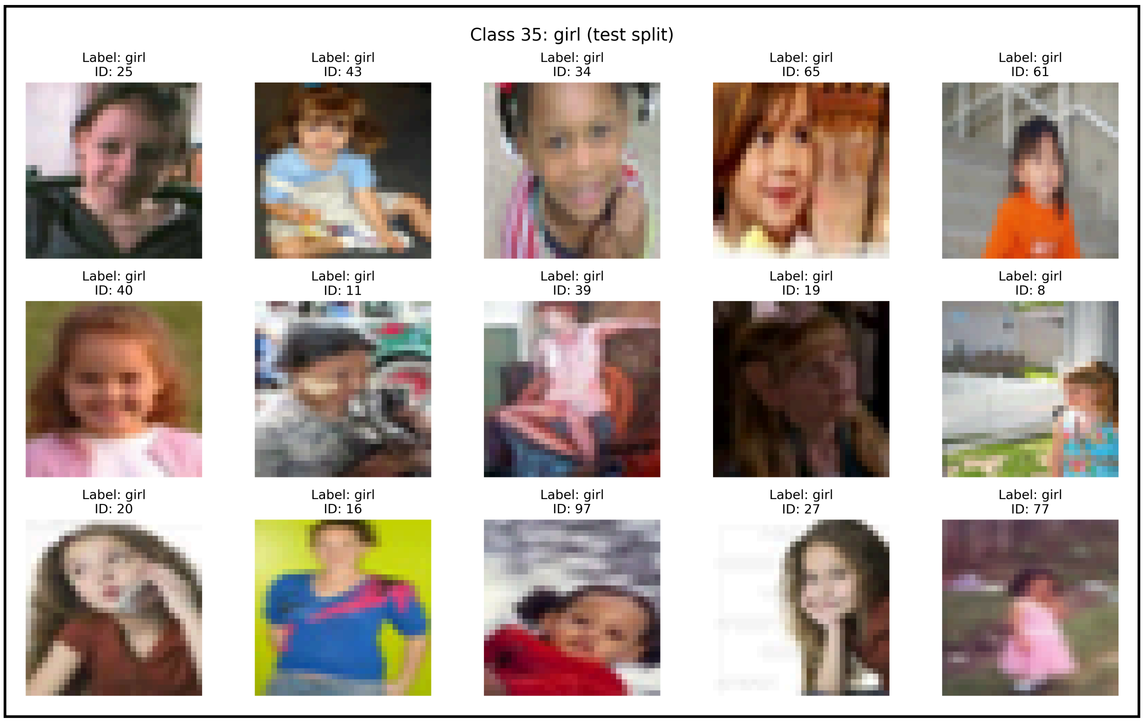

Figure 41: Sample image of Girl class

6 Discussion

This work aimed to assess the effort needed to deploy early-exit neural networks in production environments with minimal dependencies requirements. Together with it, it set itself to extend the understanding behind the performance of such models, focusing on the performance of their individual exits.

6.1 On Deployment

Prototype models (as those from research work presented in publications) are justifiably implemented not with deployment in mind, hence making them difficult to adopt. The new implementation presented here answers to the production environment requirements previously mentioned, since it provides a way to export easily a framework-agnostic version of the model, based on ONNX standard.

Some challenges faced were related to the complexity of the original codebase, which obscures the core logic of the model. Moreover, the development of this kind of network, when taking deployment into consideration, force the developer to consider to consider the requirements set by yhe exporting tools, which likely, as was the case here, will impact in some degree the code itself.

This contribution allows for the model to be tested with ease in more environments not bounded by the need to install the heavy PyTorch package or any other development framework. Looking beyond the specific logic of the LGViT model, the implementation shows that it can be used for any case where the neural network has a decision based computation flow. Finally, the designed choices taken allow for flexibility in the specification of exit type and placement, allowing for easier benchmark, which this work directly benefited from.

6.2 On Performance

As for the benchmarking studies, the following can be concluded:

The profiling study (Section 5.3.1) shows that the inclusion of early exits in this model brings with it latency overheads at the attention block level, particularly significant in the GPU execution of it. However, the absolute cost of the early-exits is heavily related to their implementation, and therefore can be mitigated. Two examples illustrate this:

1. The mechanism that assign the value to the fast-pass token brings with it additional overhead, as can be seen in Figure 29
2. The exit mechanism brings important latency overhead when evaluating the initial non-early-exit attention blocks, as can be appreciated in Figure 26. Although it is related to the implementation, it remains unclear why this occurs.

However, despite these implementation-related performance penalties, the overall time savings from early exits remain substantial. The additional computation introduced by early exit mechanisms is significantly outweighed by the computation avoided when skipping remaining layers, confirming the effectiveness of the approach even with current implementation constraints.

From the threshold study (Section 5.3.2) we can further conclude that the exit configuration for the LGViT has more than one configuration setting, particularly for the confidence threshold, that yields satisfactory results. A very interesting result, is that for the threshold of 80%, exit three processes the significant majority of the results, confirming the observation in other works of similar behavior, as mentioned in this work's introduction.

Furthermore, changing the threshold shows that the model has more than one satisfactory confidence configuration. When changing the threshold to 99%, the model main exit is then the one at the seventh layer. This configuration increases the accuracy of all exits, and although yield a lower speed-up factor than the original 80% configuration, it is in fact interesting that it would perform better, particularly because the model was trained with the 80% threshold.

I theorize that this has to do with the distillation characteristic of the training; The *vanilla* ViT with no exits scores 90.24% and its trained classifier is used in training the intermediate ones via knowledge distillation. It seems that this yields benefits for the performance of these intermediate classifiers.

From class-wise study of Section 5.3.4, the analysis can be quite rich. Seeing that there is a big performance difference among classes (Figure 35), we can initially conclude that some characteristic within each class renders them easier or more challenging to predict, which is hardly a discovery. What is more interesting is that the most accurately predicted classes are also the fastest to exit the model.

This observation suggests a relationship between visual feature consistency and early exit behavior. Classes with low intra-class variability appear to be processed more efficiently by the model, allowing confident predictions at earlier network stages. The model effectively learns to recognize these more consistent visual patterns with fewer layers of computation, reserving deeper processing only for more varied or challenging examples.

This is explained by the fact that classes being exited early require by definition less processing, and the fact that they are being accurately classified implies that their visual features are less complex than those from other classes.

This leads to the conclusion that classes with low variance in their visual features will exit early and be classified more accurately. Supporting this claim we see the examples from the best-performing/fastest classes in figures 38 and 39 and for the worst performing classes, we see figures 41 and 40.

The platform benchmark comparison (Section 5.3.3) shows that the benefits of early exiting aer also profitable on edge computer. We see a clear best performer in the GPU case for the edge computer, and interestingly, it's

CPU performance being worse than both CPU and GPU cases for the consumer computer, which must be noted carries a relatively old processors CPU/GPU. This might be related to the fact that the GPU in the Orin is of ARM architecture with a lower cache size despite having a higher memory bandwidth (204.8 GB/s for the Orin and 21.33 GB/s for the PC).

6.3 General Remarks

This work has contributed a study on the ease of deployment and performance characteristics of LGViT early-exit model. We can conclude that the model, after thorough re-factorization has been made deployable to environments that cannot have the framework in which it was originally developed. This new version of the model displays same level of performance of the overall model.

Furthermore, based on the studies carried out, some deeper understanding of the behavior of the model has been found, which invites interesting conclusions.

This relationship between visual feature consistency and early exit behavior has interesting implications for real-world applications, among others, in domains like robotics navigation. Consider a vision model trained on data captured under good lighting conditions. When deployed in similar favorable conditions, the model would likely process most inputs through early exits, allowing for faster inference speeds. However, if lighting conditions deteriorate, the model would automatically route more samples through deeper layers, maintaining prediction quality at the cost of increased processing time.

Such adaptive processing creates opportunities for dynamic control of operations, based on perception difficulty. In robotics navigation, for example, this behavior could enable speed optimization based on environmental conditions. When visual conditions are optimal, a robot could operate at higher speeds since the perception module processes visual information rapidly through early exits. Conversely, in challenging visual environments such as low light or complex scenes, the system would naturally slow down as more inputs require deeper processing, effectively mimicking how humans adjust driving speed based on visibility conditions.

This parallel to human behavior is instructive - just as drivers naturally reduce speed at night when visibility is compromised, an early exit vision system takes more processing time when input quality degrades. The result is a control system that dynamically adapts its control signal rate based on input complexity, operating always in the safest, fastest rate across varying conditions that circumstances allow.

Furthermore, from a safety assessment perspective, the dynamic nature of this models raises the need to specify their behavior ina more detailed way than by just stating their latency. Some insight in the relation of that latency to the dataset is necessary, in order to better estimate the performance of the model in novel data entries. The need for low latency can be amplified further if we could understand better which data can yield the best results.

7 Future Work

The early-exit paradigm used in Vision Transformers could be leveraged to benefit more practical tasks. For example, applying this principle to the Detection Transformer (DeTr) (Carion et al., 2020) would be valuable, targeting both the CNN backbone and the transformer's encoder or decoder components.

The main challenges would likely be in developing an appropriate training scheme and determining the correct intermediate heads, particularly for the decoder's cross-attention mechanism. This research direction is especially interesting as it connects with work that has already leveraged transformers for real-time applications, such as RT-DeTr (Zhao et al., 2024) or MonoNav (Simon & Majumdar, 2023), both of which employ transformers in scenarios where low latency is highly desirable.

Beyond classification tasks, it would be interesting to explore applying early-exits to other data structures where transformers have proven effective. For instance, the Point Transformer (Zhao et al., 2021), used for point-cloud classification, could serve as an excellent starting point for extending these techniques to 3D applications.

Following the insights from our class-wise study, feature visualization studies could help better understand the relationship between early exiting and the high accuracy we observed. Similar to the attention mapping done in the original ViT work (Dosovitskiy et al., 2021), a qualitative assessment of what each exit learns would be valuable. While LGViT conducted similar work, they did so to determine what kind of highway to place where, rather than examining the exits after training with highways in their final positions.

Bibliography

[1] D. Azizov *et al.*, “A Decade of Deep Learning: A Survey on The Magnificent Seven,” *ArXiv*, 2024, [Online]. Available: https://api.semanticscholar.org/CorpusID:274982049

[2] S. Teerapittayanon, B. McDanel, and H. T. Kung, “BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks,” in *23rd International Conference on Pattern Recognition (ICPR)*, 2016, pp. 2464–2469.

[3] F. D. Keles, P. M. Wijewardena, and C. Hegde, “On The Computational Complexity of Self-Attention,” 2022, [Online]. Available: https://arxiv.org/abs/2209.04881

[4] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-Attention with Linear Complexity,” *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2006.04768

[5] Z. Liu *et al.*, “Swin Transformer: Hierarchical Vision Transformer using Shifted Windows,” *CoRR*, 2021, [Online]. Available: https://arxiv.org/abs/2103.14030

[6] F. Montello, R. Güldenring, S. Scardapane, and L. Nalpantidis, “A Survey on Dynamic Neural Networks: from Computer Vision to Multi-modal Sensor Fusion.” [Online]. Available: https://arxiv.org/abs/2501.07451

[7] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, “Why should we add early exits to neural networks?,” *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2004.12814

[8] P. Panda, A. Sengupta, and K. Roy, “Conditional Deep Learning for Energy-Efficient and Enhanced Pattern Recognition,” *CoRR*, 2015, [Online]. Available: http://arxiv.org/abs/1509.08971

[9] Y. Kaya and T. Dumitras, “How to Stop Off-the-Shelf Deep Neural Networks from Overthinking,” *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1810.07052

[10] G. Xu *et al.*, “LGViT: Dynamic Early Exiting for Accelerating Vision Transformer,” in *Proceedings of the 31st ACM International Conference on Multimedia*, in MM '23. Ottawa ON, Canada: Association for Computing Machinery, 2023, pp. 9103–9114. doi: 10.1145/3581783.3611762.

[11] A. Vaswani *et al.*, “Attention is All You Need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[12] A. Rush, “The Annotated Transformer.” 2018.

[13] A. Dosovitskiy *et al.*, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” in *International Conference on Learning Representations*, 2021.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1810.04805

[15] A. Bakhtiarnia, Q. Zhang, and A. Iosifidis, “Multi-Exit Vision Transformer for Dynamic Inference,” *arXiv preprint arXiv:2106.15183*, 2021.

[16] X. Li *et al.*, “Predictive Exit: Prediction of Fine-Grained Early Exits for Computation- and Energy-Efficient Inference,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 7, pp. 8657–8665, Jun. 2023, doi: 10.1609/aaai.v37i7.26042.

[17] H. Rahmath P, V. Srivastava, K. Chaurasia, R. G. Pacheco, and R. S. Couto, “Early-exit deep neural network-a comprehensive survey,” *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–37, 2024.

[18] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, “Adaptive Neural Networks for Fast Test-Time Prediction,” *CoRR*, 2017, [Online]. Available: http://arxiv.org/abs/1702.07811

[19] F. Xia, “Exploring Early Exiting Strategies for Deep Neural Networks,” Princeton, NJ, 2024. [Online]. Available: http://arks.princeton.edu/ark:/88435/dsp0179408155k

[20] ONNX Community, “ONNX Documentation.” [Online]. Available: https://onnx.ai/onnx/index.html#

[21] ONNX Project Contributors, “Open Neural Network Exchange Intermediate Representation (ONNX IR) Specification.” [Online]. Available: https://github.com/onnx/onnx/blob/main/docs/IR.md

[22] R. Mario, “JIT — PyTorch Training Performance Guide.” [Online]. Available: https://residentmario.github.io/PyTorch-training-performance-guide/jit.html

[23] P. S. Foundation, “PEP 523 - Changing Frame Evaluation API.” [Online]. Available: https://peps.python.org/pep-0523/

[24] P. Team, “torch.onnx.” [Online]. Available: https://pytorch.org/docs/stable/onnx.html

[25] PyTorch Team, “Higher-Order Operators.” [Online]. Available: https://dev-discuss.pytorch.org/t/higher-order-operators-2023-10/1565

[26] T. Wolf *et al.*, “HuggingFace's Transformers: State-of-the-art Natural Language Processing.” [Online]. Available: https://arxiv.org/abs/1910.03771

[27] A. Krizhevsky, “Learning multiple layers of features from tiny images,” 2009.

[28] P. Wang, “vit-PyTorch: Implementation of Vision Transformer in PyTorch.” GitHub, 2025.

[29] P. Team, "CPU Threading and TorchScript Inference." [Online]. Available: https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html

[30] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-End Object Detection with Transformers," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2005.12872

[31] Y. Zhao *et al.*, "DETRs Beat YOLOs on Real-time Object Detection." [Online]. Available: https://arxiv.org/abs/2304.08069

[32] N. Simon and A. Majumdar, "MonoNav: MAV Navigation via Monocular Depth Estimation and Reconstruction," in *Symposium on Experimental Robotics (ISER)*, 2023. [Online]. Available: https://arxiv.org/abs/2311.14100

[33] H. Zhao, L. Jiang, J. Jia, P. H. Torr, and V. Koltun, "Point transformer," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 16259–16268.

Annex

7.1 EEVIT UML diagram of components

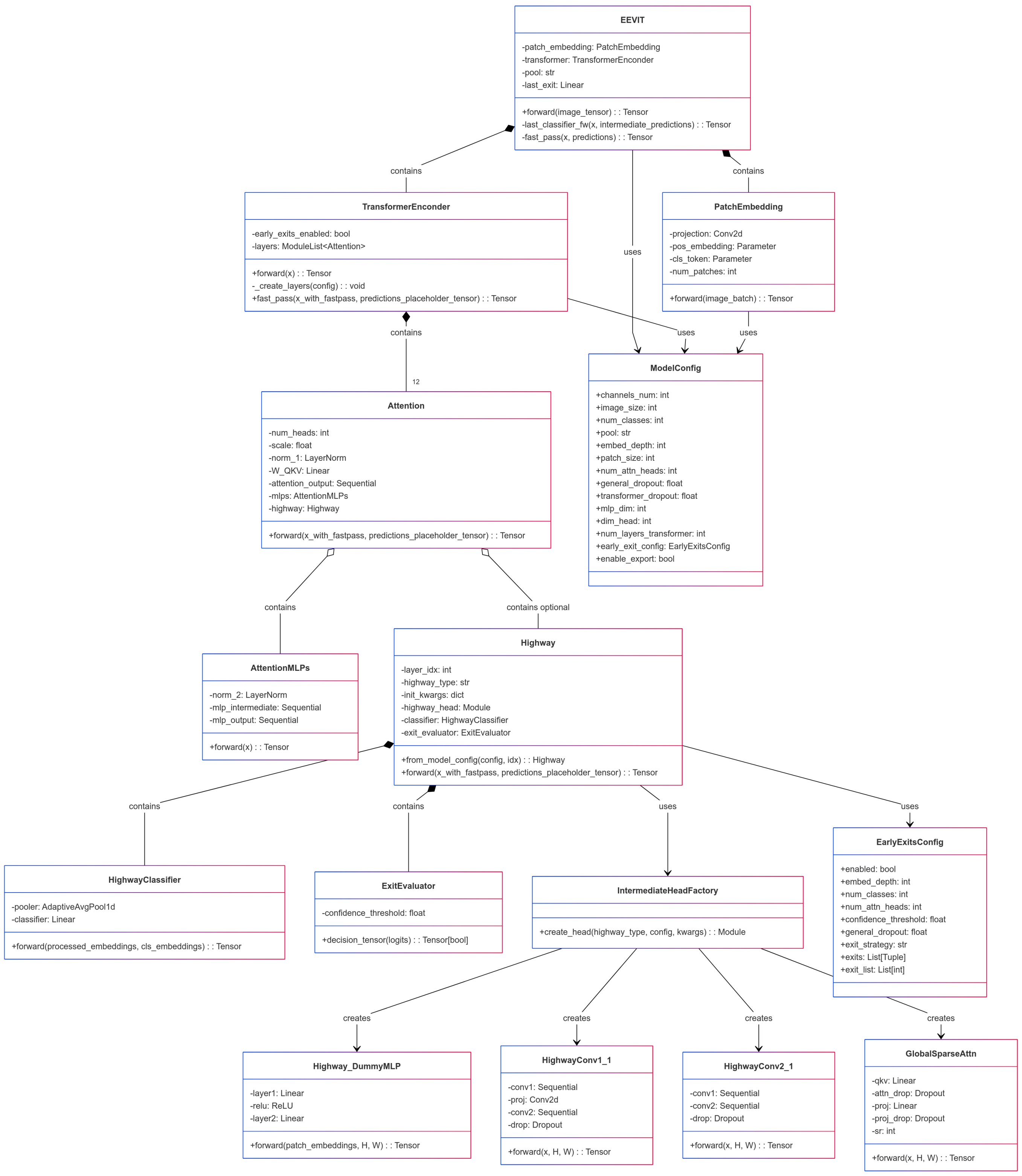

Figure 42: Simplified Attention-Highway class diagram

7.2 Profiling additional results

Layer Group	Avg (ms)	Std Dev	Min (ms)	Max (ms)	Count
0	14.925	11.806	11.106	93.273	200
1	15.874	2.500	13.753	24.238	200
2	15.686	2.293	12.780	21.535	200

Table 12: Layer groups profiling with **100** warmup iterations. We see group 0 behaving more stable and slightly faster than the rest

Layer	Avg (ms)	Std Dev	Min (ms)	Max (ms)	Count
Layer_0	15.626	13.778	11.610	92.001	50
Layer_1	17.277	18.809	11.106	93.273	50
Layer_2	13.223	2.517	11.197	20.527	50
Layer_3	16.323	2.699	14.209	23.681	50
Layer_4	15.880	2.657	13.962	24.238	50
Layer_5	15.601	2.374	13.753	22.270	50
Layer_6	15.694	2.253	13.783	22.215	50
Layer_7	15.330	2.366	13.060	21.535	50
Layer_8	15.772	2.218	12.780	21.214	50
Layer_9	16.002	2.394	12.981	21.193	50
Layer_10	15.638	2.207	13.099	20.441	50
Layer_11	13.577	2.273	11.160	19.045	50

Table 13: Layer profiling with **100** warmup iterations for the all attention layers

Layer	Avg (ms)	Std Dev	Min (ms)	Max (ms)	Count
Layer_0	14.455	11.135	11.148	89.387	50
Layer_1	17.717	19.433	11.218	98.959	50
Layer_2	13.388	2.435	11.196	22.365	50
Layer_3	18.133	8.951	14.597	77.302	50
Layer_4	16.513	2.473	14.223	25.441	50
Layer_5	16.120	2.140	13.944	23.289	50
Layer_6	16.285	2.215	14.644	23.824	50
Layer_7	15.805	2.396	13.932	22.290	50
Layer_8	15.853	2.407	13.755	23.913	50
Layer_9	15.512	2.498	13.735	22.641	50
Layer_10	15.265	2.420	13.614	22.510	50
Layer_11	13.546	2.393	11.860	21.772	50

Table 14: Layer profiling with **200** warmup iterations for the all attention layers

Layer	**Avg (ms)**	**Std Dev**	**Min (ms)**	**Max (ms)**	**Count**
Layer_0	2.880	9.757	1.284	70.473	50
Layer_1	5.588	17.110	1.134	72.840	50
Layer_2	1.363	0.242	1.119	2.165	50
Layer_3	5.258	2.587	4.684	23.045	50
Layer_4	4.911	0.387	4.661	6.328	50
Layer_5	5.074	0.314	4.656	5.996	50
Layer_6	4.865	0.280	4.625	6.075	50
Layer_7	4.652	0.278	4.439	5.717	50
Layer_8	4.625	0.212	4.491	5.911	50
Layer_9	4.639	0.310	4.425	6.012	50
Layer_10	4.588	0.164	4.487	5.501	50
Layer_11	1.267	0.204	1.092	2.147	50

Table 15: Attention Latency by Individual Layer for the **GPU** case. TODO: move this to an annex for the detailed report of the profile study