Reinforcement Learning Based Layer Skipping Vision Transformer for Efficient Inference

By

Amanuel Negash Mersha

Submitted to the School of Information Technology and Engineering

in partial fulfillment of the requirements for the degree of

Master of Science in Artificial Intelligence

Addis Ababa Institute of Technology

Addis Ababa University

Addis Ababa, Ethiopia

May 2023

Approval

This is to certify that this thesis titled ***Reinforcement Learning Based Layer Skipping Vision Transformer for Efficient Inference*** is prepared by ***Amanuel Negash Mersha*** and submitted in partial fulfillment of the thesis-option requirements for the Degree of Master of Science in Artificial Intelligence at School of Information Technology & Engineering, Addis Ababa Institute of Technology.

Advisor

Name	Signature	Date

Co-Advisor

Name	Signature	Date

Examiner

Name	Signature	Date

Examiner

Name	Signature	Date

Abstract

Recent advancements in language and vision tasks owe their success largely to the Transformer architecture. However, the computational requirements of these models have limited their applicability in resource-constrained environments. To address this issue, various techniques, such as Weight pruning, have been proven effective in reducing the deployment cost of such models. Additionally, methods tailored just for transformers, such as linear self-attention and token early exiting, have shown promise in making transformers more cost-effective. Nevertheless, these techniques often come with drawbacks such as decreased performance or additional training costs. This thesis proposes a layer-skipping dynamic vision transformer (ViT) network that skips layers depending on the given input based on decisions made by a reinforcement learning agent (RL). To the best of our knowledge, this work is the first to introduce such a model that not only significantly reduces the computational demands of transformers, but also improves performance. The proposed technique is extensively tested on various model sizes and three standard benchmarking datasets: CIFAR-10, CIFAR-100, and Tiny-ImageNet. First, we show that the dynamic models improve performance when compared to their state-of-the-art static counterparts. Second, we show that in comparison to these static models, they achieve an average inference speed boost of 53% across different model sizes, datasets, and batch sizes. Similarly, the technique lowers working space memory consumption by 53%, enabling larger input processing at a time without imposing an accuracy-speed trade-off. In addition, these models achieve very high accuracy when tested in transfer learning scenarios. We then show that, although these models have high accuracy, they can be optimized even more through post-training using genetic algorithms (NSGA-II). As such, we propose the joint RL-NSGA-II optimization technique, where the GA is aware of the dynamics of skipping through the RL reward. These optimized models achieve competitive performance compared to the already high-performing dynamic models while reducing the number of layers by 33%. In real-world applications, the technique translates to an average of 53% faster throughput, reduced power consumption, or lower computing costs without loss of accuracy.

Acknowledgements

I am deeply grateful to my advisor, Dr. Beakal Gizachew, for his invaluable guidance and continuous encouragement throughout my M.Sc. thesis journey. His expertise, profound insights, and constructive feedback have played a pivotal role in shaping my research ideas and transforming them into tangible results. In addition, I would like to thank Dr. Sammy Assefa for his unwavering support throughout the work.

I would also like to also express my sincere appreciation to the thesis examiners, namely Dr. Henock Mulugeta, Dr. Sileshi Demesie, Dr. Elefelious Getachew, Dr. Adane Mamuye, Dr. Fantahun Bogale, and Dr. Natnael Argaw, for dedicating their time and providing valuable feedback during the course of my thesis work.

Furthermore, I extend my gratitude to my friends and colleagues, Michael Sheleme, Tewodros Wondifraw, and Selamab Setargew, for their insightful comments and suggestions, which have significantly contributed to the enhancement of my thesis from various perspectives. I am also thankful to Sara Hooker and CohereAI, as well as Tewodros Wondifraw and Selamab Setargew, for their generous support in terms of computing power. Their assistance has been instrumental and has greatly propelled my experiments.

I would like to acknowledge my classmates for engaging in stimulating discussions and exchanging valuable ideas. In particular, I want to thank Dereje Tadesse, Mubarek Mohammed, Eyob Solomon, and Tesfaye Mengistu for all the support, idea brainstorming, and fun we had working together. Similarly, I would like to thank all the students that I have taught over the years. I'm lucky to teach such a sharp group of students who kept challenging me and pushing me to advance myself. Lastly, none of this would have been possible without the support and dedication of the School of Information Technology and Engineering (SITE). I am immensely grateful to all of my colleagues for their contributions to making this program a reality.

Dedication

I wish to express my heartfelt dedication to this work to two remarkable individuals in my life: my father, **Negash Mersha Direta**, and my sister, **Firehiwot Negash Mersha**. Despite facing numerous challenges, they have consistently created a perfect world for me, allowing me to concentrate on my education and excel in my studies throughout my life. I am forever grateful to them for their unwavering support. Additionally, I would like to extend my dedication to my uncle, **Solomon Mersha Direta**, whose mentorship and encouragement have constantly pushed me to surpass my own limits. You will always be in my heart.

Contents

List of Tables

List of Figures

List of Abbreviations

AI Artificial Intelligence.

BERT Bidirectional Encoder Representations from Transformers.

BLEU Bilingual Evaluation Understudy.

BLOOM BigScience Large Open-science Open-access Multilingual Language Model.

BPTT Backpropagation Through Time.

CIFAR Canadian Institute for Advanced Research.

CNN Convolutional Neural Network.

CO2 Carbon Dioxide.

CPU Central processing unit.

DQN Deep Q-Networks.

FFN Feedforward Neural Network.

GPT Generative Pre-trained Transformer.

GPU Graphical Processing Unit.

GRU Gated Recurrent Unit.

LSTM Long Short-Term Memory.

MLP Multilayer Perceptron.

MLP Multi-layer Perceptron.

MOO Multi-Objective Optimization.

NLP Natural Language Processing.

NMT Neural Machine Translation.

OPT Open Pre-trained Transformer.

PPO Proximal Policy Optimization.

RL Reinforcement Learning.

Seq2Seq Sequence-to-Sequence.

SMT Statistical Machine Translation.

SVD Singular Value Decomposition.

UCB Upper Confidence Bound.

ViT Vision Transformer.

WMT Workshop on Statistical Machine Translation.

XOR Exclusive OR.

Chapter 1 Introduction

Over the past decades, machine learning has shown impressive growth as it became ubiquitous in both academia and industry [4]. In particular, Deep learning, over the past decade, became the go-to strategy for complex and massive datasets [5]. Applications such as speech recognition [6], speech generation [7], image generation [8], image classification [3], protein folding [9], game playing [10], and language models [11] have greatly benefited from this innovation. Such success of deep learning is mostly propelled by the abundance of data and the ever-increasing computational power of Graphical Processing Units (GPU) [12]. The internet, social media, and investments made by large technology corporations enabled the gathering of vast amounts of data [13, 14, 15]. At the same time, the innovation in the production of computer chips both for CPU as well as GPU accelerated the accessibility of powerful computing power [12]. As a result, researchers and engineers were able to develop large models that can learn the vast complexities of a given data [16].

Deep learning is any learning system that transforms signals or inputs through multiple levels to turn them into prediction or output [4]. As such, an input is propagated through several layers of transformation [5]. As complexities in the data increase, the required number of layers also increases. Hence, datasets containing images or text require the possible largest number of layers depending on the model type [13, 17, 3]. Furthermore, each layer has to sufficiently represent a certain transformation. Due to these reasons, sufficient deep learning models have both depth and width [18]. Figure 1 shows how much deep learning-based language models have grown in size over the years.

In particular, the Transformer [19], a recent deep learning architecture, has gained significant popularity in the fields of natural language processing [19], computer vision [3], multimodal data [20], and beyond [21, 22]. It adopts the mechanism of self-attention, which allows it to differentially weigh the significance of each part of the input data. Unlike traditional

recurrent neural networks (RNNs) [23, 24] or convolutional neural networks (CNNs) [25], the Transformer does not rely on recurrence or convolutions. Instead, it utilizes attention mechanisms to capture relationships between different elements of the input [19].

In the context of NLP, the Transformer has revolutionized the field by significantly improving the performance of translation models [19, 26]. By leveraging self-attention, the model can effectively capture dependencies between words in a sentence, leading to better understanding and generation of text[19]. This has made the Transformer particularly effective for tasks such as machine translation [19], text summarization [27], sentiment analysis [28], and question answering [29]. Furthermore, the Transformer's capabilities extend beyond NLP as it has been successfully applied to image-related tasks in computer vision [3, 30]. The self-attention mechanism allows the model to capture long-range dependencies in images, enabling it to understand and analyze complex visual patterns[19]. This has led to remarkable progress in tasks such as image classification [3], image generation [8], and image captioning [31].

Moreover, the Transformer has shown its effectiveness in handling multi-modal datasets [20, 32, 33], which involve multiple types of data sources, such as text, images, and audio. Its ability to model relationships between different modalities through attention mechanisms makes it suitable for tasks that require fusion and interaction between diverse data sources such s video analysis [34], visual question answering [35], and audio-visual speech recognition [36].

Overall, the popularity of the Transformer can be attributed to its outstanding performance across a wide range of tasks and data types and efficient use of the powerful GPUs through parallelization techniques [19]. As such it has become the go-to choice for deep learning practitioners working with different types of data.

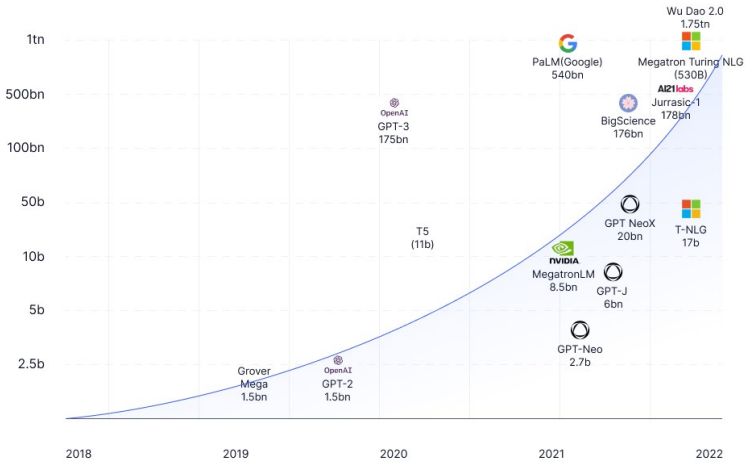


Figure 1: The ever increasing size of large language models - Taken from TextCortex. X-axis shows the release time of the models. Y-axis shows the number of parameters the models have in billions.

1.1 Motivation

1.1.1 Computational Demand

Although Transformers have shown to be an excellent choice in many settings, such versatility or impressive performance do not come without a price as they demand large datasets and sufficient compute power that comes with it [3, 19]. This high computational demand for Transformers is attributed to the self-attention costly operation and the feed-forward layer that increases the model's complexity [19, 37].

Another aspect that adds to the computational demands of Transformers is their parallel nature[19]. Unlike traditional RNNs [23] that process sequential data sequentially, Transformers can process the entire sequence in parallel due to the attention mechanism [19]. This parallel processing offers efficiency benefits but still requires significant computational resources to handle large-scale datasets or complex tasks effectively. Furthermore, the need

for extensive training data is another consideration as it increases both the data acquisition and processing cost [3].

1.1.2 Negative Environmental Impact

The high resource requirements of deep learning models can lead to a negative environmental impact primarily through carbon emissions [38, 39, 40]. The training and deployment of these models rely on power-hungry hardware infrastructure, such as high-performance servers and data centers, which consume significant amounts of electricity [41]. Often, this electricity comes from non-renewable sources like fossil fuels, resulting in the release of carbon dioxide (CO2) into the atmosphere. Additionally, the Transformer based large-scale training process, which involves processing vast datasets, further amplifies the computational requirements and subsequent environmental impact [42], necessitating the need for sustainable practices and energy-efficient approaches in AI development.

1.1.3 Widening Economy Disparity

In addition to the environmental impacts, large models can contribute to increased economic disparities globally [43, 44]. The computational demands of training and running these models often necessitate access to substantial computing infrastructure and resources, which can be costly to acquire and maintain [3]. This puts a financial burden on organizations and individuals seeking to utilize Transformer models for various applications. As a result, economically disadvantaged regions or communities may face barriers to accessing and benefiting from the advanced capabilities of these models, further widening the digital divide and exacerbating existing economic disparities [43, 44]. The unequal distribution of resources and opportunities related to Transformer models can perpetuate inequalities, limiting the participation and potential of marginalized populations in the rapidly evolving landscape of natural language processing and AI technologies [43, 45].

As the utilization of Transformer-based models continues to expand across various domains,

their advantages have garnered significant attention [26]. In light of this, the thesis places a specific emphasis on these models and strives to enhance their efficiency in terms of computational power requirements. To achieve this goal, this work devised the RL-based Dynamic Transformer Network, a novel network architecture that exhibits the remarkable ability to bypass irrelevant layers during input processing. By skipping unnecessary computations, the Dynamic Transformer Network effectively reduces the overall processing time without sacrificing performance or accuracy.

1.2 Statement of the Problem

Deep learning networks have grown in size mostly as a result of advancements in processing power, data accessibility, and algorithmic advancements [5]. Modern deep learning models like GPT-3 [11], for instance, have around 175 billion parameters. They have astounding performance on a variety of tasks due to their superior parameter count, which allows them to catch minute details and complex nuances in the input data [26, 41].

Such an enormous size offers both advantages and difficulties. On one hand, larger models may be more accurate and generalize more effectively across diverse datasets [26, 11, 3]. On the other side, the amount of processing power needed to develop and use these models can be enormous, creating massive infrastructure, energy, and memory demands[3]. In most cases of these models, the increase in model size does not result in a linear accuracy gain [46, 19, 3, 30].

Several works tried to alleviate the problem of Transformer efficiency through various techniques such as weight pruning [47, 48] quantization [49, 50, 51] and low-rank approximation [52, 53, 54, 55]. However, these methods effectively reduce the model size, which results in performance degradation[47, 48, 52, 53]. Knowledge Distillation (KD) [56, 30] has also been effective in lowering the computational requirement of Vision Transformer. However, the process of distillation works by transferring dark knowledge from a teacher network to a

student network [56]. As such the training of the teacher itself requires extra effort which is a significant cost to consider[30].

Sparsification through token and attention pruning is also another option to reduce the computation of Transformers [57, 58, 59, 60, 59, 60]. Unfortunately, similar to the pruning and quantization methods, these methods also result in a performance drop.

Adaptive techniques such as layer skipping and early exit methods, which are very close to this work, were also explored [61, 62, 63]. However, these works suffer from accuracy-speed trade-offs due to challenges in routing decision-making [64, 65].

The general trend in the above works is the accuracy-speed trade-off problem - when an efficiency strategy is applied, accuracy drops. Furthermore, some of these methods, although they increase performance, they require task-specific information [62]. In methods such as weight pruning and quantization, this trade-off is inevitable since there is a reduction in model size [66]. In other methods such as early exiting and layer skipping, [64, 65] identified that, among several challenges, decision-making of exiting or skipping is still a major challenge in dynamic networks. Nevertheless, skipping layers or early exiting comes naturally in many scenarios. For instance, in the Vision Transformer model case [30], while one can archive an 83% accuracy on ImageNet [67] using 12 layers, it has to be doubled to 24 to achieve 87% accuracy, twice the computational cost for 4% accuracy gain. This shows that the first 83% of the images do not need the additional layers. Hence, one can raise the question *Can the extra new layers be skipped for those images that have been successfully classified with the first 12 layers?* Related works such as [68, 69, 70] tried to approach this problem through various skipping strategy. However, they suffer from accuracy-speed trade-offs, require high memory usage or are simply limited to CNNs architecture.

In contrast, this work investigates whether, given an input, irrelevant layers can be skipped

without compromising performance for vision transformers [3]. As we will prove in subsection 4.2.2.1, some layers are irrelevant for certain inputs while they are critical for others. Such a need-based or adaptive layer activation mechanism results in an efficient model as it results in lower computations. Thus the statement of the problem is: *Can an adaptive layer activation mechanism be developed to skip irrelevant layers when processing inputs, leading to reduced computation while maintaining high accuracy?*

1.3 Research Questions

RQ1 How can the training process of Transformer models be modified to enable adaptiveness and the ability to skip layers, resulting in more efficient inference?

RQ2 How will the technique generalize across different datasets and model sizes in the context of a vision problem?

RQ3 How does the proposed system improve efficiency in terms of both time and space?

1.4 Objectives

1.4.1 General Objective

To develop a layer-skipping Vision Transformer that can lower computational demand with respect to the plain transformer network.

1.4.2 Specific Objective

- To develop an adaptive mechanism that dynamically identifies and skips irrelevant layers based on the input characteristics.
- To assess the generalizability of the optimized dynamic neural network on various datasets and model sizes.
- To investigate and analyze the impact of layer skipping on the execution time and memory usage of the model.

1.5 Significance

As outlined in the Motivation section (1.1), deep learning has both advantages and disadvantages. Without careful implementation, it can lead to environmental issues and widen the economic gap, placing already marginalized communities at a significant disadvantage. Therefore, it is crucial to pursue the development of affordable AI models to ensure that the benefits of this field are accessible to all of humanity. While other researches focus on reducing costs, this work equally prioritizes accuracy in the pursuit of an affordable AI model.

1.6 Contribution

Reinforcement Learning Based Layer-Skipping Dynamic Vision Transformer: This thesis presents a novel dynamic vision transformer network that incorporates reinforcement learning-based layer-skipping, significantly reducing computational demands of the Transformer network while improving performance. By conducting extensive evaluations on various model sizes and three standard benchmark datasets (CIFAR-10, CIFAR-100, and Tiny-ImageNet), the technique consistently outperforms static counterparts in terms of both efficiency and performance. The models achieve an average inference speed boost of 53% and lower working memory consumption by 53%. Overall, the proposed model shows promising results for computation under low resource settings without compromising accuracy.

Post-training Optimization Through Joint RL-NSGA-II Training: Following the introduction of the layer skipping method, this work analyzes the models and identifies key limitations. Then the joint RL-NSGA-II training mechanism is proposed to alleviate the limitations, demonstrating further improvements in efficiency and providing control over the *accuracy verses further-efficiency* trade-off.

1.7 Scope

The scope of this work is only on the Vision Transformer [3]. This is because Transformers have gained significant interest in the research as well as the industry settings due to their versatility across different problem domains. We limit the work to the vision domain, simply due to time constraints and the computation budget available. With regards to the dataset, this work focuses on three standard and publicly available datasets namely: CIFAR-10/100 [1] and Tiny-Imagenet [2].

1.8 Thesis Structure

The rest of this document discusses the formulation of the proposed technique. Chapter 2 builds the foundations of deep learning in subsection 2.1. It further explains the various ways of making deep learning models efficient by focusing on Transformers under subsection 2.2. Finally, it presents the related works that are closely related to this work both in technique and objective in subsection 2.3.

Chapter 3 discusses the methodology proposed. Subsection 3.4 formulates the reinforcement learning agent-based dynamic transformer network. Subsection 3.5 then points out three weaknesses of this model and proposes a method to mitigate them. Following that, Chapter 4 presents the experimentation and the findings. Subsection 3.4 presents the models and the algorithm used to train them. Then subsections 4.2 & 4.4 discuss the various findings.

Finally, Chapter 5 concludes the whole study by highlighting key contributions and recommendations.

Chapter 2 Literature Review

2.1 Foundations

This section will discuss the important foundations that are needed for the proposed solution.

2.1.1 Deep Learning

2.1.1.1 Perceptron

Deep learning is a mechanism where a learning system propagates signals through multiple layers where each layer uncovers important features where, eventually, these features are used to predict an output [5]. Such a model can be constructed with a neural network. The initial attempt to create such a model was the perceptron [71]. Perceptron is a linear classifier that uses a thresholding function denoted as $H(x; w, b)$, which is defined as:

$$H(x; w, b) = \begin{cases} 1, & \text{if } w \cdot x + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

where, $H(x; w, b)$ represents the output of the perceptron for a given input vector x, weight vector w, and bias term b. The perceptron calculates the weighted sum of the input vector and bias term $(w \cdot x + b)$ and compares it to 0 [71]. If the result is greater than 0, the perceptron outputs 1; otherwise, it outputs 0 [71].

One of the main problems with the perceptron algorithm is if the data cannot be separated by a straight line or plane, it will not be able to accurately classify it[72]. Minsky and Papert [72] demonstrated that the perceptron, and similar linear classifiers, are unable to solve problems that require non-linear decision boundaries, such as the XOR problem. This was a significant result, as it helped to inspire the development of multi-layer neural networks.

2.1.1.2 Multi-layer Perceptron (MLP)

The multi-layer perceptron (MLP) represents an advancement over the original perceptron in terms of its capabilities and learning abilities [73, 5]. While the original perceptron was a single-layer neural network, the MLP introduces multiple layers of interconnected neurons, offering enhanced computational power and the ability to learn complex patterns and relationships [74, 75, 76].

The perceptron was only capable of linearly separable classification tasks, as it employed a simple step function activation and could only solve linearly separable problems [72]. In contrast, the MLP overcomes this limitation by incorporating nonlinear activation functions, such as sigmoid [76], hyperbolic tangent [77] or ReLU [78], in its hidden layers. These nonlinear activation functions introduce nonlinearity into the model, enabling it to learn non-linear boundaries [75]. Additionally, the MLP employs a layered architecture, where each layer is composed of multiple neurons. The neurons in each layer are interconnected with weights, allowing information to flow through the network in a feed-forward manner [5]. This layered structure enables the MLP to learn complex functions [75, 5].

The MLP utilizes the backpropagation algorithm [79] to optimize the parameters. Backpropagation is an efficient algorithm that allows the MLP to adjust the weights of its connections based on the discrepancy between the predicted output and the desired output [5]. By iteratively updating the weights using the error signal propagated from the output layer to the input layer, the MLP can learn to improve its predictions and minimize the overall error through gradient descent [5].

Overall *Deep learning* refers to training neural networks with multiple layers of certain structures, of which multi-layer perceptron is one [5]. Through the use of multiple hidden layers, deep learning models are capable of automatically learning hierarchical representations of data [25, 46], enabling them to extract intricate patterns and features from complex

datasets. The depth of the network allows for the extraction of increasingly abstract and high-level features as information passes through each layer, contributing to the power and effectiveness of deep learning models in solving complex tasks [80]. This hierarchical representation learning, combined with the ability to train deep networks using large amounts of data, allows deep learning models to achieve state-of-the-art performance in various domains [26, 3]. Apart from the MLP, other kinds of layers can be employed. The following subsections will discuss some of them.

2.1.1.3 Convolutional Neural Networks (CNN)

Although MLPs are powerful at learning complex functions, they struggle to effectively process structured data due to their inherent design [25]. For example, when confronted with image classification tasks, MLPs encounter two primary issues. Firstly, MLPs treat each input feature as independent, disregarding the spatial relationships present in images [25]. This limitation hampers their ability to exploit the locality and hierarchical structure found within visual data. Consequently, the models are incapable of capturing translational invariance, i.e., recognizing an object regardless of its location within the image [25]. Secondly, MLPs suffer from an explosion in the number of parameters when processing high-dimensional inputs such as images [25]. Since each neuron in a fully connected layer is connected to every neuron in the preceding layer, the model becomes increasingly prone to overfitting and computationally expensive as the input dimensionality grows [25].

Convolutional Neural Networks [25] were specifically designed to overcome the limitations of MLPs in processing structured data. CNNs incorporate a specialized architecture that allows them to efficiently learn and exploit the spatial and hierarchical structure present in images [25]. At a fundamental level, CNNs comprise three key components: convolutional layers, pooling layers, and fully connected layers [25]. Convolutional layers are responsible for learning local patterns or features within the input data [25]. These layers employ a set of learnable filters that slide over the input, performing element-wise multiplications and aggregating the results [25]. This process enables the network to detect and capture local

patterns, such as edges or textures, while preserving spatial relationships [25, 80]. A 2D convolution operation involves applying a filter or kernel to an input image to extract relevant features. Let I be the input image and K be the convolutional kernel. The 2D convolution operation is defined as follows [5]:

$$\text{Output}[i, j] = \sum_{m} \sum_{n} I[i + m, j + n] \cdot K[m, n] \tag{2.2}$$

Here, i and j represent the spatial coordinates of the output feature map, and m and n represent the spatial coordinates of the kernel. The output feature map is obtained by sliding the kernel over the input image, performing element-wise multiplications between the corresponding elements of the kernel and the overlapped image region, and summing them up[25]. The 2D convolution operation can be efficiently implemented using matrix operations, such as the dot product between the input image patch and the flattened kernel weights [25].

Pooling layers or *sub-sampling* [25] follow the convolutional layers and aim to down-sample the learned features, reducing their spatial dimensions. By summarizing the information present in the feature maps, pooling layers enhance the model's spatial invariance and computational efficiency [5]. Lastly, fully connected layers, akin to those in MLPs, perform high-level feature learning and make predictions based on the learned features [81]. These layers aggregate the spatially encoded information into a compact representation, which can be further utilized for classification, regression, or retrieval tasks [13].

2.1.1.4 Recurrent Neural Networks (RNNs)

While spatial data can be handled using CNNs, sequential datasets require a different way of treatment [5]. Recurrent Neural Networks (RNNs) are a class of deep learning models specifically designed to handle sequential data by incorporating feedback connections within the network [82, 23, 24]. RNNs consist of recurrently connected nodes that maintain an internal state or memory, allowing information to persist across time steps [5]. Each node

in the network receives an input, produces an output, and updates its internal state based on the previous state and the current input [82].

The key feature of RNNs is their ability to process inputs of arbitrary length and capture dependencies across different time steps [23]. This is achieved by incorporating the output of a previous time step as input to the current time step, creating a feedback loop within the network [23, 24]. The equation of the Vanilla or basic RNN [82] is given by:

$$h_t = \sigma(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h) \tag{2.3}$$

where h_t represents the hidden state at time step t, x_t represents the input at time step t, and σ is the activation function (typically a non-linear function like the sigmoid or hyperbolic tangent). The matrices W_{hh} and W_{xh} are the weight matrices for the recurrent and input connections, respectively, and b_h is the bias term [82]. The equation can be intuitively understood as follows: the hidden state h_t at the current time step is computed by combining the previous hidden state h_{t-1}, the current input x_t, and the bias term b_h, using the weight matrices W_{hh} and W_{xh} [5]. The activation function σ introduces non-linearities to capture complex dependencies in the sequential data [5].

This RNN model in eq (3.2) has several limitations that can hinder its performance in certain tasks. One major limitation is the vanishing gradient problem [83], where gradients diminish exponentially as they propagate through time, leading to difficulties in capturing long-term dependencies. This makes it challenging for plain RNNs to effectively model and remember information over long sequences [5]. Additionally, plain RNNs are not capable of selectively forgetting or updating information, as they lack explicit mechanisms to control the flow of information [84]. This can result in the accumulation of irrelevant or outdated information, which can negatively impact the model's ability to make accurate predictions [5]. These limitations have motivated the development of more advanced architectures, such

as Long Short-Term Memory(LSTM) [23] and Gated Recurrent Unit(GRU) [24], which address these issues and have proven more effective in modeling sequential data.

LSTM [23] is a type of RNN architecture that addresses the vanishing gradient problem in traditional RNNs and is particularly effective in modeling and processing long-term dependencies in sequential data. LSTM networks are widely used in various tasks that employ sequences such as natural language processing [85], speech recognition [86], and time series analysis [87]. The key idea behind LSTM is the introduction of memory cells, which allow the network to remember or forget information over multiple time steps selectively [23]. The LSTM architecture consists of multiple memory cells, each equipped with three main components: the input gate, the forget gate, and the output gate. The following shows the LSTM mathematical definition [23]:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{(Forget gate)} \tag{2.4}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \text{(Input gate)} \tag{2.5}$$

$$\tilde{C}t = \tanh(W_C \cdot [ht - 1, x_t] + b_C) \quad \text{(Candidate cell state)} \tag{2.6}$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}t \quad \text{(Cell state)} \tag{2.7}$$

$$o_t = \sigma(W_o \cdot [ht - 1, x_t] + b_o) \quad \text{(Output gate)} \tag{2.8}$$

$$h_t = o_t \cdot \tanh(C_t) \quad \text{(Output)} \tag{2.9}$$

the LSTM cell consists of several key components. The forget gate f_t determines which information from the previous cell state C_{t-1} should be discarded, based on the combination of the previous hidden state h_{t-1} and the current input x_t [23]. The input gate i_t determines the relevance of the new information from the input x_t [23]. The candidate cell state $\tilde{C}_t$ represents the new information that could be added to the cell state[23]. The updated cell state C_t is a combination of the previous cell state and the new candidate cell state, controlled by the forget gate and the input gate[23]. The output gate o_t determines which parts of the cell state should be exposed as the output [23]. Finally, the hidden state h_t is computed by applying the output gate to the cell state after passing it through a hyperbolic tangent

function[23]. The LSTM architecture addresses the vanishing gradient problem and allows for long-term memory storage and controlled information flow, enabling effective modeling of sequential data as shown by [23, 24].

2.1.2 Sequence to Sequence Models (Seq2Seq)

RNNs are designed for multiple inputs and a single output scenario [23]. However, many real-world problems involve variable-length sequence outputs, such as machine translation [88], text summarization [27], speech recognition [86], and more. The sequence-to-sequence [85] model addresses this challenge by introducing a two-step process: an encoder-decoder architecture. The encoder processes the input sequence and captures its essential information, transforming it into a fixed-length representation called a context vector or latent representation[5]. This context vector serves as a compressed representation of the input sequence's meaning and captures its semantic and contextual information [85, 89].

The decoder then takes this context vector as input and generates the output sequence step by step, one element at a time [85]. At each step, the decoder predicts the next element based on the previously generated elements and the context vector. The decoder can be an RNN, typically using a type of RNN called Long short-term memory (LSTM), to capture dependencies and generate output elements sequentially [23, 85, 89]. The following equations show the formulation of the Sequence-to-Sequence architecture[85].

$$\text{Encoder:} \quad h = \text{Encoder}(x_{1:S}) \tag{2.10}$$

$$\text{Decoder:} \quad p(y_{1:T}|x_{1:S}) = \prod_{t=1}^{T} p(y_t|y_{<t}, h, x_{1:S}) \tag{2.11}$$

where $x_{1:S}$ denotes the input sequence of length S, $y_{1:T}$ denotes the output sequence of length T, h represents the hidden representation produced by the encoder, $p(y_t|y_{<t}, h, x_{1:S})$ represents the conditional probability of generating the $t-th$ output token given $y_{<t}$, h, and $x_{1:S}$ [85].

The sequence-to-sequence model allows for the handling of variable-length input and output sequences, making it applicable to a wide range of tasks[85]. Leveraging the encoder-decoder architecture enables the model to learn meaningful representations of the input sequence and generate output sequences that preserve the desired characteristics or semantics[5]. This approach has revolutionized many areas of natural language processing, speech recognition [90, 91, 92], and other sequence-related tasks [93, 94], enabling the development of more accurate and context-aware models.

2.1.3 Attention

The Seq2Seq (Sequence-to-Sequence) [85] model revolutionized various natural language processing (NLP) tasks, such as machine translation [88] and text summarization [95], by utilizing RNNs to process sequential input. However, traditional Seq2Seq models faced challenges when dealing with long sequences[88]. The fixed-length context vector produced by the encoder RNN was expected to contain all the relevant information from the source sequence, making it difficult to capture the dependencies between distant words[88]. As such [88] introduced the Attention mechanism to mitigate this limitation.

Attention enhances the Seq2Seq model by allowing it to focus on different parts of the input sequence dynamically [88]. Rather than relying solely on a fixed-length context vector, attention mechanisms enable the decoder to selectively attend to specific parts of the source sequence during the decoding process[88]. This attention mechanism ensures that the model assigns appropriate weights to different input elements, giving more importance to relevant information [88]. Attention mechanisms operate in the context of an encoder-decoder architecture[88].

The attention mechanism can be represented mathematically as[88]:

$$\text{Encoder:} \quad h_1, h_2, \ldots, h_S = \text{Encoder}(x_{1:S}) \tag{2.12}$$

$$\text{Attention:} \quad e_{t,s} = \text{score}(h_s, \tilde{h}_t) \tag{2.13}$$

$$\text{Context Vector:} \quad \alpha_{t,s} = \text{softmax}(e_{t,s}) \tag{2.14}$$

$$\text{Decoder:} \quad \hat{y}_t = \text{Decoder}(y_{t-1}, \tilde{h}_t) \tag{2.15}$$

$$\text{Output:} \quad p(y_t|y_{<t}, x_{1:S}) = \text{softmax}(\hat{y}_t) \tag{2.16}$$

$$\text{where} \quad \tilde{h}_t = \sum_{s=1}^{S} \alpha_{t,s} h_s \tag{2.17}$$

where $x_{1:S}$ denotes the input sequence of length S, y_{t-1} represents the previously generated token at time step $t-1$, h_s represents the hidden representation of the S-th input token, $e_{t,s}$ denotes the alignment score between $\tilde{h}_t$, h_s $\alpha_{t,s}$ represents the attention weight for aligning $\tilde{h}_t$ with h_s, $\tilde{h}_t$ denotes the context vector at time step t, computed as the weighted sum of encoder hidden states, $\hat{y}_t$ represents the predicted output at time step t, $p(y_t|y_{<t}, x_{1:S})$ denotes the probability of generating the t-th output token given the previous tokens and the input sequence [88].

It has been shown that Attention is able to effectively capture the alignment between source and target sentences without relying on a fixed-length vector representation [88]. They demonstrated the ability of the Attention mechanism to automatically learn relevant parts of the source sentence during translation [88]. Their experimental result showed that the joint learning of alignment and translation in Neural Machine Translation (NMT) can yield a better representation of sequence alignment and improve upon the existing approaches in machine translation such as plain Seq2Seq [85].

2.1.4 Transformer Network

The Seq2Seq-Attention architecture [88, 85] has one major limitation which is the serial processing of sequences and therefore, it could not take the parallelizability advantage of recent Graphical Processing Units(GPU). Furthermore, as sequence length increases, the

performance of such a model drops [96]. The Transformer was motivated by the need for a more efficient and effective way to capture long-range dependencies in sequences [19].

The key insight behind the Transformer model is the attention mechanism, which allows the model to focus on different parts of the input sequence when processing each element [19]. This attention mechanism enables the model to capture dependencies between words or tokens in a sequence more effectively, even when they are far apart [97].

The Transformer, just like Seq2Seq [85], has two parts - Encoder and decoder [19]. In addition, it has input embedding and positional embedding. Furthermore, Transformer based models are *isomorphic*, which are models that have layers with the same size unlike ResNet[46]. ResNets have layers with decreasing size as the level of the layer increases. The following subsections (from 2.1.4.1 to 2.1.4.4) explain the details of the original Transformer architecture by [19].

2.1.4.1 Input Embedding

In the *Input Embedding*, the input sequence is first transformed into fixed-dimensional embeddings. Each word or token in the input sequence is represented as a vector, capturing its semantic meaning [98, 99]. These embeddings are learned during the training process. In the *Positional Encoding*, since the Transformer does not have any inherent notion of word order, positional encoding is introduced to provide information about the position of each token in the input sequence [19]. Positional encodings [19] are added to the input embeddings, allowing the model to consider the sequential information.

The input embedding matrix $X \in R^{NxD}$ can be represented as follows where N is the length of the sequence and D is the embedding size:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

Positional Encoding: The positional encoding matrix PE can be represented as follows:

$$PE = \begin{bmatrix} PE_1 \\ PE_2 \\ \vdots \\ PE_N \end{bmatrix}$$

Each position i in the sequence is associated with a positional encoding vector PE_i [19]. The positional encoding vector for a given position i can be computed as follows:

$$PE_i = \begin{bmatrix} \text{pos\_enc}(i, 1) \\ \text{pos\_enc}(i, 2) \\ \vdots \\ \text{pos\_enc}(i, d) \end{bmatrix}$$

Here, d is the dimensionality of the positional encoding vector, and $pos\_enc(i, j)$ represents the value at position (i, j)in the positional encoding matrix. The specific formula for calculating $pos\_enc(i, j)$ depends on the chosen positional encoding scheme [19]. For example, one commonly used approach is based on trigonometric functions like the sine and cosine functions [19]. An other well-known method is to learn these embeddings [3].

Both the encoder and the decoder are built from the same components explained below:

2.1.4.2 Self-Attention

This is the core component of each layer. The self-attention mechanism allows the model to capture dependencies between different words in the input sequence [19]. It computes attention weights for each word based on its interactions with other words in the same sequence. Self-attention involves three types of inputs: queries, keys, and values [19]. The self-attention mechanism calculates the attention scores by computing the dot product between queries and keys and applying a softmax function to obtain the attention weights. These attention weights are then used to compute a weighted sum of the values, which forms the output of the self-attention mechanism [19].

$$\text{Attention Scores} = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) \tag{2.18}$$

Here, the attention scores represent the relevance or similarity between the query vectors (Q) and the key vectors (K). The dot product between the linearly transformed query vector ($Q = XW_Q$) and the linearly transformed key vectors ($K = XW_K$) is divided by the square root of the dimensionality of the query/key vectors (d_k) to scale the dot products [19]. The softmax function is then applied to obtain normalized attention scores that sum up to 1, representing the weights for each word's contribution to the self-attention output [19]. The linear transformations are performed using the weight matrices W_Q, W_K, and W_V with appropriate sizes. The input sequence X has a size of (sequence_length, embedding_size), and the weight matrices have sizes (embedding_size, d_k) for W_Q and W_K, and (embedding_size, d_v) for W_V, where d_k and d_v are the desired dimensions for the query/key and value vectors, respectively [19].

$$\text{Self-Attention Output} = \text{Attention Scores} \cdot V \tag{2.19}$$

Here, the Self-Attention Output represents the weighted sum of the value vectors (V) using the attention scores [19]. The attention scores, obtained from the softmax-normalized dot product between the query vectors (Q) and the key vectors (K), determine the weights assigned to each value vector [19]. The matrix multiplication of the attention scores with the value vectors results in a weighted sum of the values. The attention scores have a size of

(sequence_length, sequence_length), representing the pairwise relevance between each word in the input sequence. The value vectors (V) have a size of (sequence_length, d_v), where d_v is the dimensionality of the value vectors [19].

2.1.4.3 Multi-head attention

The outputs of all attention heads are concatenated along the last dimension and linearly transformed to obtain the final multi-head attention output [19].

$$O = \text{Concat}(O'_1, O'_2, \ldots, O'_h)W_C \tag{2.20}$$

Here, W_C is another learnable weight matrix used for the concatenation and linear transformation step [19]. The final output O represents the result of multi-head attention, capturing different types of dependencies in parallel. It allows the model to learn diverse representations and capture various aspects of the input sequence.

2.1.4.4 Feed-forward Network (FFN)

In the Transformer model, the basic feed-forward network (FFN) is a position-wise fully connected neural network applied to each position independently in the sequence [19]. It consists of two linear transformations followed by a non-linear activation function, such as the ReLU (Rectified Linear Unit) [78, 19]. Mathematically, given an input X of size (sequence_length, d_{model}), where d_{model} is the dimensionality of the input, the FFN computes the output Y as:

$$Y = \text{ReLU}(XW_1 + b_1)W_2 + b_2 \tag{2.21}$$

Here, W_1 and W_2 are learnable weight matrices, and b_1 and b_2 are bias vectors. The ReLU activation function is applied element-wise. This formulation allows the FFN to perform non-linear transformations on each position independently, enabling the model to capture complex relationships within the sequence [19].

2.1.4.5 Transformer Layer

$$\text{Multi-head Attention} : \text{Output}_1 = \text{MultiheadAttention}(\text{Input}, \text{Input}) \quad (2.22)$$

$$\text{Residual and Layer Normalization} : \text{Output}_2 = \text{LayerNorm}(\text{Input} + \text{Output}_1) \quad (2.23)$$

$$\text{Feed-Forward Network} : \text{Output}_3 = \text{FFN}(\text{Output}_2) \quad (2.24)$$

$$\text{Residual and Layer Normalization} : \text{Output}_4 = \text{LayerNorm}(\text{Output}_2 + \text{Output}_3) \quad (2.25)$$

Here, $Input$ represents the input to the Transformer layer. The $MultiheadAttention$ function performs multi-head attention, generating $Output_1$. The FFN function represents the feed-forward network, transforming $Output_2$ to $Output_3$. The $LayerNorm$ function denotes the layer normalization operation. The residual connections [46] add the original input with the corresponding outputs and the layer normalization [100] applies normalization to the summed outputs. The final output is represented as $Output_4$ [19]. There are various deep-learning architectures in the literature and enumerating all of them requires significant space. However, Figure 2 shows some of these architectures along with their derivatives.

2.1.5 Dynamic Neural Networks

Dynamic neural networks refer to a type of neural network that can adapt its structure or parameters based on different inputs, providing advantages in terms of accuracy, computational efficiency, and adaptiveness [64]. Unlike static models with fixed computational graphs and parameters at the inference stage, dynamic networks have the ability to adjust their architecture dynamically to handle varying input data [64]. These networks have applications in various domains, including computer vision problems [68, 101], decision-making [61], and adaptive inference [69].

There are different types of dynamic neural networks based on their focus and problem domains. Sample Wise Dynamic Networks [102, 69] allocate network resources to individual samples, which is useful for tasks like anomaly detection. Spatial Wise Dynamic Networks [103], on the other hand, are designed for computer vision problems and adapt their struc-

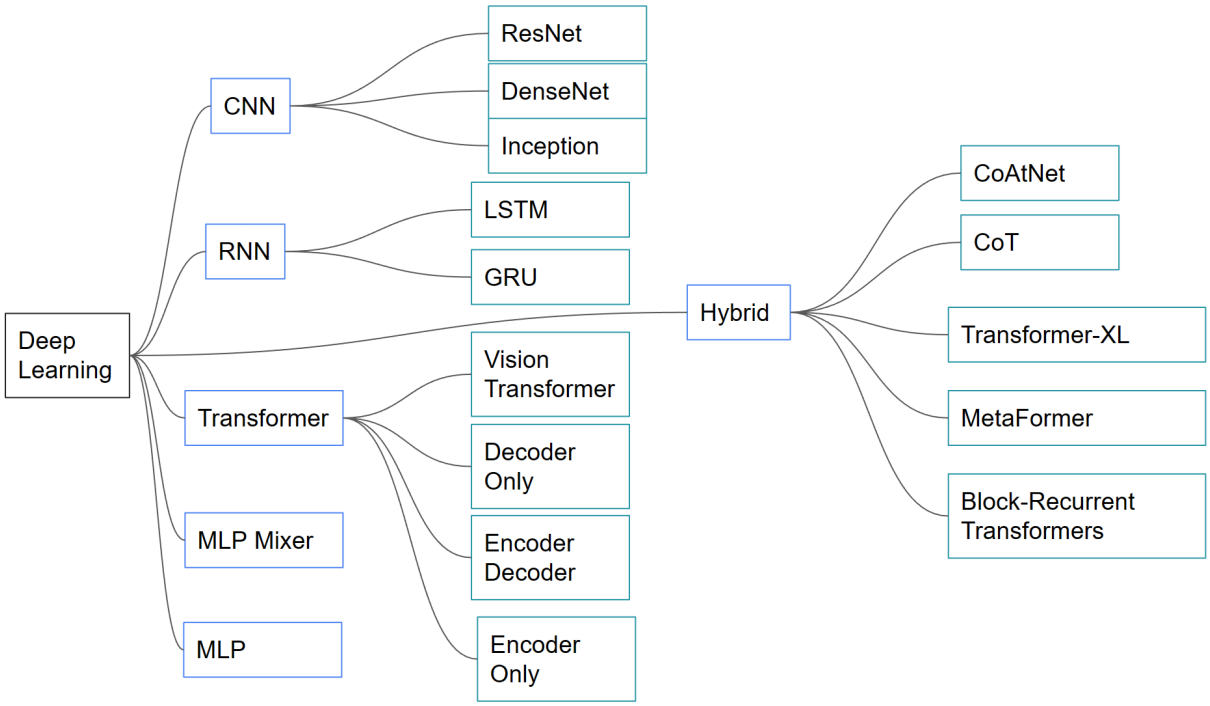


Figure 2: Deep Learning Architecture Family. All models rely on MLP as an internal structure, although implementation might differ in someways. Hybrid systems combine different elements from different architectures.

ture to handle varying spatial patterns in images. Overall, dynamic neural networks are new, diverse, and a promising area of research in deep learning [64]. They provide the ability to adapt and optimize network structures based on input data, leading to improved performance and flexibility in various applications.

2.1.6 Sparse and adaptive computation

Sparse computation involves selectively activating or processing only a subset of the available computational units while disregarding the rest. As such, pruning [104], low-rank approximation [76], and similar strategies are collectively called *sparsification* [105, 106, 58]. This strategy is particularly useful when the input data exhibits sparsity, meaning that a large portion of the input is irrelevant or contains redundant information. By concentrating computational resources on the relevant parts of the input, sparse computation reduces compu-

tational complexity, memory requirements, and energy consumption [106, 107, 105, 108, 109].

Adaptive computation, on the other hand, refers to dynamically adjusting the computational strategy or model parameters based on the input data characteristics or changing conditions [68, 110, 101]. This adaptability enables computational systems to tailor their behavior or structure to match the specific requirements of the current task or environment. Such techniques have been shown to be effective in addressing the high compute demand of deep learning models [111, 112, 69, 62]

Overall, the relationship between adaptive computation and sparse models can be characterized by the utilization of adaptive techniques to identify, exploit, or adjust sparsity in models and data [68, 62, 57]. Adaptive computation helps optimize the use of computational resources and algorithms based on the sparsity patterns present, leading to improved efficiency and interoperability [69].

2.1.7 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that focuses on developing intelligent agents capable of making optimal decisions through interactions with an environment [113]. RL agents learn by trial and error, using feedback in the form of rewards or penalties to improve their decision-making policies [113]. The main notion of RL is an agent that takes actions in an environment to maximize a notion of cumulative reward [113]. The agent interacts with the environment, observes its state, takes actions, and receives rewards as feedback [113]. The goal is to learn a policy—a mapping from states to actions—that maximizes the expected cumulative reward over time [113].

One widely used RL algorithm is Q-learning, which falls under the category of value-based methods [114, 113]. Q-learning learns an action-value function (Q-function) that estimates the expected cumulative reward for taking a specific action in a given state [114]. By itera-

tively updating the Q-function based on the observed rewards and the estimated values of the next state, Q-learning converges to an optimal policy [113].

Another popular algorithm is policy gradient [115, 113], which belongs to the class of policy-based methods. Policy gradient directly learns the policy without explicitly estimating the value function [115, 113]. It uses gradient ascent to update the policy parameters, seeking to maximize the expected cumulative reward [115, 113]. Policy gradient methods offer flexibility in handling continuous action spaces and are well-suited for problems with stochastic policies [115].

Actor-Critic algorithms combine the advantages of both value-based and policy-based methods [116]. They maintain both a value function (the critic) and a policy (the actor). The critic provides value estimates, which guide the actor's policy updates. This combination allows for more stable learning and improved convergence [115].

Proximal Policy Optimization (PPO) is a state-of-the-art policy optimization algorithm [117]. PPO focuses on updating the policy in a way that avoids large policy updates, which can lead to unstable learning. By enforcing a constraint on the policy update using a surrogate objective function, PPO ensures more stable and gradual policy improvements [117].

Deep Q-Networks (DQN) leverage deep neural networks to approximate the Q-function in high-dimensional state spaces [118]. DQN combines Q-learning with function approximation, allowing RL agents to handle complex and continuous state representations. DQN utilizes experience replay—a technique that stores and samples past experiences—to improve learning stability and efficiency [117, 113].

In addition to these algorithms, there are several extensions and variations in reinforcement learning, such as Monte Carlo methods [119], temporal difference learning [114], and

exploration-exploitation strategies like epsilon-greedy [120] and Upper Confidence Bound [121]. In this work, as subsection 3.4.1 explains, *REINFORCE with baseline* [115] algorithm is employed to train the agent. The following subsection discusses the algorithm in detail.

2.1.7.1 REINFORCE with baseline

REINFORCE enables agents to learn parameterized policies directly by optimizing them to maximize the expected cumulative reward [122]. The algorithm follows an episodic approach, where it collects trajectories by interacting with the environment. After each episode, REINFORCE performs updates to the policy parameters based on the observed rewards [115]. The key idea behind REINFORCE is to estimate the gradient of the expected cumulative reward with respect to the policy parameters, and use this gradient to update the policy in a way that increases the likelihood of actions that lead to higher rewards [122]. This is achieved by computing the policy gradient through the Monte Carlo sampling method, where the observed returns from each trajectory are used as unbiased estimates of the expected cumulative reward [115]. The algorithm iteratively improves the policy by repeatedly sampling trajectories, estimating the policy gradient [82], and updating the policy parameters in the direction of the gradient [122]. The following specifies its mathematical foundation:

Let's denote the policy parameters as θ and the policy itself as $\pi(a|s;\theta)$, which represents the probability of taking action a in state s given the parameters θ [122]. The expected cumulative reward for a trajectory τ following policy π can be written as eq (2.26) [115]:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right] \tag{2.26}$$

where $R(\tau)$ represents the cumulative reward obtained along the trajectory τ. To update the policy parameters θ using the policy gradient, the gradient of the expected cumulative reward is computed using eq(2.27) [122]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta)\cdot R(\tau)\right] \tag{2.27}$$

where T is the length of the trajectory, and $\nabla_\theta \log \pi(a_t|s_t;\theta)$ represents the gradient of the logarithm of the policy with respect to the parameters [122]. The REINFORCE update rule can be expressed using eq (2.28):

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla_\theta J(\theta) \tag{2.28}$$

where α is the learning rate.

This plain REINFORCE algorithm [122], while effective in some cases, has certain limitations. One limitation is the high variance in gradient estimates, which can lead to slow convergence and unstable learning [115]. Additionally, the plain REINFORCE algorithm does not utilize information about the quality of different actions, making it difficult to distinguish between good and bad actions [115, 113].

To address these limitations, the REINFORCE algorithm can be enhanced by incorporating a *baseline* function, usually referred to as *REINFORCE with baseline* [115]. The *baseline* is a learned function that provides an estimate of the expected cumulative reward in a given state, independent of the chosen action [115]. The baseline helps to reduce the variance in gradient estimates and provides a more accurate measure of the action's contribution to the overall reward [115]. Let's denote the baseline function as $B(s)$, which estimates the expected cumulative reward in state s. The updated gradient calculation with the baseline [115] can be expressed using eq (2.29):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi(a_t|s_t;\theta) \cdot (R(\tau) - B(s_t))\right] \tag{2.29}$$

By subtracting the baseline from the cumulative reward, the variance in the gradient estimates is reduced, as it focuses on the relative advantage of action over the baseline estimate [115, 113]. This makes the learning process more stable and efficient [123, 115]. It further laid the foundation for algorithms like PPO [117].

2.1.8 Multi-Objective Optimization

Multi-objective optimization (MOO) is a computational approach that deals with optimizing multiple conflicting objectives simultaneously [124, 125]. Unlike traditional single-objective optimization, where a single optimal solution is sought, MOO aims to find a set of solutions known as the Pareto front, which represents the trade-off between the different objectives [124, 125]. These objectives can be diverse, such as maximizing performance, minimizing cost, and reducing energy consumption [126]. MOO algorithms explore the solution space to identify a range of solutions that offer different trade-offs between the objectives, providing decision-makers with a set of options to choose from [125]. It is particularly valuable in complex real-world problems where there is no single optimal solution, as it enables decision-makers to make informed decisions based on a range of desirable outcomes [124].

The Non-dominated sorting genetic algorithms (NSGA) algorithms operate on the principle of non-dominated sorting, which involves organizing individuals into a hierarchical structure based on their dominance relationships as illustrated in Algorithm 1 [124, 127, 128]. Individuals that are not dominated by any other individuals are referred to as Pareto optimal individuals [127, 128]. The NSGA algorithms utilize non-dominated sorting to detect Pareto optimal individuals and subsequently choose them for the process of reproduction [127, 128].

In this work, we apply a late-stage optimization on the proposed layer-skipping network to improve it even more through a genetic algorithm. Specifically, we use the Non-dominated sorting genetic algorithms II (NSGA-II) [127]. NSGA-II and a more recent updated version of this algorithm, NSGA-III [128] have been employed in various deep learning models for further pruning [110, 129, 130]. However, we chose NSGA-II since it does not require a reference direction, which guides NSGA-III into a set of solution spaces [128]. In this work, since the goal is to explore various accuracy-speed trade-off options in the post-training stage and there is no preferred reference direction, we applied NSGA-II [127]. Algorithm 1 shows the high-level steps of the NSGA-II algorithm [127].

Algorithm 1 Non-dominated Sorting Genetic Algorithm II (NSGA-II) [127]

```
procedure NSGA-II(Population)
    Initialize the population
    while Not converged do
        Sample a batch of images B
        Evaluate the fitness of each individual in P on B
        Perform a non-dominated sorting
        Apply the crowding distance operator
        Select the parents for the next generation
        Generate the offspring population
        Replace the old population with the new population
    end while
    Return the best solution
end procedure
```

Algorithm 1 shows the major pseudocode of the NSGA-II [127] algorithm. It takes a list of chromosomes, usually referred to as *Population*, and evolves them into a population that maximizes or minimizes the objective functions. Then it results in a list of options that offer various degrees of desired properties that compromise one another with all points of the highest quality achievable.

The Hypervolume [131, 132] metric is a performance measure used in multi-objective optimization problems to evaluate the quality and diversity of a set of solutions. It quantifies the volume of the objective space enclosed by a set of solutions, with larger hypervolumes indicating better performance [131]. The hypervolume metric considers both the closeness of solutions to the desired objectives and their spread across the objective space [131]. By comparing the hypervolume values of different solution sets, it provides a way to assess and compare the performance of different optimization algorithms or approaches [132]. This indicator was selected to evaluate the Pareto fronts in this work since it preserves dominance

properties and distribution without relying on knowledge of the Pareto front [132]. Furthermore, it is effective on a small number of objective functions when compared to alternative performance indicators [133].

2.2 Efficient Methods in Deep Learning

Numerous techniques have been proposed to improve either compute efficiency or memory efficiency, and they encompass both training and deployment stages [134]. Among the proposed techniques are Knowledge Distillation [56], Weight Pruning [135, 66], Low-Rank Approximation [136], Sparsification [137], Binarization [138], and Quantization [139]. Each of these approaches contributes to optimizing the overall efficiency of the system. The following are some of the methods that are directly or indirectly related to this work with regard to efficiency in Transformers.

2.2.1 Weight pruning

Weight pruning [66] is a technique used in neural networks to reduce the complexity and size of the model by removing unnecessary connections between neurons. It involves identifying and eliminating weights with low magnitudes or near-zero values, effectively *pruning* them from the network [66]. This process helps compress the model, reduce memory requirements, and improve computational efficiency during training and inference [66]. Pruning can be done using various criteria and here are some of the well-known methods:

- Magnitude-based weight pruning methods identify and remove weights with small magnitudes [140, 104]. This can be done by thresholding the magnitudes of the weights or by using a greedy algorithm to iteratively remove weights [140].

- Structure-based weight pruning methods identify and remove entire layers or groups of weights [141, 142]. This can be done by using a clustering algorithm to group weights that are similar to each other or by using a reinforcement learning algorithm to learn a policy for pruning weights [141].

- Filter-based weight pruning methods identify and remove entire filters [143, 144]. This can be done by thresholding the activations of the filters or by using a greedy algorithm to remove filters iteratively [143].

- Threshold-based pruning sets a threshold value below which weights are discarded, while magnitude-based pruning ranks the weights by their magnitude and removes the ones with the smallest magnitude [145].

Zhang et al. [48] addressed the challenge of reducing the size of large Transformer models while maintaining their performance. It incorporates the concept of upper confidence bound (UCB)[121] to capture the uncertainty of importance scores, preventing the premature pruning of crucial weights. Their experimental results show that PLATON outperforms existing methods, achieving significant model size reduction while preserving accuracy and enabling the deployment of large Transformer models in real-world applications with improved efficiency [48]. Kwon et al. [146] proposed a fast post-training pruning framework to address the high inference cost of Transformer models. Their structured sparsity-based approach enables pruning without the need for retraining, overcoming the limitations of previous methods [146]. The framework achieves significant reductions in floating-point operations (FLOPs) and inference latency while maintaining high accuracy, making it a practical solution for reducing inference costs in Transformers [146].

Zhu et al. [47] proposed a three-step pipeline for pruning vision transformers to address the challenges of deploying them on mobile devices with limited resources. Their methodology involves training the model with sparsity regularization, pruning dimensions of linear projections based on importance scores, and fine-tuning the pruned model [47]. The results demonstrate that their pruning approach effectively reduces the storage, memory, and computational demands of vision transformers while maintaining competitive performance, making them suitable for deployment on resource-constrained mobile devices [47]. Frantar and Alistarh [147] introduced SparseGPT, a pruning method specifically designed for large-scale GPT models. SparseGPT enables efficient and accurate pruning without retraining, allowing GPT models to be pruned to at least 50% sparsity in one-shot while maintaining minimal loss of accuracy [147]. The method achieves high levels of unstructured sparsity, such as 60%, in models like OPT-175B and BLOOM-176B, resulting in significant compu-

tational savings during inference without sacrificing model performance [147].

2.2.2 Quantization

Quantization in deep learning is the process of reducing the precision of the weights and activations of a neural network [148]. This can be done by rounding the weights and activations to a lower number of bits [148]. Quantization can be used to reduce the memory footprint and computational cost of neural networks, making them more efficient to deploy on mobile and embedded devices [139, 148, 149, 150, 151]. Bondarenko et al. [152] addressed the challenges of quantizing transformers for efficient deployment on resource-limited devices in NLP tasks. The authors propose three solutions, including a novel quantization scheme called *per-embedding-group quantization* to overcome these challenges [152]. The methods are evaluated using the GLUE benchmark with BERT and achieve state-of-the-art results for post-training quantization, offering effective strategies for reducing memory footprint and latency in transformer architectures [152].

Liu et al. [50] addressed the challenge of deploying vision transformers on mobile devices by proposing an effective post-training quantization algorithm. The authors aim to reduce memory storage and computational costs associated with vision transformers while preserving their performance [50]. The proposed method introduces optimal quantization intervals, a ranking loss to maintain the functionality of the attention mechanism, and a mixed-precision quantization scheme, outperforming state-of-the-art techniques and enabling an efficient deployment of vision transformers on resource-constrained devices [50]. Yang et al. [153] introduced Dynamic Stashing Quantization (DSQ) as a dynamic quantization strategy to address the computational and memory challenges of training Large Language Models (LLMs) in Natural Language Processing (NLP) tasks [153]. DSQ significantly reduces the number of arithmetic and DRAM operations compared to the widely used 16-bit fixed-point format, making LLM training more efficient for on-device learning [153]. The experimental results demonstrate the effectiveness of DSQ in achieving computational savings and reduc-

ing memory operations, making LLMs more feasible for deployment in resource-constrained environments [153].

2.2.3 Knowledge distillation

Knowledge distillation [56, 154, 155] is a machine learning technique that transfers the knowledge from a large model, known as the teacher, to a smaller model, known as the student. The teacher model is typically trained on a large dataset, while the student model is trained on a smaller dataset [56]. The teacher model is used to generate soft labels for the student model, which are probability distributions over the possible classes. The student model then learns to predict the same labels as the teacher model [56]. The method has been used in various domains including image classification models [56, 156, 157, 158], natural language processing models [159, 160], and speech recognition models [161, 162, 163].

Touvron et al. [30] introduced a data-efficient approach for training image transformers by proposing a convolution-free transformer model trained solely on the ImageNet dataset [30]. They achieve competitive performance on ImageNet without using external data, demonstrating the effectiveness of their approach [30]. Additionally, the authors introduce a teacher-student strategy using a distillation token that allows the student model to learn from the teacher model through attention, yielding promising results and showcasing the potential of transformer models in image understanding tasks [30].

Wang et al. [164] addressed the problem of distilling knowledge from one self-supervised Vision Transformer (ViT) model to another [164]. They propose a method called AttnDistill, which directly distills information from the attention mechanism of the teacher model to the student model [164]. AttnDistill outperforms existing self-supervised knowledge distillation methods and achieves state-of-the-art results without requiring additional labeled data or complex architectures [164]. Jia et al. [165] focused on reducing the computational overhead of vision transformers for edge device deployment [165]. They propose a fine-grained

manifold distillation method that trains a compact student model to match a pre-trained teacher model at the patch level [165]. By dividing the manifold matching loss into three terms, they achieve high accuracy with a compact DeiT-Tiny model on the ImageNet-1k dataset [165].

2.2.4 Low-rank Approximation (Factorization)

Low-rank approximation or Factorization is a technique used in deep learning to reduce computational complexity and memory requirements [136]. It involves approximating weight matrices with lower-dimensional matrices of reduced rank, exploiting the redundancy in neural network layers [136]. Singular Value Decomposition (SVD) is a commonly used technique for low-rank approximation, where smaller singular values are truncated to obtain the approximation [166]. The low-rank approximation can be incorporated by directly applying decomposition to weight matrices or during model compression, where low-rank factors are learned or optimized [167, 136].

Cahyawijaya [53] focused on improving the computational and memory efficiency of transformer models through the use of low-rank approximation [53]. The proposed approach, called Greenformers, effectively reduces the size and computational costs associated with large-scale transformer models [53]. The results demonstrate that Greenformers enables faster training, reduced resource requirements, and mitigated environmental impact, addressing the challenges posed by the complexity and size of transformer models [53]. Winata et al. [168] introduced the low-rank transformer (LRT) as a solution for lightweight and efficient end-to-end speech recognition on portable devices [168]. The LRT model significantly reduces the number of network parameters by over 50% compared to the baseline transformer model, improving memory efficiency and speeding up training and inference processes [168]. Experimental results demonstrate that the LRT model achieves lower error rates on validation and test sets, outperforming existing works without the need for external language models or acoustic data, making it a practical solution for resource-constrained environments

[168].

2.2.5 Tranformer-specific Efficiency Methods

The ALBERT (A Lite BERT) model [169] is a modified version of the BERT model designed to improve language representation models in NLP. It shares the same architecture as BERT but incorporates parameter reduction techniques and introduces the sentence order prediction objective to enhance training speed and model understanding. Pre-trained on large datasets, ALBERT outperforms BERT on multiple language benchmark tests [169]. Reformer: The Reformer [170], an efficient variant of the Transformer model, addresses the computational challenges associated with training large models on long sequences. It introduces locality-sensitive hashing to replace dot-product attention, reducing computational complexity [170]. Additionally, reversible residual layers enable storing activations only once, significantly reducing memory usage [170]. The Reformer model achieves comparable performance to traditional Transformers but with improved memory efficiency and faster processing for long sequences [170].

Performers [37] are a type of Transformer architecture that addresses the limitations of traditional attention mechanisms. They offer efficient models with provable accuracy, without relying on assumptions like sparsity or low-rankness [37]. Performers utilize the FAVOR+ approach to approximate attention kernels, allowing for efficient modeling of attention mechanisms beyond softmax [37]. They are linear architectures compatible with regular Transformers, providing strong theoretical guarantees and offering improved efficiency and accuracy for large-scale tasks [37].

2.2.6 Summary

In the space of deep learning research, various methods have been developed to make such models efficient [134]. Table **??** summarizes the advantage and disadvantages of the well-studied methods and techniques.

Method	Advantage	Disadvantage
Knowledge Distillation	• Might reduce the model size [160] • Enables knowledge transfer [56] • Improves generalization [30]	• Requires a teacher model [56] • Loss of fine-grained details [171]
Weight Pruning	• Reduces the model size [66] • Speeds up inference [66] • Increases sparsity [66]	• Pruning process can be computationally expensive [172] • Fine-tuning may be required [160] • Accuracy drops if excessive pruning [54] • Minority classes might be affected severely [171]
Quantization	• Reduces memory and storage requirements [139] • Faster inference • Lowers energy consumption	• Quantization-aware training may be necessary [139] • Loss of performance on various metrics [171]
Low-Rank Approximation	• Reduces the model size [76] • Accelerates inference	• Accuracy drop if with excessive approximation [51]
Adaptive methods	• Faster inference [76] • Explainability [69]	• Exit and skip strategies are hard to design [64, 65] • Accuracy-speed trade-off [68]

Table 1: Summary of advantages and disadvantages of deep learning efficiency methods.

2.3 Related Works

Dynamic neural networks (DNNs) [64] have gained significant attention as an emerging research topic in deep learning. Unlike static models that possess fixed computational graphs and parameters during inference, dynamic networks have the ability to adapt their structures or parameters based on different inputs [89, 69]. This adaptability provides notable advantages in terms of accuracy, computational efficiency, and adaptiveness [64, 69].

There are various kinds of dynamic neural networks, often having significantly different titles and categories [64]. In the following section, networks that are closely related to this work are discussed:

2.3.1 Early Exiting

Early exiting in neural networks refers to a technique that improves the computational efficiency of DNNs by allowing the network to make predictions before reaching the final layers [65]. This technique is particularly useful for edge AI applications with limited resources at test time [173]. The idea behind early exiting is that different test data samples may not require the same amount of computation for a correct prediction [101]. By incorporating early exits, the network can exit early from full layers of the DNN if the prediction is already confident enough [101]. This can save computation time and energy, especially in deep and large networks where the added latency and energy usage become more prohibitive [101, 173].

Early exiting can be achieved by inserting auxiliary classifiers at intermediate layers of the network. These auxiliary classifiers are associated with their own loss functions, and the network is jointly trained on a combination of all losses. The weights of the auxiliary classifiers can be trained simultaneously with the network parameters.

Ilhan et al. [174] addressed the enhancement of the computational efficiency of DNNs used in edge AI applications with limited resources. The authors propose a multi-exit DNN inference

framework called EENet, which allows early exiting from full layers of DNN inference for specific test examples [174]. It introduces early exit utility scores that combine confidence measures and class-wise prediction scores to make informed decisions about when to exit early during the inference process [174]. The results demonstrate that EENet outperforms existing hand-tuned or manual rule-based policies, achieving a better balance between latency, throughput, and performance while effectively utilizing limited resources in edge AI scenarios [174].

Teerapittayanon et al. [101] proposed a network architecture called BranchyNet that incorporates side branch classifiers for early exiting of predictions Teerapittayanon et al. [101]. By utilizing the observation that early layer features are often sufficient for classification, BranchyNet allows samples to be classified with high confidence and reduced computation Teerapittayanon et al. [101]. The authors experiment with popular networks like LeNet [25], AlexNet [13], and ResNet [46] to validate the effectiveness of BranchyNet. The results show that BranchyNet significantly reduces inference latency and energy consumption while maintaining high classification accuracy Teerapittayanon et al. [101].

Dehghani et al. [61] presents the Universal Transformer (UT) as a solution to the limitations of traditional recurrent neural networks (RNNs) and Transformers. RNNs are slow due to sequential computation, while feed-forward models struggle to generalize when input lengths exceed training data [61]. The UT combines the strengths of both approaches by introducing a parallel-in-time self-attentive recurrent sequence model [61]. The UT incorporates adaptive computation time (ACT) to dynamically adjust processing steps, improving efficiency [61]. Experimental results show that the UT outperforms RNNs and feed-forward models, achieving state-of-the-art performance and better generalization ability in various sequence modeling tasks, making it a promising solution [61].

2.3.2 Layer Skipping

Dynamic neural networks that skip layers are a type of neural network that can dynamically adjust layer activation during inference [64]. It involves finding the optimal path for data to travel through the network, allowing for flexibility and adaptability in information propagation [64]. This is done by using a gating mechanism to decide whether or not to execute a layer [64, 68]. The gating mechanism is typically a small neural network that is trained to predict whether or not a layer is necessary for a given input [68].

Wang et al. [68] addressed the issue of excessive computation in deep convolutional networks for visual perception tasks. The authors propose SkipNet, a modified residual network [46] that selectively skips convolutional layers based on input-dependent activations [68]. The skipping decision is treated as a sequential decision-making problem and solved using a hybrid learning algorithm that combines supervised and reinforcement learning [68]. The results demonstrate that SkipNet reduces computation by 30-90% while maintaining good accuracy [68]. McGill and Perona [175] proposed dynamic routing which involves using graphs to guide input signals through learned transformations along different paths [175]. The authors find that although the approaches have their advantages, the resulting networks exhibit similar qualitative behavior [175]. Specifically, in image classification tasks, the layers and branches of dynamically-routed networks become specialized in processing distinct image categories.

Cai et al. [102] propose Dynamic Routing Networks (DRNets) that selectively choose transformation branches based on instance-specific importance weights [102]. These weights are generated by lightweight hypernetworks called RouterNets and recalibrated using Gumbel-Softmax for differentiable selection [102]. The instance-aware inference of DRNets, achieved through dynamic routing and branch selection, enables efficient deployment of deep neural networks in real-world applications while maintaining accuracy [102].

2.3.3 Summary

This summary groups the various efficiency method proposed and outlines their major flows.

Zhu et al. [47], Zhang et al. [48], Kumar [54] tried to minimize the memory footprint as well as the computational time through weight pruning [66]. However, these methods fail from the accuracy-compression trade-off which is the decrease of their accuracy as more pruning is applied [47]. Yuan et al. [49], Liu et al. [50], Ding et al. [51] applied quantization techniques to reduce the amount of memory footprint along with the computational requirements. However, similar to the weight pruning methods, accuracy drops as the quantization level increases [49, 50, 51].

Knowledge Distillation (KD) [56] has also been effective in lowering the computational requirement of Vision Transformer. Touvron et al. [30], Wu et al. [176], Wang et al. [164] have shown the effectiveness of this method in the case of vision transformers. In some cases, the technique results in similar performance compared to models trained from scratch with large dataset [30] while in others, performance drops [160]. However, the process of distillation works by transferring dark knowledge from a teacher network to a student network [56]. As such the training of the teacher itself requires extra effort which is a significant cost to consider.

Another well-studied efficiency technique is the low-rank approximation or factorization method where a large matrix multiplication in a neural net is replaced with lower-dimensional matrices of reduced rank. [52, 53, 54, 55] explored different flavors of low-rank approximation and showed an increased speed of the Transformer models. However, they all suffer from decreasing performance depending on the approximation magnitude. LinFormer [177] is a type of transformer that replaced the quadratic-time self-attention of the standard Transformer [19] with a linear time low-rank self-attention. Although it was only tested on NLP tasks, it interestingly performed comparably with models such as BERT [178] while achieving more

than 1.5x the speed of the standard Transformer [19]. However, Tay et al. [179] later showed that the model does not scale well as the original Transformer.

Methods specific to Transformers have also been well-studied. [57] proposed the DynamicViT to dynamically prune the less important tokens in the inference phase. [58] proposed to learn the sparsity pattern of the vision transformer [3] and use such knowledge to prune the connectivity of the tokens thereby minimizing the computation. [59] proposed the block level sparse attention mechanism and showed its effeteness on NLP tasks. [60] further explored the effectiveness of factorizing the attention matrix through the specification method which takes $O(n\sqrt{n})$ time instead of $O(n^2)$. Unfortunately, similar to the pruning and quantization methods, these methods also result in accuracy degradation.

Other methods closely related to this work are layer skipping and the early exiting methods generally termed Adaptive Computation Time (ACT) or dynamic neural networks [61, 63, 64, 65]. Although not on Transformers, [68] proposed the SkipNet, a convolutional neural network (CNN) based model[46, 25], which skips layers depending on the input. [69] developed a mechanism to adaptively learn the role of parts of a CNN network and later use such information to classify easy inputs using earlier layers and propagate harder inputs to the upper layer when needed. Both SkipNet [68] and the adaptive CNN [69] suffer from accuracy degradation.

[101] also proposed a branching technique called BranchyNet where similar to [69], easier inputs exit and classified early while hard examples are processed with more stages. Their AlexNet [13] and LeNet [25] based BranchyNet showed improved performance with lesser inference cost while it under-performed in the ResNet [46] architecture case. Furthermore, due to the branches, BranchNet requires significant extra memory space to store the branches.

In the context of Transformers [19], [61, 62] proposed the early token exits by accompa-

nying the Transformer with LSTM [23]. However, both works suffer from accuracy-speed trade-offs although, in some benchmarks, the depth-adaptive transformer [62] gains slightly better accuracy. Furthermore, [179] showed that the Universal Transformer does not scale as the standard Transformer in a variety of tasks.

Later [63] introduced the depth adaptive Transformer that uses two approaches to decide the routing of tokens: the mutual information (MI) and the reconstruction loss-based estimation. Similar to the Depth-Adaptive Transformer [62] and the Universal Transformer [112], tokens are routed adaptively. Although it achieves comparable performance with the Transformer, the performance rapidly degrades once the number of layers is reduced. [180] later introduced the Length-Adaptive Transformer which uses the *LengthDrop* strategy in which depth is learned through the stochastic mechanism followed by a *Drop-and-Restore* process and an evolutionary search of layer connections. The last stage is used to produce a model with a desired accuracy-efficiency trade-off level. However, once such a model is produced, the computation will be the same for all future samples. Furthermore, the training process takes significant computing power. For example, self-distillation is applied where both the full and subset of the networks are updated separately, a significant cost to consider. Furthermore, the late-stage evolutionary optimization costs extra training resources.

Paper	Method	Domain	Dataset	Speed	Space	Accuracy
Zhu et al. (2021)	Pruning	Vision	ImageNet	21%	20%	▼
Zhang et al. (2022)	Pruning	Vision	CIFAR-100	21%	-	▼
Kumar (2022)	Pruning	Vision	CIFAR-10	7%	-	▼
Yuan et al. (2021)	Quantization	Vision	ImageNet	-	-	▼
Liu et al. (2021)	Quantization	Vision	CIFAR-10	75%	-	▼
Liu et al. (2021)	Quantization	Vision	CIFAR-100	75%	-	▼
Ding et al. (2022)	Quantization	Vision	ImageNet	75%	-	▼
Wu et al. (2022)	Distillation	Vision	CIFAR-10	-	1166%	▼
Wu et al. (2022)	Distillation	Vision	CIFAR-100	-	1166%	▼
Wang et al. (2022)	Distillation	Vision	CIFAR-10	-	0%	▼
Wang et al. (2022)	Distillation	Vision	CIFAR-100	-	0%	▼
Chen et al. (2021)	Low-rank	NLP	WikiText-103	1.58×	98%	▼
Dass et al. (2022)	Low-rank	Vision	ImageNet	1.58×	40%	▼
Wang et al. (2020)	Low-rank	NLP	Wikipedia	1.75×	-	0
Rao et al. (2021)	Sparse Attention	Vision	ImageNet	24%	2%	▼
Wei et al. (2023)	Sparse Attention	Vision	ImageNet	50%	60%	▼
Qiu et al. (2019)	Sparse Attention	NLP	ImageNet	27%	25%	0
Child et al. (2019)	Sparse Attention	NLP	ImageNet	20%	-	▼
Liu et al. (2020)	Adaptive	NLP	Amazon-16	-	-	▲
Wang et al. (2017)	Adaptive	Vision	CIFAR-10	50%	-	▼
Wang et al. (2017)	Adaptive	Vision	CIFAR-100	37%	-	▼
Kim and Cho (2020)	Adaptive	NLP	SQuAD1.1	55%	-	▲
Bolukbasi et al. (2017)	Adaptive	Vision	ImageNet	2.8×	-	▼
Elbayad et al. (2020)	Adaptive	NLP	IWSLT De-En	-	61%	▲

Table 2: Comparison of various techniques of efficiency methods.

Table 2 shows various efficiency methods proposed in the literature. Direct comparison of the existing approaches is impossible since different authors use different datasets and different model sizes. Therefore, Table 2 highlights the contribution made by the works in terms of

speed increase, memory usage reduction, and accuracy improvement. Each of the columns shows the following:

- **Paper:** refers to the paper that proposed the specific technique, model, or method.
- **Method:** refers to the broad type of the proposed efficiency solution.
- **Domain:** refers to the research area in deep learning namely: NLP, Vision, Video, and more.
- **Dataset:** refers to the dataset the authors used.
- **Speed:** refers to the speed increase resulting from the proposed method.
- **Space:** refers to the memory usage reduction resulted due to the proposed method.
- **Accuracy:** refers to whether accuracy increased or decreased due to the proposed method. ▲ shows accuracy improvement, ▼ shows degradation and 0 shows *no change*. Note that reference of the improvement is the baseline specified in the respective papers.

The hyphens (−) in each of the columns show that the authors did not specify the respective column metric. Overall, it can be seen that the models either result in memory usage reduction or speed increase. Furthermore, most proposed solutions do not result in increased accuracy. Those who increased accuracy require extra computation [180] or are limited to token routing, still activating layers [62, 63].

Chapter 3 Methodology

The objective of this work is to propose a new model that minimizes computational costs while maintaining the accuracy of static models. We achieve this through the introduction of an RL agent [113] on top of the Transformer network to manage the activation of the layers of the Transformer [3]. We then apply a post-training optimization using NSGA-II [127] to mitigate three limitations of the model and improve its efficiency. Subsection 3.1 below discusses the research methodology employed.

3.1 Research Methodology

To conduct the research, we adopted the Design Science Research Process (DSRP) [181] as our methodology, as it aligns well with our goal of addressing an existing problem and designing a new method to solve it. The DSRP method consists of six major steps: Problem Identification & Motivation, Objective Formulation, Design and Development, Demonstration, Evaluation, and Communication. However, in our case, the Demonstration and Evaluation steps were combined into a single step called Evaluation, as our evaluation involved assessing the performance gains of the proposed architecture.

Since the motivation for this work stems from the existing problem of accuracy-speed trade-off, we adopted a Problem-Centered Approach for our research. We followed the following steps throughout the research process which is also depicted in Figure 3:

- **Problem Definition:** We began by clearly defining the problem based on three key motivations: the cost of deployment, economic disparity, and environmental impacts. We conducted a preliminary literature review, examining survey articles, seminal publications on major deep learning architectures, and related works that attempted to address the problem. In Subsection 1.2, we stated the problem as redundant computations on input that do not require further transformation, resulting in extra cost. We then formulated three research questions in Subsection 1.3 to guide our specific

approach and solution. Next, we delved deeper into related works, identifying gaps and shortcomings in existing techniques that aimed to solve the problem. This process helped us identify relevant works, analyze their mechanisms, and gather important metrics such as accuracy, memory usage, and computational overheads. By doing so, we clearly identified the research gap and determined the direction for designing our solution.

- **Objective Formulation:** To establish the broad aims and goals of our research, we stated the general objective in Subsection 1.4. This objective provided an overall direction and purpose for the research project. We then defined specific objectives that broke down the general objective into smaller, manageable components, guiding the specific actions we needed to take. These specific objectives provided a clear roadmap for the research process and helped us stay focused and on track.

- **Design & Prototyping Iteration:** Building upon the information gathered in the previous steps, we engaged in brainstorming sessions to generate various designs for a new Transformer architecture that could potentially reduce computation overhead by eliminating redundant layer computations. We explored ideas such as reward functions, loss functions, and agent architecture, and quickly prototyped and iterated over these ideas. Through this iterative process, we arrived at a feasible design based on reinforcement learning (RL) principles that could be applied to the Transformer architecture.

- **Evaluation:** Once we finalized the design through the design and prototyping iterations, we proceeded to evaluate its performance and viability through multiple stages of testing. Initially, we tested the design on models with an increasing number of layers to determine the effect of hyperparameters. Using the discovered hyperparameters, we then tested the design on state-of-the-art models to assess its effectiveness. Furthermore, we evaluated the applicability of the design in a transfer learning scenario to gauge its real-world usability. Finally, during the evaluation process, we identified

three limitations of this high-performing model, which led us to introduce a joint RL-NSGA post-training optimization technique to enhance the usefulness of the proposed architecture in practice.

- **Knowledge Contribution:** Finally, we analyzed the results and the architecture to identify key findings and contributions. These findings were derived from the evaluations conducted during the Evaluation stage. They included performance and efficiency reports as well as insights about how the model works.

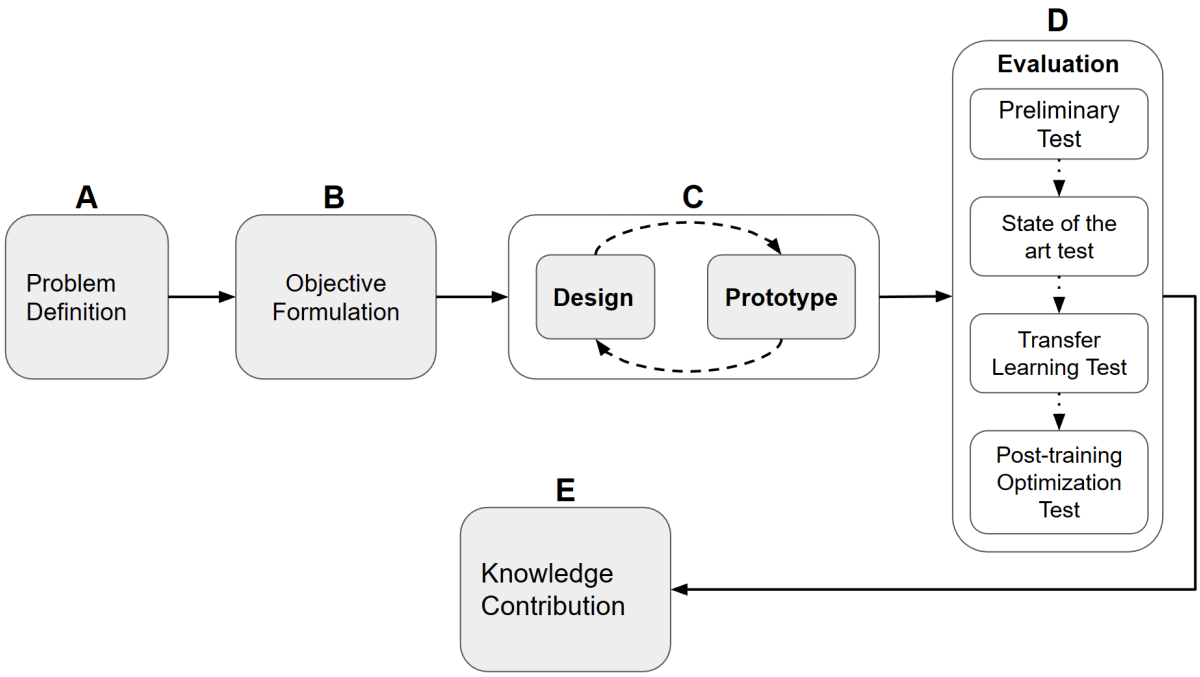


Figure 3: Design Science Research Process based problem-centered approach research methodology of this thesis work.

Figure 3 shows the process of developing our proposed models. The solid black arrows show what step follows after a given step.

- Block **(A)** shows that the thesis work starts by identifying a problem.

- Block **(B)** shows the objective formulation based on the assessment done on the previous works.

- Block **(C)** shows the iterative design and prototyping phase where different ideas are being proposed and implemented. The broken black lines show the iterative process between design and prototyping.

- Block **(D)** shows the full-scale evaluation phase of the candidate design. It passes through multiple stages of testing as shown by the broken arrows in the **(D)** block.

- Block **(E)** shows the knowledge contribution phase where through probing, data analysis, and other methods, key findings, discussions, limitations, and future works are presented.

While the Problem identification, objective and related works are explained in Chapters 1 to 3, this chapter and the rest focus on the proposed technique, the experiment, and the key findings.

The proposed method follows the typical machine learning training procedure. Figure 4 illustrates the entire procedure, divided into five main operations: Data Acquisition (A), Preprocessing (B), Training (C), Post-training Pruning (D), and finally Deployment (E). Subsection 3.2 explains the data acquisition process. Following that, subsection 3.3 discusses the preprocessing process. Subsection 3.4 then outlines the design of the RL agent that learns a layer-skipping policy. Then subsection 3.5 discusses a method to improve this model using NSGA-II even more. The last step *Deployment (E)*, however, is not discussed in this work as it is not in the scope of the work.

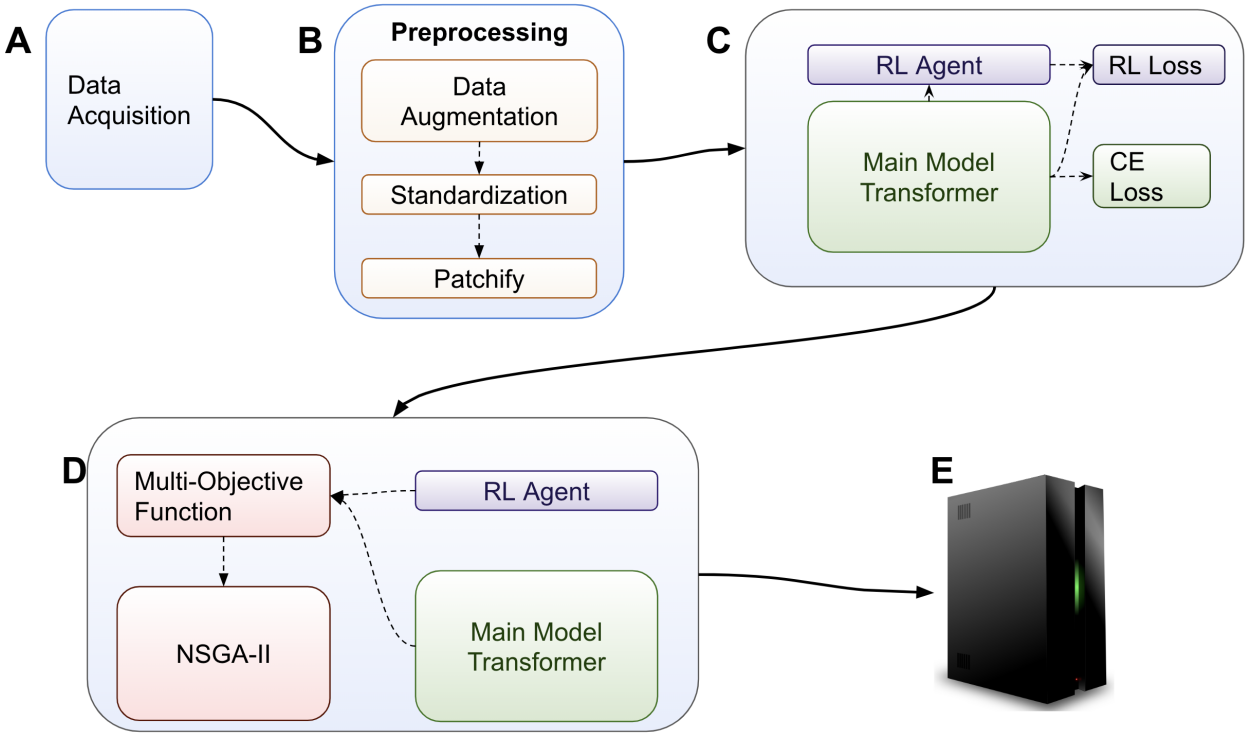


Figure 4: Visualization of the design of the RL-based layer skipping transformer and its post-training optimization.

Figure 4 shows the process of developing our proposed models. The solid arrows show the action taken from one step to another.

- Block **(A)** shows data acquisition process.

- Block **B** shows the preprocessing process. Each of the blocks in it shows a single step where one follows another. The broken arrows show the step followed.

- Block **(C)** shows the proposed layer skipping model containing the main Transformer and RL models, along with their loss computations. The line from the main model to the RL agent shows the transfer of the class token. The RL Loss takes predicted actions from RL sub-agents and the accuracy from the main model to calculate the loss. The Cross-Entropy (CE) Loss takes the prediction of the main model and backpropagation is applied based on the result.

- Block **(D)** shows the post-training optimization using NSGA-II operating on multiple objectives that are coming from the RL and the Transformer networks. The broken arrows from the main model and the RL agent show the transfer of the depth count, the RL agent reward, and the accuracy of the Transformer model to the NSGA-II multi-objective setup. The broken line from the multi-objective block to the NSGA-II block shows that NSGA-II algorithm operates on the multi-objective function.

- Block **(E)** shows the last step where, after the training is done, the model is deployed on a server.

3.2 Data Acquisition

This project relies on already collected standard benchmarking datasets namely: CIFAR-10, CIFAR-100, and Tiny-ImageNet [1, 2]. These datasets are used to study new algorithms, techniques, and architectures as they pose the same challenge for all new ideas. This thesis work proposes new techniques and as such, these datasets are used to compare our method against other existing approaches. Their size and training steps are specified in Table 3. Overall, CIFAR-10 is the easiest to achieve good performance as shown in various works [68, 182], and Tiny-Imagenet is the hardest among the three [2, 183]. Note that, TinyIma-

Dataset	Train Size	Test Size	No. of Classes	Size	Training Steps
CIFAR-10	50k	10k	10	32×32	60k
CIFAR-100	50k	10k	100	32×32	82k
Tiny-ImageNet	100k	10k	200	64×64	156k

Table 3: Details of the experiment datasets.

Table 3 shows the details of the datasets used in the experiments. CIFAR-100[1] and Tiny-ImageNet datasets require more steps as they are harder datasets due to the increased number of classes. Tiny-ImageNet requires even more steps than the CIFAR-100 due to the increased number of training samples. Furthermore, Tiny-Imagenet[2] is twice as large as the other two in terms of image size.

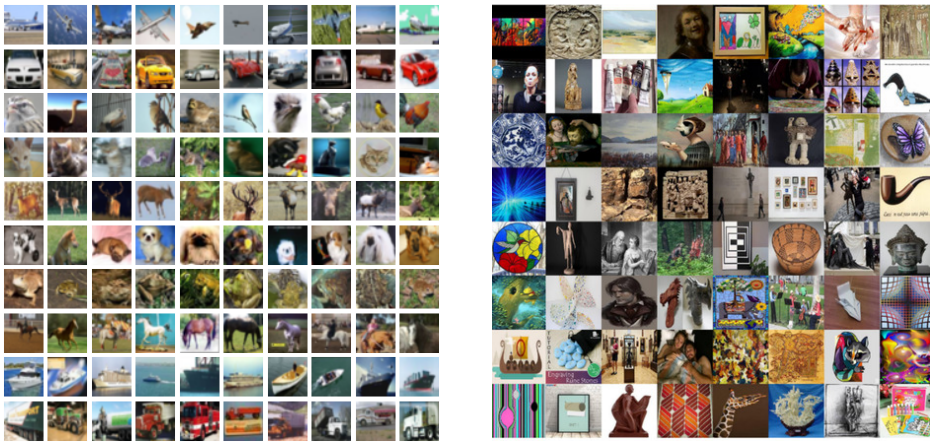


Figure 5: Sample images of CIFAR-10 (left)[1] and Tiny-ImageNet (right)[2]

genet [2] is the downgraded and smaller version of ImageNet [67]. CIFAR-10 has 10 classes, each with 6,000 images with classes such as *airplane*, *automobile*, *cat*, and *dog*. CIFAR-100, on the other hand, has 100 classes, each with 600 images, and the classes are grouped into 20 superclasses such as *aquatic mammals*, *flowers*, and *vehicles* [13]. Figure 5 shows sample images.

3.3 Preprocessing

After the data is collected, the next step is preprocessing [13]. However, in practice, it is integrated and automated in the training pipeline and this work follows a similar procedure developed by [183]. This preprocessing contains four parts. First, data augmentation methods are applied: Random Resizing, flipping, and Cropping [13], AutoAugment [184], and Random Erasing [185]. Second, Standardization [13] is applied based on the respective datasets' mean and standard deviations specified in Table 4. It is applied based on eq (3.1). Finally, the Patchify [3] operation is applied, where the image is divided into small patches as shown in Figure 6. In, Figure 6, the top figure shows patchifying. in the In this work, similar to [183] for CIFAR-10 and CIFAR-100, the patch size was set to 4×4 and 8×8 for Tiny-ImageNet.

Dataset	**Mean (RGB)**	**Standard deviation (RGB)**
CIFAR-10	0.4914, 0.4822, 0.4465	0.2023, 0.1994, 0.2010
CIFAR-100	0.5071, 0.4867, 0.4408	0.2675, 0.2565, 0.2761
Tiny-ImageNet	0.4802, 0.4481, 0.3975	0.2302, 0.2265, 0.2262

Table 4: Mean and standard deviation of CIFAR-10, CIFAR-100 and Tiny-ImageNet dataset

$$z = \frac{x - \mu}{\sigma} \tag{3.1}$$

where z is the standardized image, x is the input image, μ is the mean value of the image and σ is the standard deviation of the image.

Figure 6, the top figure shows patchifying. The bottom figure shows the flattened version of the image. This patch grid is restructured to be a list and each of the patches will be flattened into a vector. For instance, the CIFAR-10 image has a size of 32×32 [1]. After patchifying it into 4×4 pieces, it will be a grid of 64 patches. Each of these patches has $4 \times 4 \times 3$ size in the width, height, and depth dimensions respectively. This size will then be

Figure 6: Patchifying and Flattening an image. The top figure shows patchifying an image. The bottom figure shows the flattening of the patchified image.[3]

flattened to have a size of 48 [183]. Collectively, the input will be a 64×48 float tensor due to the 64 patches [183]. Generally, this tensor will have $N \times P$ dimension where N is the number of patches or sequence length and P is the dimension of the flattened patches [3]. The next step is converting the patches into *embeddings* through a learnable weight matrix [3]. The term *embedding* is borrowed from the field of NLP where the meaning of a word is embedded in a fixed dimensional vector [99]. The conversion to embedding is a simple linear transformation function *PatchEmbed* given by eq (3.2) [3]:

$$\text{PatchEmbed}(X) = X \cdot W + B \tag{3.2}$$

where X is the image tensor with size $N \times P$, W is the projection matrix with size $P \times D$ and b is the bias term with size D [3]. The resulting tensor will have the size of $N \times D$ where D is the hidden size of the Transformer [19]. Again, borrowing from NLP literature, each element of this tensor are called *token* [98]. Hence, there are N tokens in the sequence each having D size.

Once the image is embedded into tokens, a randomly generated *class token* [19] is appended to the sequence, making the sequence length $N + 1$. At the end of the forward propagation, this class token is used for classification [3]. Finally, this sequence is fed to the Transformer

as described in subsection 2.1.4. The overall operation of Vision Transformer is summarized in Figure 7.

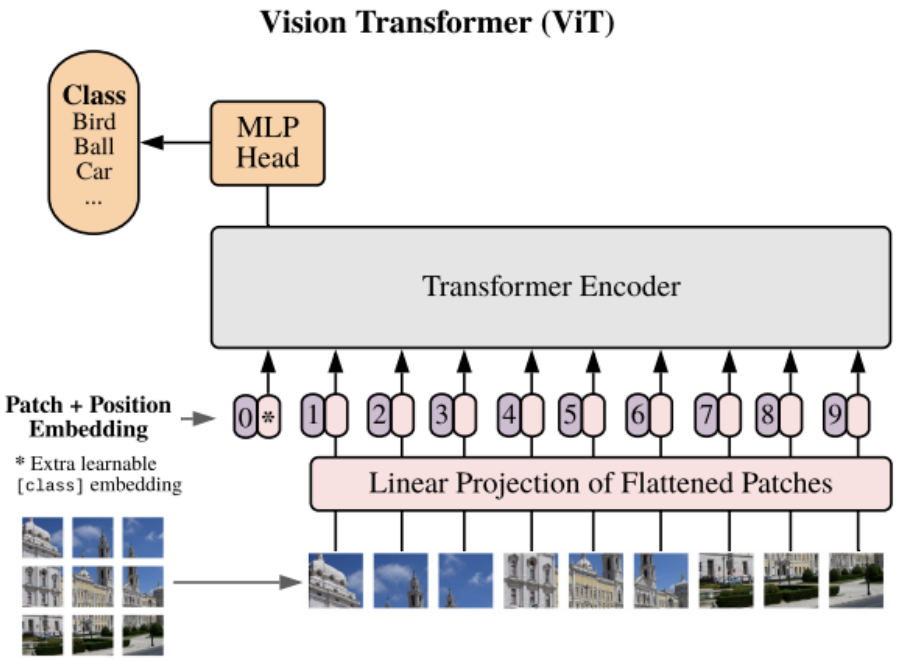


Figure 7: Vision Transformer [3]

Figure 7 illustrates the Vision Transformer (ViT). The input image is first patchified. Then, as shown at the bottom, they are flattened to look like a sequence and fed to the ViT. They are then converted to embeddings using the *Linear Projector* described in eq (3.2). Position Encoder is then added to the embeddings and fed to the first Transformer layer. These sequences of embeddings or tokens are propagated until the last layer without a change in size. This is due to the fact that the ViT is isomorphic, having a similar shape across all the layers. Finally, as shown in the orange box, the class token is extracted (MLP Head) and used as a classification embedding.

3.4 Design of RL-based layer skipping Transformer

Instead of following the conventional approach of constructing a vision transformer[3] with sequential layers, the proposed model comprises two distinct components. The first component is a standard image classification[13] model that undergoes supervised training. The second component involves an RL agent trained using the REINFORCE algorithm [115]. Figure 9 shows the difference between a static Transformer [3] and the layer-skipping one. The agent is a collection of linear layers residing at each transformer layer referred to as *sub-agents*.

Let's denote the variable j to index the transformer's j^th layer. T_j is the j^th layer and c_j is the class token [3] of the output of that layer. At each T_j, there is a sub-agent - a binary logistic regression model - which takes c_j and decides whether to skip the next layer T_j or not. Each of the sub-agents learns the probability of skipping the next layer given the class token of the current input: $p_j(a_j = skip|c_j) = f_j(c_j)$ where $f_j(c_j)$ is given by eq (3.4). Since this decision is a binary decision, a sigmoid function [75] given by eq (3.3) is employed. The sigmoid function maps the output of the linear layer into a probability [5] which later is used as a thresholding value to decide whether to activate the next layer or skip it. Plot 8 shows the range of the function. This decision function $\pi(c_j)$ is shown in eq (3.5).

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{3.3}$$

$$f_j(x) = \sigma(X.W_j + b_j) \tag{3.4}$$

$$\pi(x) = \begin{cases} 1 & \text{if } f(x) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

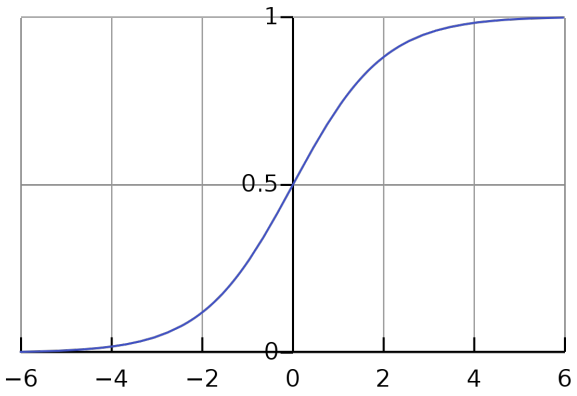


Figure 8: Sigmoid Function

If $\pi(c_j) = 1$, the next layer is activated, otherwise it is skipped. At each layer, the agent issues decisions for each sample on whether to skip layer T_j or not based on c_j. Those samples x_j^T with $\pi_j(c_j) = 1$ are extracted from the batch and pass through the T_j. The remaining, x_j^I, will be cloned to become a_j^I (identity function). After the first group passes through the transformer layer T_j, the activations a_j^T will be merged with the a_j^I. The merging is simply placing each activation in its original batch index. Algorithm 2 summarizes the whole operation of a layer and a sub-agent.

Algorithm 2 Skipping Mechanism in a Transformer Layer

1: Execute $\pi_j(c_j)$ and get s_j

2: Extract inputs with $s_j > 0.5$ and pass them through T_j

3: Concatenate the result with the rest on the batch axis and get a_{j+1}

4: Return a_{j+1}

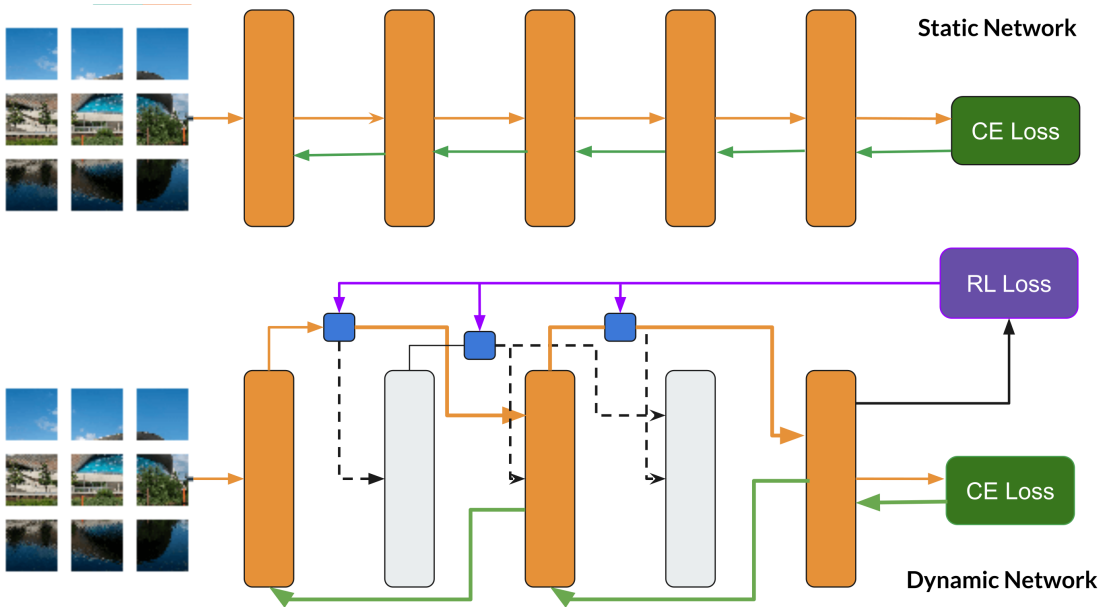


Figure 9: Static vs Layer skipping Vision Transformers

Figure 9 shows the architectural difference between static and dynamic networks.

- The upper architecture is the well-known feed-forward static Transformer.
- The bottom one is the Dynamic Transformer architecture.
- *CE Loss* is cross-entropy loss.
- *RL Loss* is the REINFORCE loss for the agent.
- Blue boxes are the sub-agents.
- Orange lines are the forward propagation path selected by the agents.
- The gray layers are layers that were skipped.
- Green lines show back-propagation.
- Broken black lines show the potential path of the inputs.

3.4.1 Agent Training Algorithm

Although there are various RL algorithms[113] to choose from as described in subsection 2.1.7, the following constraints shaped the choice of the RL training algorithm:

- Since the inputs in the current RL setting is a continuous space vector (class token), the

learning method should be gradient-based optimization [115]. Hence, tabular methods are not applicable.

- Since the rewards in the intermediate steps are not known, instead of value-based methods such as Actor-Critic or Deep Q-Learning, Monte Carlo-based [113] approaches are appropriate.

- Since computing a single trajectory is expensive, methods such as Proximal Policy Learning that require multiple updates are expensive [117].

As such, the algorithm that satisfies the above constraints is *REINFORCE with baseline* [115], a variant of the Monte-Carlo-based Policy Gradient method [113].

3.4.2 RL Agent

As described above, each of the transformer's layers T_j has an accompanying sub-agent made of a linear layer that predicts whether T_j should be applied to the incoming inputs x_j or not. Following this operation, Figure 9 depicts the difference between a static transformer and a dynamic transformer that uses RL sub-agents. These linear layers are trained using policy gradient [115] as discussed below.

The sub-agents are trained by maximizing the expected reward R similar to [186]. The reward signals what we would like the model to achieve. Reiterating, the statement of the problem, we would like to build a model that skips irrelevant layers without compromising the accuracy. As such, intuitively, we can maximize the *Accuracy* and minimize the *number of activated* layers. Hence, the problem is a multi-objective function (MOO) [125] where there are two conflicting functions: accuracy function $A(I)$ and layer activation function $L(I)$ given an input image I. It has been shown that increasing the number of layers increases accuracy [46, 3, 165] and as such reducing the number of computed layers conflicts with the aim of keeping accuracy intact. Hence, the central *magic* lies at activating layers only when they are needed similar to [68]. Hence, our strong hypothesis is that an RL agent

can learn such operation as it has been shown to design neural network architectures by its own [186] and skip layers [68].

In MOO [126], the values of the objective functions should be normalized to an equal range so that one function does not get higher importance [187]. The accuracy function is already in the range of 0 and 1. The activation of the layers can be set in the range of 0 - 1 by dividing it by the depth of the model. Therefore, given an image I, the reward of the agent R_i can be computed using eq (3.8):

$$A(I) = \begin{cases} 1 & \text{if } \phi(I) = \text{true label} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

$$L(I) = \frac{\sum_{j=1}^{N} \pi_j(c_j^I)}{N} \tag{3.7}$$

$$R_i = A(I) - L(I) \tag{3.8}$$

where ϕ is the image classifier Transformer, I is the current input, *true label* is the actual label of the I, $A(I)$ is whether the image is accurately classified (1) or not (0), N is the total number of layers, j is the layer index, c_j^I is the class token of the image I on the jth layer, π_j is the policy function/agent at jth layer, $L(I)$ is the fraction of the activate layers, and finally R_i is the reward the agent gets on input I. Note that, this reward is only available in the last layer and there is no reward for the intermediate layers. As such, it can be propagated back to each layer using the discounted reward [113] as shown below in eq (3.10):

$$G_i = R_i = A(I) - P(I) \tag{3.9}$$

$$G_i^j = \gamma \times G_i^{j+1} \tag{3.10}$$

G_i is the reward in the last classifier layer. Now that, the reward for each of the sub-agents is propagated, they can be trained using standard RL formulation [113]. At jth layer, the actions $a_j \in \{\text{skip}(0), \text{activate}(1)\}$. The objective of the jth sub-agent is to

maximize expected reward, given by the $J_j(\theta_j)$ in eq (3.12) similar to [186]. Then it can be approximated using eq (3.13) using gradient-based learning [113]:

$$J_j(\theta_j) = E_{P_j(a_j|c_j;\theta_j)}[R_j] \tag{3.11}$$

$$\nabla_{\theta_j} J_j(\theta_j) = \sum_{i=1}^{N} E_{P_j(a_j|c_j;\theta_j)}[\nabla_{\theta_j} logP(a_j|c_j;\theta_j)R_j] \tag{3.12}$$

$$\nabla_{\theta_j} J_j(\theta_j) = \frac{1}{N}\sum_{i=1}^{N}(G_j^i - b_j(c_j^i))logP_j(a_j|c_j;\theta_j) \tag{3.13}$$

where N is the number of images in a given batch, R_j is the reward of the agent at layer j, $\nabla_{\theta_j} J_j(\theta_j)$ is the gradient of the expected reward for sub-agent j, that need to be maximized. Hence, it is a gradient ascent learning. G_j^i is the discounted reward[113] of sub-agent j driven through eq (3.10). $P_j(a_j|c_j;\theta_j)$ is the probability of activation of the next layer using the sigmoid function defined in eq (3.3). $b_j(c_j^i)$ is the baseline function [115] for sub-agent j which is described in subsection 2.1.7.1. In the training phase, the mentioned operations are exclusively applied. However, in testing and real-world scenarios, the sub-agents have learned to skip based on the training labels, rendering the need for labels unnecessary. As a result, the agents can autonomously make decisions without relying on explicit labels.

3.5 Post-training Optimization

As we will prove in subsection 4.2, the model described above can outperform state of the art static model while reducing the number of activated layers. Upon further examination, the proposed approach can be subject to criticism due to three significant limitations. This section will first outline these limitations and subsequently propose a solution through the application of late-stage pruning using the joint RL-NSGA-II method. The identified problems with this model are as follows:

- **Control:** Methods such as weight pruning [66], low-rank approximation [76], knowledge distillation [160], and quantization [139] offer a control mechanism on how much the model should be efficient. The designer then can specify how much the scarification of the accuracy should be depending on the deployment scenario. The plain dynamic

model does not offer this capability. As such, the entire model needs to be deployed on the edge and mobile devices to be operational, which may not be ideal.

- **Threshold Value:** The skip prediction is implemented using the sigmoid function [75, 25]. As such, 0.5 is set as the skip decision-making threshold similar to the binary classification problem in the literature [5]. However, as we show later in subsection 4.3, this may not be the optimal value and need to be optimized based on the data.

- **Unnecessary Layer Activation:** Despite being much more efficient than the static model, the layer skipping model still activates unnecessary layers for an input. To prove this, first, let's represent the computation path as follows: a layer activation can be represented with 1 and a skip can be represented by 0. Hence, if there are 9 layers in the model and an input skips the 5^{th}, 7^{th} and 8^{th}, then the following string reflects the path of the input propagation:

$$1 - 1 - 1 - 1 - 0 - 1 - 0 - 0 - 1$$

  For each image, all of such paths were collected. Then, by enumerating all the available paths using the data, at a time one of the layers is manually set to 0 (forcing a layer to be skipped disregarding the decision made by the agent). Then the input is processed using the manually updated forward propagation path. If the image was not classified, since the only change is the deactivation of that layer, we can deduce that that layer is critical for that particular image. If the image was correctly classified, however, it means that the layer is unnecessary computation. This is natural to expect as all machine learning models are expected to make mistakes. After examining each image and evaluating whether any unnecessary layers were being activated, we discovered that, on average, 56% of the images in CIFAR-10 and CIFAR-100 activate at least one layer unnecessarily. Table 5 show the amount for each class ID. In the CIFAR-100 case, since there are 100 classes, classes 0 - 9 were taken as shown in the second row of Table 5.

Table 5 presents the amount of unnecessary activation of layers. For each class (column) in

Class ID	0	1	2	3	4	5	6	7	8	9	**ave. %**
CIFAR-10	70	65	68	67	67	66	65	73	68	65	**61**
CIFAR-100	50	56	48	50	53	49	52	50	54	56	**52**

Table 5: Unnecessary layer activation in RL-based dynamic transformer. Overall, the model activates a lesser number of unnecessary layers on CIFAR-100 than CIFAR-10.

the data, the respective percent of images activate at least one layer unnecessarily. The first row is for CIFAR-10 and the second row is for CIFAR-100. On average, in CIFAR-10 case, 61% images activate unnecessary layers while it is 52% in CIFAR-100. Luckily, all these three problems can be fixed using a post-training optimization method similar to [50]. In this work, we employed a genetic algorithm [188], specifically, the Non-Dominating Sorting Genetic Algorithm (NSGA-II) [127, 188] as it has been employed to prune vision transformer and found successful [129, 128].

The aim of the NSGA-II [127] is to come up with a list of options usually known as *Pareto Front* where each option contains a compromising set of objective values such as efficiency level and performance specified in eq (3.8). This gives developers the ability to choose a solution among the Pareto fronts for deployment based on their requirements. As we show later in subsection 4.3, the algorithm also can also be trained to optimize the right skipping threshold value for each layer. In addition, it reduces redundant computation shown in Table 5. The design of this method using NSGA-II is as follows:

- **Gene:** Firstly, the gene should encode whether a layer should be pruned or not. Secondly, it should encode the skipping threshold value for that layer. Both of these can be encoded using a single float value g where if $g < 0.0$, the layer is removed. Otherwise, g will be the thresholding value for that layer. As such either $g = -1$ or $0 \leq g \leq 1$. However, we discretize the continuous value of g it is easier to apply crossover and mutation operations [124]. For instance, with a step value of 0.25, the possible values of g are:

$$\{-1, 0.0, 0.25, 0.5, 0.75, 1.0\}$$

- **Chromosome:** For L number of layers, the chromosome is a L size vector containing the gene for each layer. For instance, given a 9-layer model, a chromosome might be:

$$[0.5, 0.2, -1, 0.8, 0.8, -1, 0.8, 0.2, 0.5]$$

- **Crossover:** We used the *One Point Crossover*. *Uniform Crossover* and *Multi Point Crossover* performed similarly in the preliminary experiments. As such, *One Point Crossover* is employed to minimize the computation. Therefore, given two individuals or chromosomes, a random index is first generated which will be a point of crossover. Then the left sides of the chromosomes based on the selected random index are exchanged. Figure 10 depicts this process using possible values of the genes:

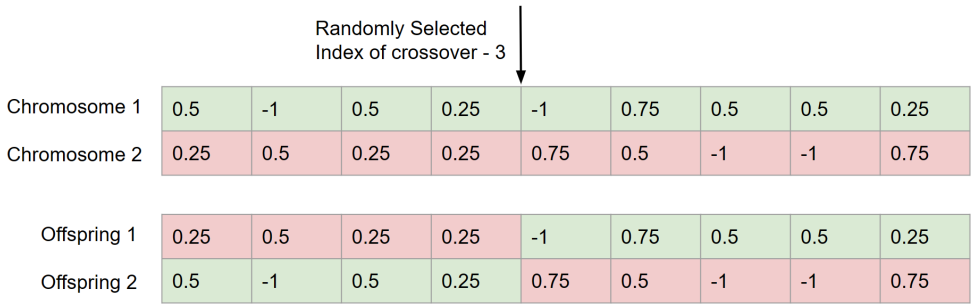


Figure 10: Crossover method using *One Point Crossover*. Using index 3 as a point reference, the left sides of the rows are exchanged.

- **Mutation:** Random replacement of the gene based on a uniform distribution from the possible values of g.

- **Sampling:** Random selection using uniform distribution.

The search space for this problem is defined as $2^L \times k$, where L represents the number of layers and k represents the number of possible values for the threshold. As demonstrated in subsection 3.4.1, a larger value of k corresponds to a superior model, but it also entails increased computational demands during hyperparameter search. Conversely, a smaller value of k reduces the computational burden but runs the risk of an underfitting configuration.

This is because selecting under limited choices may result in the optimal value not being discovered.

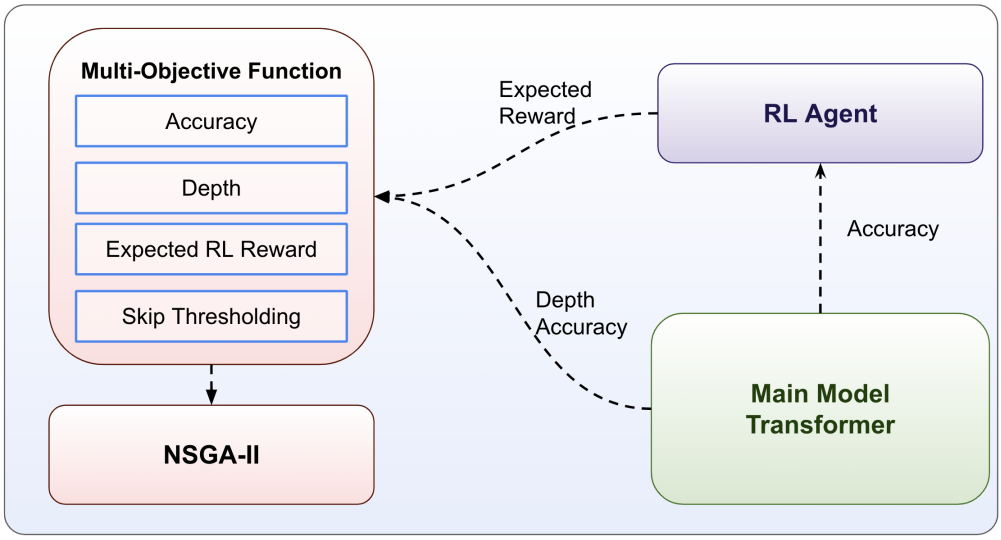


Figure 11: Integration of NSGA-II with RL-based post-training Optimization

Figure 11 shows how the late-stage optimization integrates the RL and the NSGA-II algorithms. The multi-objective function is constructed from the current depth of the transformer, the expected reward of the RL agent, and the accuracy of the transformer. The NSGA-II algorithm then outputs Pareto Fronts that have various degrees of compromises between accuracy and speed.

The NSGA-II algorithm tries to minimize the number of layers activated and optimize the activation rate. As such, it depends on the accuracy of the model, the number of activated layers, and the expected reward of the agent. Figure 11 illustrates the operation pictorially. Eq (3.15) defines an optimization problem for this post-training optimization.

$$\text{Minimize:} \quad \mathrm{L}(f) \tag{3.14}$$

$$\text{Maximize:} \quad \mathrm{A}(f) \tag{3.15}$$

where f is a neural network, L(f) represents the depth of the neural network (the number of layers) defined in eq (3.8) and A(f) represents the classification accuracy of the network defined in eq (3.6).

In this setting, the RL agent does not know about the optimization made by the NSGA-II algorithm. We refer to this training *Plain-NSGA* training. We then propose to create communication between the RL agent and NSGA-II by injecting the expected reward of the RL agent into the objectives of NSGA-II and updating the RL agent over the NSGA-II training. Note that, the vision transformer will also be updated in a similar manner to [129]. The major difference is, the agent's expected reward is included so that NSGA-II is aware of the skipping dynamics. Hence, the expected reward of the agent $E_{P_j(a_j|c_j;\theta_j)}[R_j]$, which is approximated using the moving average (eq 3.19)[113] is included in the objective list of the NSGA-II as shown in eq (3.18).

$$\text{Minimize:} \quad \text{L}(f) \tag{3.16}$$

$$\text{Maximize:} \quad \text{A}(f) \tag{3.17}$$

$$\text{Maximize:} \quad E_{P_j(a_j|c_j;\theta_j)}[R_j] \tag{3.18}$$

$$\mu_t = \mu_{t-1} + \frac{x_t - \mu_{t-1}}{N} \tag{3.19}$$

Equation 3.19 takes the previous mean and adds the new incoming reward to compute the new mean. As such, μ_t holds the mean of the RL-agent reward. The RL agent can now be fine-tuned on the new environment that is being optimized by the NSGA-II. Hence, the updated way of pruning is defined in the eq (3.15). We refer to this training *Joint RL-NSGA* training. Algorithm 3 illustrates the details steps of the joint training of the genetic algorithm and the model and it is visualized in Figure 11

The difference between Plain-NSGA and Joint RL-NSGA is, in the Plain-NSGA case, the NSGA-II algorithm might not make the optimal decisions that will also maximize the agent's

reward over the dataset. Furthermore, as the RL is not updated but layers are pruned, the RL may not be effective as it was since the environment has now changed. In the Joint RL-NSGA training, however, the NSGA-II will optimize and increase the agent's reward over the entire dataset. Furthermore, the agent is trained together with the NSGA-II. Hence, it will have an updated knowledge of the new pruned model structure. As we show in subsection 4.3, the second method improves both the performance and efficiency of the model when compared to Plain-NSGA.

Algorithm 3 Joint RL-NSGA Algorithm

1: **procedure** JOINT-RL-NSGA(Population, $RL - ViT(\theta)$)
2: Initialize the population
3: **while** Not converged **do**
4: Sample a batch of images B
5: Evaluate the fitness of each individual in P on B
6: Perform a non-dominated sorting
7: Apply the crowding distance operator
8: Select the parents for the next generation
9: Generate the offspring population
10: Replace the old population with the new population
11: Update θ by backpropagating using SGD
12: **end while**
13: Return the best solution
14: **end procedure**

Algorithm 3 shows the pseudocode of the joint RL-NSGA post-training optimization. Similar to Algorithm 1 of NSGA-II, it takes a list of chromosomes as Population. The difference is, this one takes the RL-based ViT model parameterized by θ. Given the batch of image inputs shown in line 4, it updates the parameter θ to increase the accuracy on the current NSGA selected setting. Eventually, both the changes made by the NSGA and the accuracy of the model stabilize and the algorithm results in Pareto-front model settings which will

have different accuracies achieved in various numbers of layers.

Chapter 4 Experimentation

4.1 RL-Based Layer Skipping Dynamic Transformer

To evaluate the RL-based layer skipping formulated in subsection 3.4, a baseline static ViT model is first implemented with $mlp\_hidden = hidden = 384$. Then 7, 10, and 16-layer models were trained on CIFAR-10 and CIFAR-100 for baseline comparison. The full list of hyperparameters is specified in Table 6 and different models were run based on the specified hyperparameters, after which the highest model was selected for each size category of the models. The bold ones in Table 6 show the final selected model hyperparameters. Generally, the choice of the hyperparameter ranges was affected by the available limited computing power. However, significant trials were done to select the best setting. The following list describes the hyperparameter choices:

- **Number of layers:** In the initial trial, the number of layers 3 to 6 produced significantly lower than the state-of-the-art performance. However, searching for model sizes greater than 6 one by one was too costly. Hence, to test how the model performs as the scale increases, 7, 10, and 16 were chosen with increasing numbers of 3 and 6.

- **Batch Size:** Small batch size takes longer to finish a training session due large number of steps while the higher batch size is faster due to the small number of steps. As such, the batch size started at 128 and increased to 768. The accuracy increased as batch size increased and as such 768 was taken as the final batch size.

- **Dropout:** The dropout starts with 0.0, which is similar to no dropout. Then it was increased by 0.1 until 0.4. The accuracy increased from 0.0 to 0.2 and decreased from 0.3 to 0.4. Hence, 0.2 was taken for the final setting.

- **Learning Rate:** The learning rate started at 0.01 and was divided by 10 until 0.0001. In the 0.01 case, the gradients kept exploding and training halts due to exploding gradients. The 0.0001 learning needed longer epochs since the gradient updates were

Num of layers	7, 10, 16
batch size	128, 256, 512, **768**
dropout	0.0, 0.1, **0.2**, 0.3, 0.4
learning rate	0.01, **0.001**, 0.0001
num of heads	6, 8, **12**

Table 6: Model hyperparameters. Bold shows the parameter that was selected for the final model.

very small. 0.001 balances between the two and the best result was attained at this value.

- **Number of attention heads:** The standard Transformer [19] employed 12 attention heads. However, to check if lowering this number benefits small vision transformers, 6 and 8 were tried. 12 attention heads had better results than the other two.

The skip policy agents were then added to those base models (7, 10 & 16-layer static transformers). All models were trained using the standard back-propagation algorithm of stochastic gradient descent. As described in subsection 3.3, preprocessing and regularization methods were applied. Furthermore, the models were set 16-bit to efficiently use the GPU memory available. The optimizer is AdamW [189] as its industry and literature standard. All models were trained on the three standard image datasets namely: CIFAR-10 [1], CIFAR-100[1], Tiny-Imagenet [2]. The training epoch was set to 200 as it stopped improving beyond 200 epochs.

4.2 Result

Table 7 shows the performance of the RL-based dynamic transformer models. Indeed, the agents learned optimal policies and achieved the static models' performances. Increasing the

number of layers in the agent did not improve upon the single-layer FFN. Hence, to avoid extra computation, the single-layer FFN was used. In the baseline network case, however, a preliminary experiment on a 10-layers dynamic model showed that a two-layer network is significantly better than a single one ($\approx 2\%$ increase). The investigation of this phenomenon is out of the scope of this thesis and it can be pursued as a future work. However, the policy network only needed a linear transformer, whereas the baseline required two layers with a non-linear activation function. This leads us to hypothesize that estimating the value of a state involves a non-linear operation.

On the other hand, sharing the parameters across the layers (through LSTM or the same FFN) severely hurts both the skip ratio and accuracy of the model. Precisely, the agent issues almost the same decision in all the layers despite being fed different *CLS* tokens. After propagation of the reward from the top layer to the bottom, whitening is applied as it is standard practice. Without the whitening, the model either results in abysmal performance or halts due to numerical stability issues. Finally, in the case where the accuracy and processing ratio become equal, the reward becomes zero. Such reward creates numerical instability as zero rewards are propagated. Hence, we added $\epsilon = -10^{-5}$ on the reward so that such cases can be avoided. A positive epsilon value showed a small drop in the accuracy of the model. This suggests that punishing the agent by a small amount in such a scenario nudges the model either to increase the accuracy or lower the processing ratio.

Lee et al. [183] experimented with vision transformer on small dataset such as CIFAR-10, CIFAR-100 and Tiny-Imagenet. To validate the technique formulated here, their model was used as a baseline and then integrated with the RL-based skipping technique. First, their result was replicated. In their setting, they run the training with batch size 128. It was doubled to minimize the time it takes and as such, the accuracy of the base model itself increased. Apart from the batch size, nothing has changed in the entire experiment process. The dynamic version of the same model increased the performance as expected. The

Model	Layers	CIFAR-10	CIFAR-100	Speed Img/ms	No. Params
Vanilla ViT	7	91.24	67.43	3581	6M
Dynamic ViT	7	91.34	68.26	4948 (↑ 38%)	6M
Vanilla ViT	10	91.37	68.11	2575	9M
Dynamic ViT	10	91.96	68.89	3737 (↑ 45%)	9M
Vanilla ViT	16	90.88	68.29	1251	14M
Dynamic ViT	16	**92.17**	**69.62**	3072 (↑ 145%)	14M

Table 7: Performance of the RL-based dynamic transformer models (7, 10, and 16 layers) on CIFAR datasets.

results are summarized in Table 8. The dynamic model outperformed the well-tuned static model. Note that, Lee et al. [183] introduced the Locality Self-Attention (LSA) and Shifted Patch Tokenization (SPT) to increase the accuracy. However, this experiment is done on the plain small vision transformer so that, their technique does not affect the result in any way. Furthermore, since the LSA and SPT techniques are formulated specifically for small vision transformers and not general enough for various domains, the experiments were focused on the plain model.

Direct comparison to other techniques is nearly impossible since the literature shows efficiency results on different domains (Vision, NLP), different model sizes, and different datasets even in the same domain. Nevertheless, we compare the result to the previous works such as SkipNet [68] - a reinforce-based layer skipping mechanism, [190] Lightweight Parameter Pruning - Pruning-based optimization, quantized version of the static vision transformer [183]. Note that, we show their best result along with ResNet [46] baseline.

Model	T-ImageNet	CIFAR-10	CIFAR-100
SkipNet [68] (Dynamic)	-	92.38	67.79
Pruning [190]	-	92.82	70.01
ResNet Baseline [46]	-	93.68	71.85
ViT Quantization (Ours)	-	90.3	70.10
ViT Pruning [190]	-	92.82	72.61
ViT [183]	57.07	93.58	73.81
ViT (Ours) Baseline	58.00	95.07	75.44
Dynamic ViT (Ours)	**58.71 (+0.71)**	**95.33 (+0.26)**	**76.44 (+1.0)**

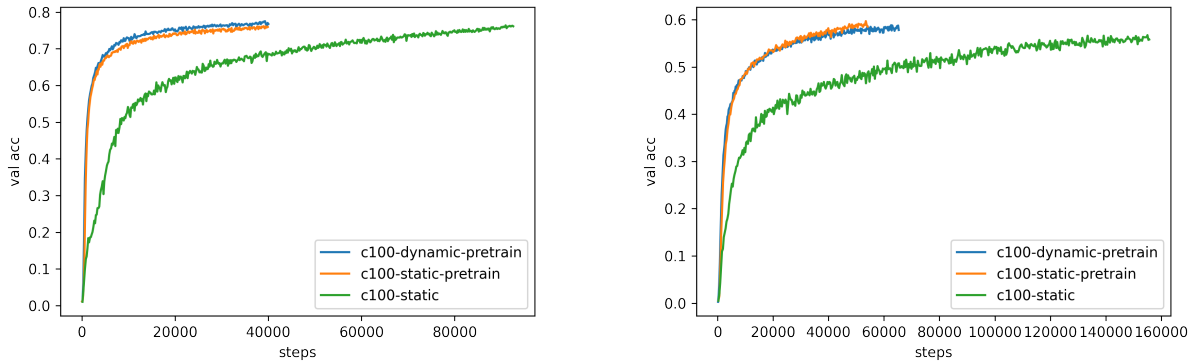
Table 8: Performance of the models on CIFAR and Tiny-ImageNet datasets

4.2.1 Transfer Learning Performance

Transfer learning is a machine learning technique that leverages knowledge gained from training one neural network on a particular task and applies it to a different but related task [3, 19, 178, 17]. Instead of starting the training process from scratch, transfer learning allows us to utilize the learned representations and weights from a pre-trained network as a starting point. By doing so, we can overcome the limitations of limited data availability and improve the performance of the target task [3, 19, 178, 17].

In transfer learning, the pre-trained network acts as a feature extractor. The lower-level layers are responsible for learning generic and low-level features while the higher-level layers are responsible for task-specific features [191, 13, 80]. Typically, these higher-level layers or fine-tuned to adapt to the new task [3]. This approach allows the network to learn high-level representations that are already optimized for general feature extraction, saving significant time and computational resources. By leveraging the pre-trained network's knowledge, transfer learning enables effective learning with smaller datasets, improves convergence speed, and boosts overall performance in various domains such as image recognition [3] and natural language processing [17, 19].

As this technique is a commonly practiced approach to training neural networks in the literature and industry, its important to investigate the effectiveness of the dynamic vision transformer in a transfer learning scenario. Hence, a model is first trained on the CIFAR-10 dataset. Then, by freezing all the layers of the model except the last layer, it is used as a feature extractor for CIFAR-100 and Tiny-ImageNet. The reason behind choosing CIFAR-10 trained model as a pre-trained layer is due to the fact that it's cheaper to train the CIFAR-10 model as shown in Table 3. Furthermore, CIFAR-100 and Tiny-ImageNet datasets are more challenging to achieve very high accuracy when compared to the CIFAR-10 dataset as shown in Table 7 and 8. As such, the effectiveness of the transfer learning technique outshines better on those challenging datasets.



(a) Performance of pretrained model on CIFAR-100

(b) Performance of pretrained model on Tiny-ImgeNet

Figure 12: The performance of CIFAR-10 pretrained model on CIFAR-100 and Tiny-ImageNet. Curves with *pretrained* extensions are the results of pretrained models. Without the *pretrained* extension, the curve is the result of models that are trained from scratch.

Figure 12 shows the validation curve of the transfer learning from CIFAR-10 to CIFAR100 (Left) and Tiny-ImageNet (Right). Firstly, as Transfer Learning promises, both the static

Model	CIFAR-100	Tiny-ImageNet
static ViT scratch	75.44	57.07
static ViT pretrained	75.97	**58.50**
dynamic ViT scratch	76.44	58.71
dynamic ViT pretrained	**76.77 (+0.8%)**	57.81 (-0.7%)

Table 9: Accuracy of CIFAR-10 pretrained models on CIFAR-100 and Tiny-ImageNet. Bold shows the highest accuracy by the pretrained model in their categories.

(Orange) and dynamic (Blue) pretrained models greatly benefit from faster learning. The models trained from scratch (green) have slow learning as they take a larger number of training steps to achieve similar accuracy to the pretrained ones.

Secondly, in the CIFAR-100 (Left) case, the dynamic pretrained model achieved better performance than the static pretrained model. In the Tiny-ImageNet (Right) case, however, the pretrained static model achieved better performance than the dynamic counterpart with a very small margin. We hypothesize that the inferior result in the Tiny-ImageNet case is the result of a drastic change in the distribution of the images. CIFAR-10 and CIFAR-100 are very similar to each other while Tiny-Imagent is the downgraded version of ImageNet [67]. As such, the policy learned by the CIFAR datasets may not be optimal for Tiny-ImageNet. Table 9 summarizes the results. In the table, *scratch* refers to the models that were trained without transfer learning.

4.2.2 Inference Speed

The throughput of the Dynamic ViT is tested in batch sizes from 1 to 1024 on Google ColabPro and RTX Titan GPUs. It is computed as the number of images processed in a microsecond. Table 7 shows the throughput of the 7, 10, and 16-layer models. As the number of layers increases, the thought also increases compared to the static one. This is due to the fact that the dynamic model activates just the needed amount of layers. Except for the 16 batch sizes, there is a large throughput increase in all batch sizes. Averaging over all batch

Model	ColabPro	RTX Titan
Dynamic ViT 7-layer	51%	23%
Dynamic ViT 10-layer	43%	35%
Dynamic ViT 16-layer	266%	46%

Table 10: Speed gain of the Dynamic ViT models on CIFAR-100 compared to the vanilla ViT.

sizes, Google ColabPro gains about 43% throughput increase while RTX Titan gains about 35%. Figure 13 and 14 show the throughput percentage increase of the 10 and 7-layer ViTs compared to fixed networks. ColabPro uses a single thread and hence, it benefits highly (up to 266%) in a bigger (16 layers) model. Figure 15 shows the speed increase in the 16-layer Dynamic ViT case. Overall, Table 10 summarizes the average percentage increase of the models on the two GPUs.

Similarly, Table 8 presents the throughput on the Google Colab for the small model from [183]. Note that this model is significantly lower than the others with just 1.8M parameters. As such, skipping a layer does not result in massive execution time reduction as the big models presented in Table 7. Overall, as Table 11 presents, the method increases throughput in all of the settings. With the exception of the 16 batch size, both GPUs show a significant throughput increase. The increase in speed is attributed to the later layers. Earlier layers process all of the inputs and later layers are selective. This might be due to the fact that lower layers are concerned with processing low-level features that are present in all inputs [25]. Figure 17 shows the fraction of the inputs in a batch that are being processed on each layer of DynamicViT-7 and DynamicViT-10. Layers 1 to 5 process all inputs. The rest of the layers process much smaller fractions of the batch inputs. Furthermore, 16 shows the fraction of the batch images processed by a layer as time goes by in the training phase.

Batch Size	static-ViT	dynamic-ViT	%
1	207	250	20%
16	3007	3057	1%
64	7676	8305	8%
256	7593	11322	49%
1024	7593	11471	51%

Table 11: Average throughput of static ViT and dynamic ViT over the CIFAR-10, CIFAR-100 & Tiny-ImageNet datasets. Overall an average of 25.8% speed boost is observed.

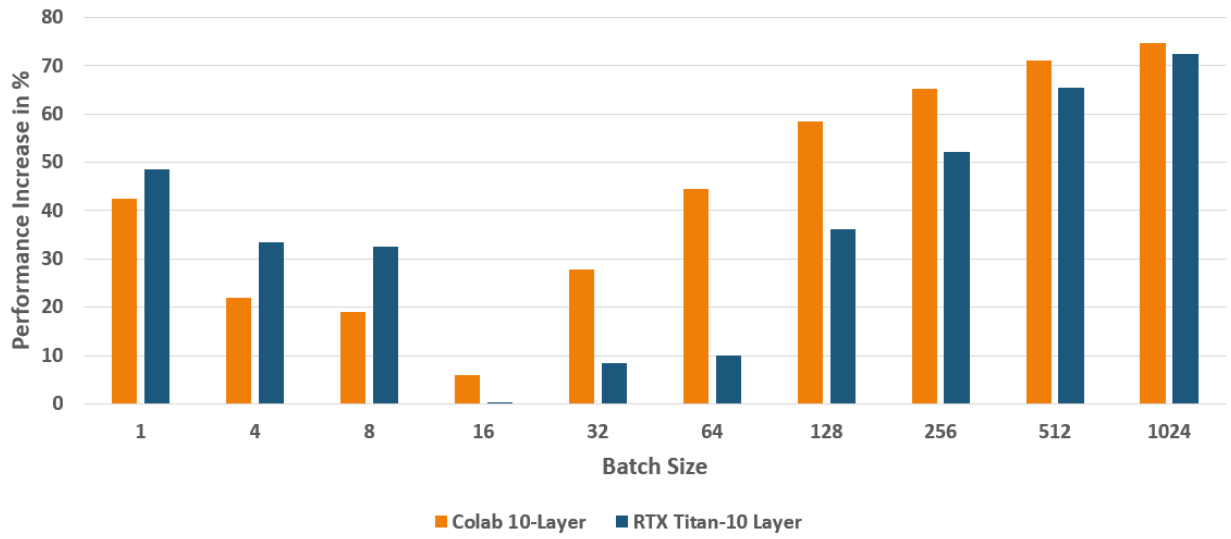


Figure 13: Throughput improvement (in %) of 10-layer Dynamic ViT over the vanilla ViT in batch sizes from 1 - 1024. Overall, an average of 38% speed boost is observed.

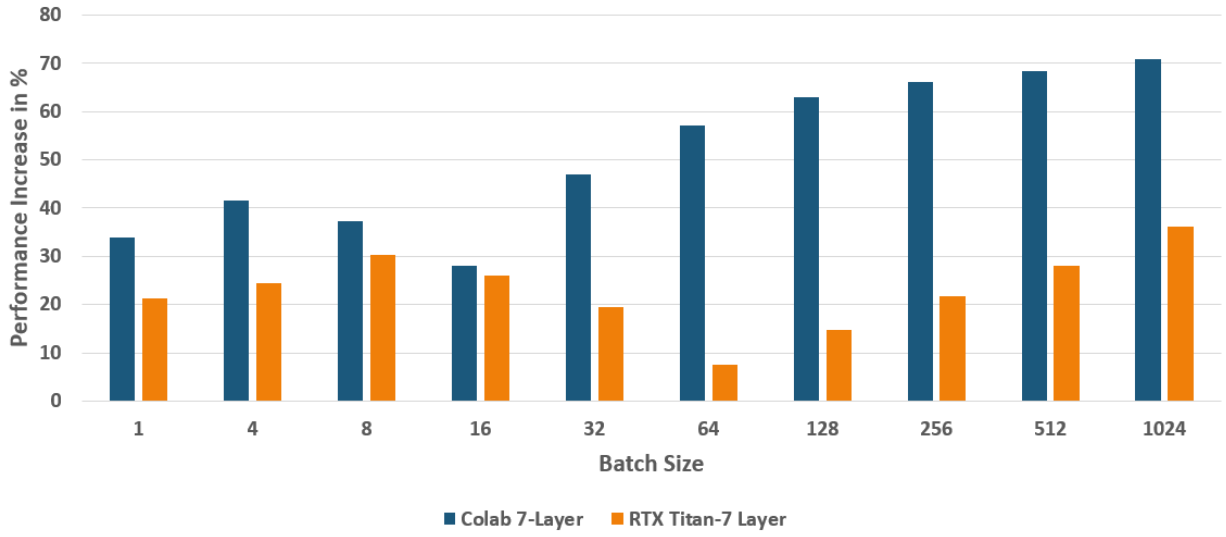


Figure 14: Throughput improvement (in %) of 7-layer Dynamic ViT over the vanilla ViT in batch sizes from 1 - 1024. Overall, an average of 48% speed boost is observed.

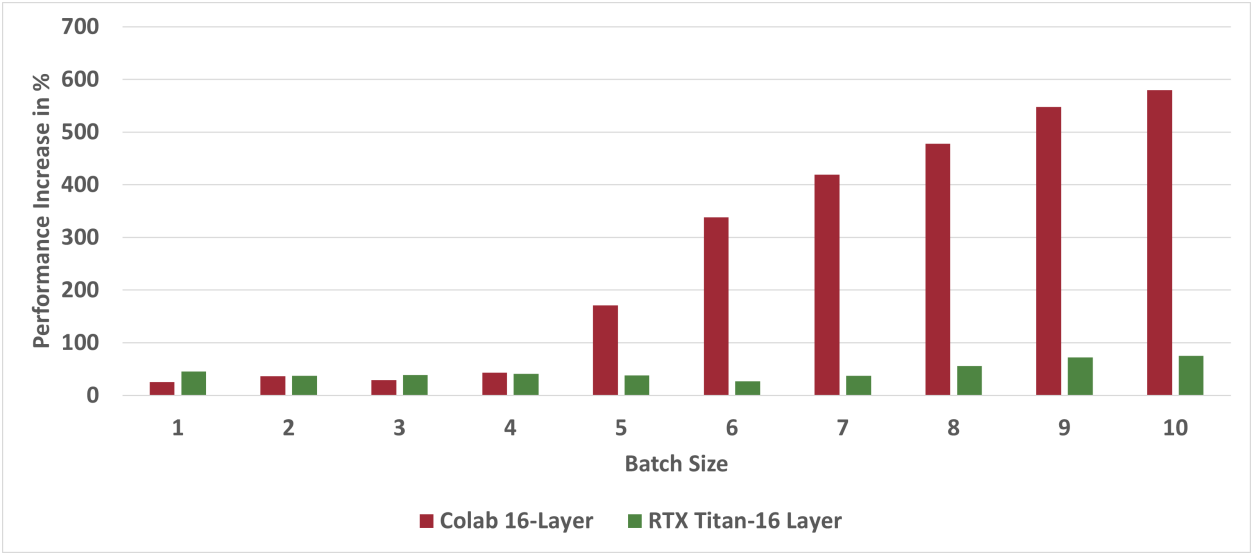


Figure 15: Throughput improvement (in %) of 16-layer Dynamic ViT over the vanilla ViT in batch sizes from 1 - 1024. Overall, an average of 145% speed boost is observed.

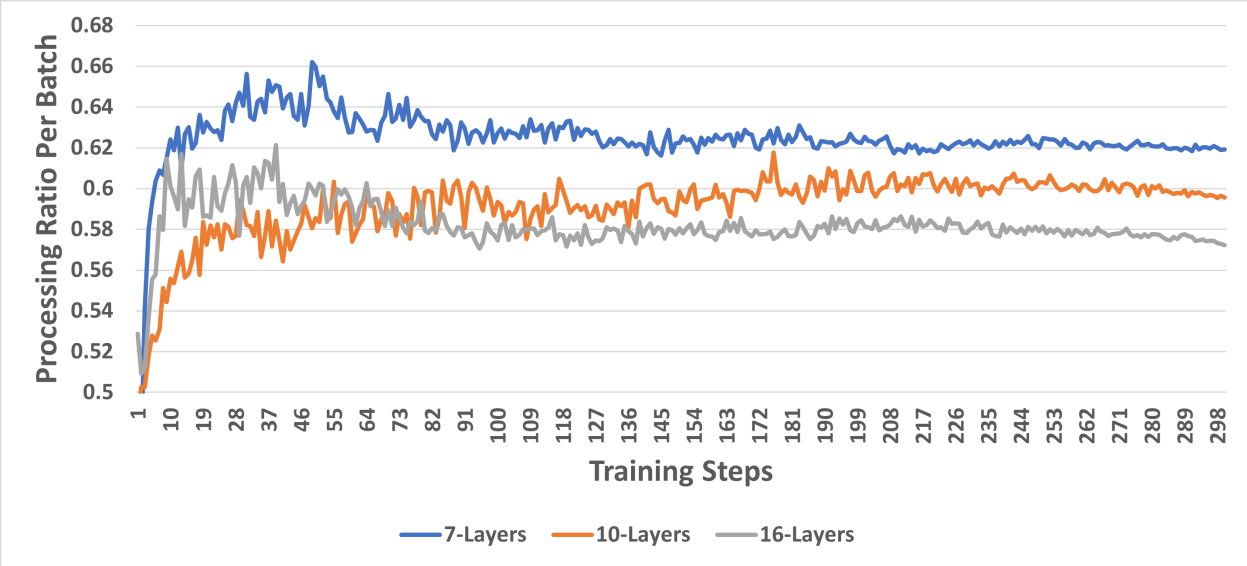


Figure 16: Processing fraction per batch of the models over the training steps: Blue (ViT-7), Orange (ViT-10), and Gray (ViT-16)

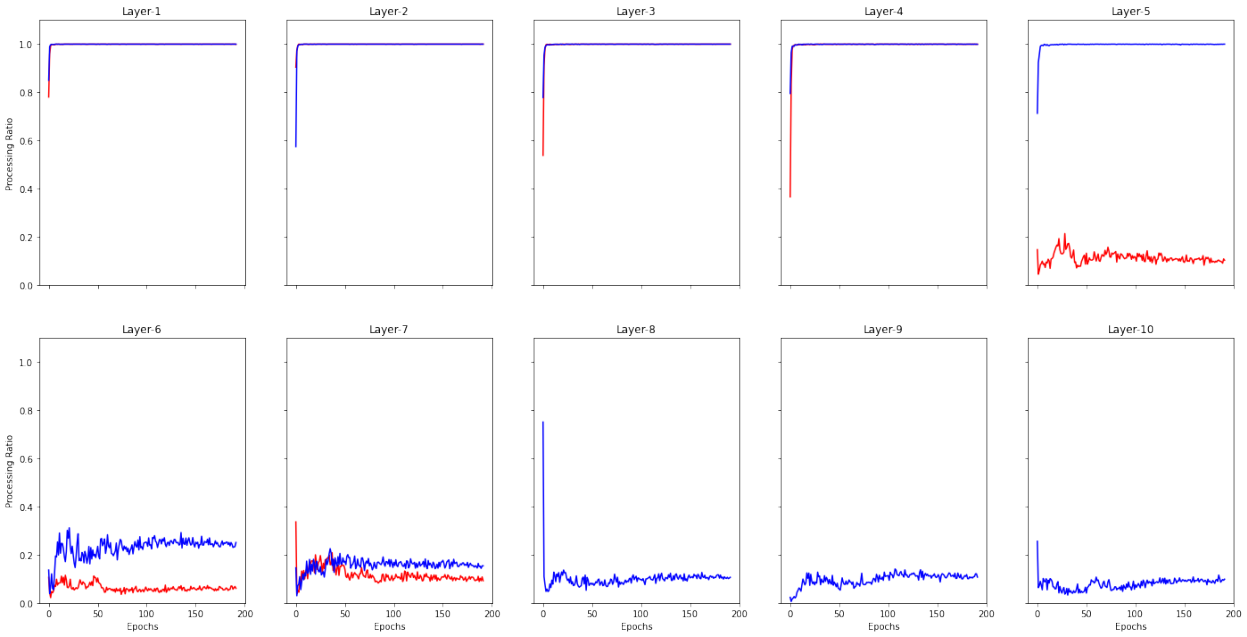


Figure 17: The processing ratio at each layer of ViT-7 (red) and Vit-10 (blue). DynamicViT-7 uses all the first 4 layers. However, starting at the 5th layer, it becomes selective by processing less than 20% of the incoming batch. Similarly, DynamicViT-10 utilizes 5 layers, and the rest is used selectively.

4.2.2.1 Source of Efficiency

Zooming in on the model's layer execution, it becomes apparent that the model aims to activate the minimal number of layers possible. Once an input is fed into the model, it traverses through the various layers. At each layer, there are two options: either 1 (pass through the layer) or 0 (skip it). This path of propagation can be represented using a string of 0s and 1s. For example, an input may pass through the first 5 layers, skip layers 6 and 7, and then pass through layers 8 and 9. This path can be expressed as:

$$1 - 1 - 1 - 1 - 1 - 0 - 0 - 1 - 1$$

To quantify these decisions made by the model, the paths for each test input can be collected to compute the distribution of these decisions. The distribution of these decisions for the CIFAR-10 dataset and the 9-layer dynamic model, based on [183], is presented in the table below (Table 12). It reveals that approximately 53% of the images are processed using only the first 5 layers. The next approximately 25% of the images are processed using a maximum of 7 layers. Consequently, collectively, around 75% of the images require 7 or fewer layers, resulting in reduced computational time compared to the static approach, which requires all 9 layers for every input.

Path ID	path	Images	Percentage
0	1-1-1-1-1-0-0-0-0	5031	52.74%
1	1-1-1-1-1-1-0-0-0	892	9.35%
2	1-1-1-1-1-0-1-0-1	586	6.14%
3	1-1-1-1-1-0-0-0-1	570	5.98%
4	1-1-1-1-1-0-0-1-0	509	5.34%
5	1-1-1-1-1-0-1-0-0	404	4.24%
6	1-1-1-1-1-1-1-0-0	354	3.71%
7	1-1-1-1-1-0-1-1-0	264	2.77%
8	1-1-1-1-1-1-1-0-1	222	2.33%
9	1-1-1-1-1-1-0-1-0	175	1.83%
10	1-1-1-1-1-1-1-1-1	166	1.74%
11	1-1-1-1-1-1-1-1-0	146	1.53%
12	1-1-1-1-1-0-1-1-1	108	1.13%
13	1-1-1-1-1-0-0-1-1	55	0.58%
14	1-1-1-1-1-1-0-0-1	39	0.41%
15	1-1-1-1-1-1-0-1-1	18	0.19%

Table 12: Layer execution path distribution of the dynamic ViT on CIFAR-10.

4.2.3 Memory Usage

The memory usage of the Transformer is analyzed on two fronts: the number of parameters and the working space of the model. The number of parameters in the Transformer can be computed in the following way:

Let I be the Input size of a single element in the sequence

Let D be the embedding dimensionality

Let H be the number of attention heads

Let d be the Query, Key, and Value projections dimension

Let S be the sequence length

$$\text{Embedding Layer:} \quad E = I \times D \tag{4.1}$$

$$\text{Self-Attention Layer:} \quad A = H \times (4 \times h \times d) \tag{4.2}$$

$$\text{FFN Layer:} \quad F = (h \times 4h) + (4h \times h) \tag{4.3}$$

$$\text{Total Layer Parameter:} \quad T = E + A + F \tag{4.4}$$

The given equations provide a computation for the number of parameters in a single Transformer layer, excluding minor layers like $LayerNorm$ which have negligible parameter counts compared to the ones mentioned. If we include the agent, the total parameter count for the layer becomes:

$$T = E + A + F + h \tag{4.5}$$

Here, the agent takes a class token (representing one element of the sequence) with a size of h and maps it to a single output, which is a binary classification decision to either skip or pass. Therefore, the total number of parameters is equivalent to h. However, as the quadratic equation in the FFN (Feed-Forward Network) layer causes h^2 to grow significantly, the contribution of h becomes less significant. For example, considering the models in Table 7 and Table 8, the number of parameters for all numbers of layers would be: In the scenario of

Model	**E + A + F**	**h**	**Increase in %**
ViT-10 (Table 7)	1.77M	384	0.021%
ViT-9 (Table 11)	0.43M	192	0.043%

Table 13: Parameter increase in % due to the agent

working space memory usage, the cost of the agent becomes even more insignificant, and its benefits outweigh the cost, leading to an improved overall memory footprint. The working space memory is determined by the sequence length S. Therefore, the following equations

depict the memory requirements during inference time with a batch size of 1. The subscript i denotes that these are memory spaces created during the inference stage.

$$\text{Embedding Layer:} \quad E_i = D \times S \tag{4.6}$$

$$\text{Self-Attention Layer:} \quad A_i = S \times H \times (h \times h) + (4 \times h \times d) \tag{4.7}$$

$$\text{FFN Layer:} \quad F = S_i \times h \tag{4.8}$$

$$\text{Total Layer Parameter:} \quad T_i = E_i + A_i + F_i \tag{4.9}$$

The above equations ignore low-level simple computations and focus on the higher-level heavy ones of the activations. The following table shows the corresponding memory spaces needed for the intermediate results in the models: In the above table, it is evident that the

Model	$\boldsymbol{E_i + A_i + F_i}$	**h**	**Increase in %**
ViT-10 (Table 7)	230M	1	$\sim 0.0\%$
ViT-9 (Table 11)	28M	1	$\sim 0.0\%$

Table 14: Working space memory requirement in % due to the agent

activations within a layer will be much greater than the layer weight size itself, and thus skipping a layer results not only in lesser usage of GPU memory but also in saving the time of creating those memories. For the additional cost h size agent in a layer, the model gains from avoiding the computation of an entire activation list of the layer. Assuming that model skips layers 53% of the time, the memory that is saved is also 53%.

Note that since the transformer model is an isomorphic model where all the layers have the same size, removing a layer results in $\frac{1}{L}$ lesser memory footprint and computational time for a model with L number of layers. As such, in the layer-skipping vision transformer scenario, the cost of the model linearly increases and decreases with the number of activated layers.

4.3 Post-training Optimization

Since there was no benchmarking on this model before, we run a random hyper-parameter search for the Plain-NSGA and joint RL-NSGA training for the post-training optimization. There are five defining factors regarding this process: generation time step, population size, skip thresholding value, mutation, and crossover probabilities. Table 15 shows the possible values of each hyperparameter. The choice of these values is shaped by the following reasoning. The result of the Generation Step and Population size below 5 and 10 was very poor in a preliminary experiment. When both factors increase, the time it takes to train also increases. As such, we limited both values to 30. The skip Threshold Step is evenly distributed between 0.01 and 0.5. Below 0.01, the search space increases significantly. As such it was limited to just 0.01. For reference, the model described in subsection 3.4 only uses 0.5 as a thresholding value. With a step value of 0.01, the model has the ability to search for the right value in 100 possible values. The mutation was evenly distributed between 0.05 and 0.8. Crossover was also evenly distributed between 0.1 and 0.5. Due to symmetry, searching beyond 0.5 is redundant.

50 random hyperparameters were sampled based on the possible values of the above settings. It was limited to just 50 because, on average, one experiment takes about eight minutes on the machine available, and increases beyond that were costly. Furthermore, as the main model is an already trained one, similar to fine-tuning[3], the post-training operation converges faster. Once the 50 hyperparameters are selected, both were applied to the CIFAR-10 and CIFAR-100 datasets. The training lasted for 6.5 hours on each dataset using an NVIDIA GTX 745 GPU, Core i7, 16GB RAM-based computer. Once the training is done, for each hyperparameter, Hypervolume was computed based on the 50 hyperparameter samples for both datasets.

Note that, in this hyperparameter search, only the NSGA-II algorithm is trained without any of the model hyperparameters being updated. This saves significant computing power

as this operation by itself is costly. Furthermore, as we show in the result tables, using these hyperparameters, the joint RL-NSGA training results in a significantly efficient model, sometimes outperforming the other tested models. Figure 18 shows the hypervolume result on each of the hyperparameters. It also shows the average of the two datasets. The final hyperparameters were selected based on the average values to minimize the data bias in the selection process. The bold values on Table 15 show the final values of the hyperparameter.

Hyperparameter	**Possible values**
Generation Step	5, 10, 15, 20, 25, **30**
Population	10, 15, 20, **25**, 30
Skip Threshold Step	0.01, 0.05, 0.1, **0.25**, 0.5
Mutation Probability	0.05, 0.2, 0.35, 0.5, **0.65**, 0.8
Crossover Probability	0.1, 0.2, 0.3, 0.4, **0.5**

Table 15: Hyperparameters for the NSGA-II. Bold numbers show the final hyperparameters chosen based on the Hypervolume Evaluation

With the RL-NSGA optimization technique, the designer gets more control as the Pareto frontier enables model selection based on the objective function values. Figure 19 shows the Pareto optimality graph for the Plain-NSGA and RL-NSGA optimized models. The RL-NSGA is even more optimized than the plain NSGA due to the fact that it has live information on the optimization and as such, the parameters can be updated accordingly to achieve maximal classification accuracy. Interestingly, both models increased performance more than the dynamic one as shown in Table 16. Overall, their accuracy is not affected significantly. However, their throughput over the plain model has significantly increased as shown in Table 17.

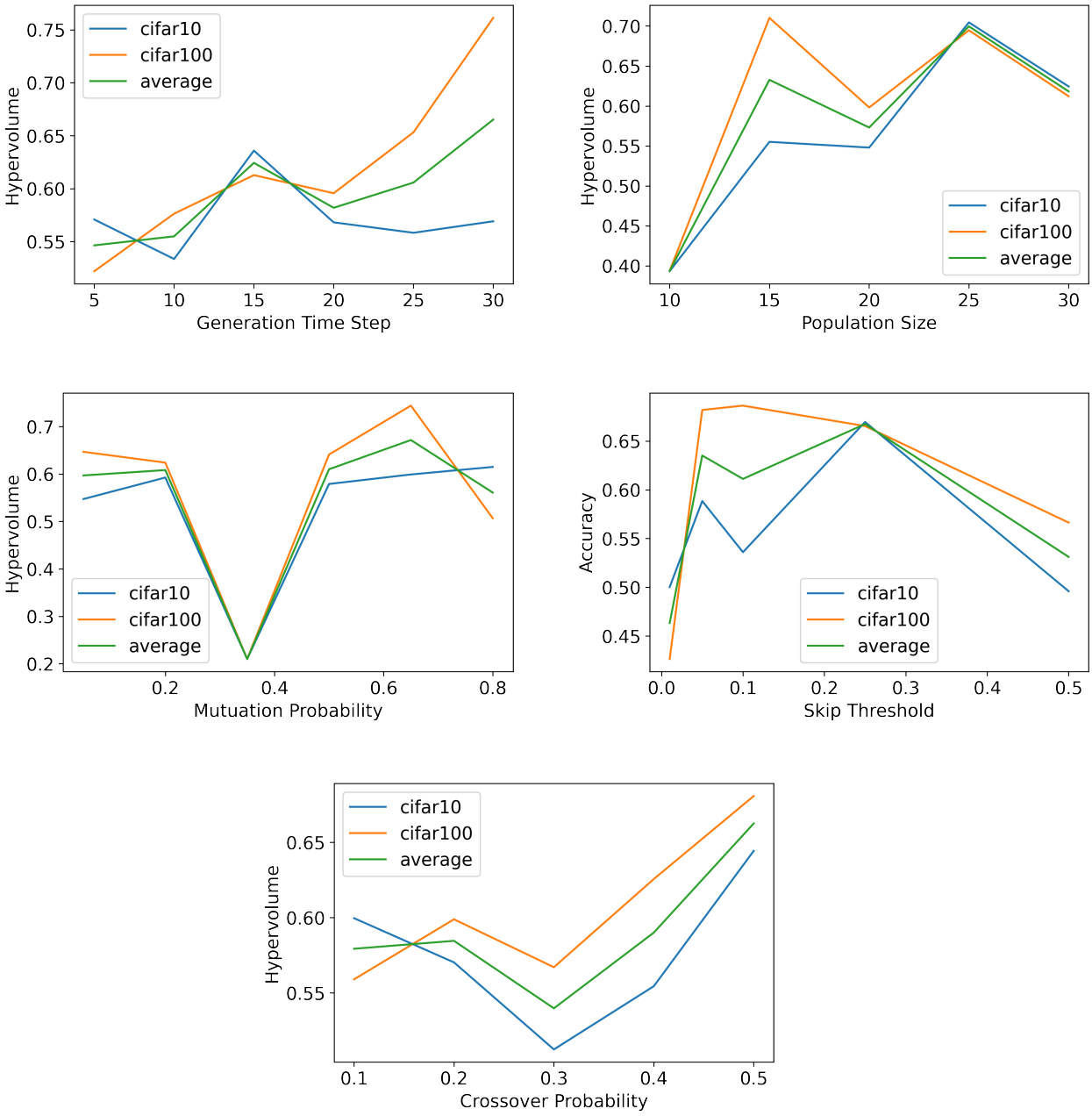
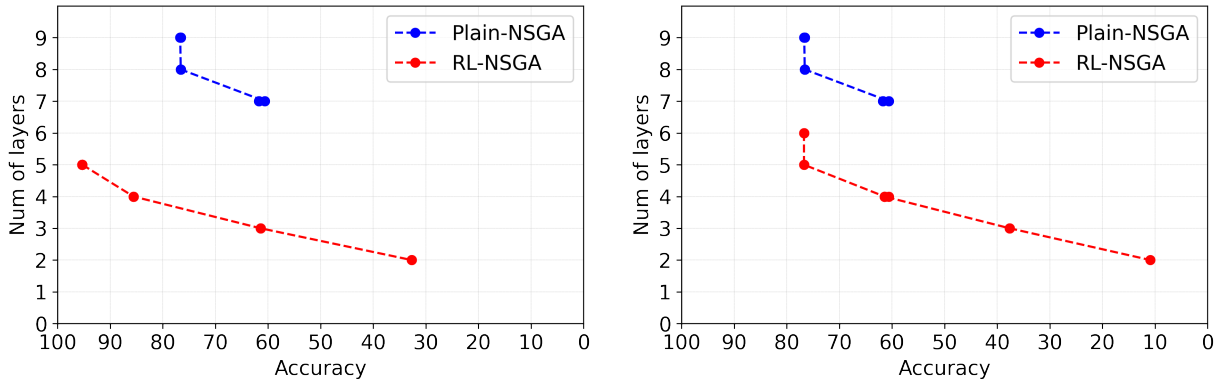


Figure 18: Accuracy vs Hyperparameter: Hyperparameter Search for the joint NSGA-II-RL Post Training Optimization. The best values were attended at the generation step of 30, a population size of 25, a mutation probability of 0.65, a skip threshold step of 0.25, and a crossover probability of 0.5.



(a) Pareto front of CIFAR-10 dataset

(b) Pareto front of CIFAR-100 dataset

Figure 19: NSGA-II optimized Pareto fronts (Accuracy - Num. of Layers) on CIFAR datasets.

Figure 19 shows the collections of solutions found by the late-stage optimization using NSGA-II using CIFAR-10 (Left) and CIFAR-100 (Right). The red dots show the Pareto frontiers found by the joint RL-NSGA optimization while the blue dots are of the plain NSGA optimization. Closer to the origin of the coordinate (100, 0) shows that an ideal model that does not use any layer but accurately labels the test sets. Rl-NSGA-based optimization, in both datasets, results in Pareto frontiers that are closer to the ideal setting when compared to the plain NSGA training.

Furthermore, the plain dynamic model might activate unnecessary layers for a given input as explained in subsection 3.5. To evaluate if this problem has been solved, the operation proposed in subsection 3.5 was re-run on the new RL-NSGA optimized model. The result is shown in Table 18. In the CIFAR-10 and CIFAR-100 datasets, we observed reductions of 47% and 42%, respectively, in unnecessary operations. This improvement in efficiency can be attributed to the model restructuring itself and utilizing layers more efficiently through the RL-NSGA-based optimization.

Model	CIFAR-10	CIFAR-100	Num. of Layers
ViT [183]	93.58	73.81	9
ViT (Ours) Baseline	95.07	75.44	9
Dynamic ViT (Ours)	**95.33**	76.44	9
RL-NSGA	95.29	**76.73**	6 (↓ 33%)
Plain-NSGA	95.28	76.70	8 (↓ 11%)

Table 16: Performance of NSGA-optimized models on CIFAR-10 and CIFAR-100 datasets.

Batch Size	static-ViT	dynamic-ViT	RL-NSGA	dynamic ViT %	RL-NSGA %
1	207	250	330	20%	32%
16	3007	3057	3919	1%	28%
64	7676	8305	11399	8%	37%
256	7593	11322	15685	49%	58%
1024	7593	11471	16248	51%	64%

Table 17: Average throughput of static ViT, dynamic ViT, and RL-NSGA ViT over the CIFAR-10, CIFAR-100 & Tiny-ImageNet datasets. The efficiency of the proposed model increased from 25.8 to 43.8 percent on average.

Dataset - **Class ID**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Ave.**
LS ViT CIFAR-10	70	65	68	67	67	66	65	73	68	65	**61**
RL-NSGA CIFAR-10	15	11	16	16	15	11	15	17	11	9	**14**
LS ViT CIFAR-100	50	56	48	50	53	49	52	50	54	56	**52**
RL-NSGA CIFAR-100	9	8	9	9	11	11	13	11	8	12	**10**

Table 18: Fixed unnecessary layer activation by the joint RL-NSGA optimization. *LS ViT* is the layer skipping ViT. In the CIFAR-10 case, the unnecessary activations lowered from 61% to 14% whereas in the CIFAR-100 case, it reduced from 52% to 10%.

Finally, the model is optimized to select the appropriate thresholding values for each of the layers through the optimization technique. Table 19 shows four samples out of the Pareto fronts for the CIFAR-100. As their accuracy decreases, the path discovered will have lesser activation of the layers.

Accuracy	**Path**
76.33%	1-1-1-1-1-1-0.25-0-0-0
61.42%	1-1-0-1-1-0-0-0-0
37.65%	1-1-0-0-0.75-0-0-0-0
10.91%	1-0-0-0-0.75-0-0-0-0

Table 19: Accuracy and Path of four samples of the RL-NSGA optimized models. 0 shows no activation ever and the layer can be removed. 1 shows the layer must be activated always. The decimal points show the newly optimized skipping threshold.

Overall, the NSGA-based post-optimization results in an efficient model. When the RL agent is fine-tuned along with the NSGA, the model becomes even more efficient, decreasing the number of layers by 33%. It also gives control over how much accuracy should be sacrificed with the given depth. Hence, similar to weight pruning and other methods, the model can be minimized to utilize the small memory space available in small devices.

4.4 Discussion

4.4.1 Key Findings

- **Most images do not need some layers:** Table 12 shows that about 53% of the images are processed using only the first five layers. For the next 10% of images add one more layer on top of the first five layers. That is the primary reason behind the efficiency gain.

- **Efficient Skip Policy:** There is no work that showed layer skipping can be efficiently applied without compromising on the performance of the model. The RL-based Dynamic Transformer Network even performs better than the static model while being 53% efficient on average on different batch sizes. As such, it can be concluded that there is redundant or unnecessary representation in the static network.

- **Skip Threshold Value Optimization:** As shown in Table 19, 0.5 may not be optimal. As such, at least in systems like this work, optimizing it to the right value results in better performance.

4.4.2 Limitations

- **Extra Memory Space:** Compared to the static transformer, the dynamic transformer does require additional memory space for the agent, both in the training and inference phases. However, in real-world applications, considering the overall size of the model and the working space required, the amount of extra memory needed is comparatively small as described in the subsection 4.2.3. Therefore, this additional memory requirement can generally be considered negligible.

- **Training Time Extra Computation:** During the training phase, the agent occasionally makes errors in its skipping decisions, leading to the activation of more layers than necessary. Additionally, in a given batch, back-propagation is applied to all activated layers regardless of how many inputs a layer processes. Consequently, the dynamic

transformer at training time closely resembles the static network as it activates all the layers. This similarity, combined with the additional computational cost incurred by the agent and the sub-batch extraction operation, results in increased training time. However, in practical scenarios, this additional cost can generally be disregarded since training is a one-time expense compared to the long-term inference cost.

- **Transfer Learning Scenario:** Although the model achieved very good performance in the transfer learning test, the result on Tiny-ImageNet have shown lesser performance by about 0.70%. This shows that, although the difference is very small, it might require retraining to alleviate the issue.

- **Post-training RL-NSGA Training:** The additional training of the NSGA and the model can be seen as an additional cost. Although, compared to the normal model training, this operation is minor, the hyperparameter search might be significantly costly. As such, developers might compare the training and deployment cost to identify if it is worth applying such operation post-training.

4.4.3 Real-world Examples

The contributions of this approach have direct implications for real-world applications. For instance, in time-sensitive tasks like self-driving, where images need to be analyzed rapidly to make decisions, the benefits are significant. By utilizing a 10-layer model as shown in Table 7 and employing layer skipping, a 53% increase in processing speed can be expected. This improvement can be likened to the effect of using a cheaper GPU from the previous year's model while maintaining the same throughput.

In the context of search engines, where inputs can be processed in large batches as long as memory constraints allow, the advantages are even more pronounced. Assuming the utilization of the same 10-layer model, a 70% increase in throughput can be achieved when processing documents. This increase in throughput is accompanied by a significant reduction in power consumption, thanks to the decreased activation of layers.

To illustrate the tangible impact, as of June 3rd, 2023, OpenAI offers the GPT-4 language model with an 8k context at a cost of \$0.03/1k tokens. By implementing layer skipping, this cost can be reduced to %0.009/1k tokens under batched input processing and %0.018/1k tokens under a single input processing. Consequently, users can now afford to subscribe to the more powerful 32k context GPT-4 model while staying within their budget.

Chapter 5 Conclusion

The transformer architecture has revolutionized deep learning by significantly advancing the field of natural language processing, computer vision, graph, and tabular data processing and powering various innovations. Unlike traditional recurrent or convolutional neural networks, transformers employ a self-attention mechanism that allows them to capture global dependencies in input sequences. This mechanism enables transformers to excel in tasks like machine translation, sentiment analysis, question answering, and language generation. The ability to model long-range dependencies effectively and capture contextual relationships has led to improved performance in various domains, leading to massive adoption of the architecture. Moreover, transformers can be pre-trained on large amounts of unlabeled data and fine-tuned for specific downstream tasks, leading to state-of-the-art results.

However, the remarkable success of transformer-based models comes at the cost of increased demand for computational power. Transformers often have significantly more parameters than traditional models, requiring larger computational resources for training and inference. The self-attention mechanism scales quadratically with the input sequence length, making it computationally expensive for long sequences. The demand for computational resources arises from the need to process massive amounts of data, perform complex matrix multiplications, and optimize the high-dimensional parameter space of transformers. The widening economic disparity and carbon emissions are also critical factors to consider in the adoption of this model. As such, addressing the computational challenges associated with transformers is crucial to ensure their widespread adoption and to continue pushing the boundaries of deep learning in NLP and other domains. Reiterating the research question, we can conclude that:

- **How can the training process of Transformer models be modified to enable**

adaptiveness and the ability to skip layers, resulting in more efficient inference? An RL-based adaptive mechanism was developed to dynamically identify and skip irrelevant layers based on the input characteristics. This mechanism allowed the model to adapt and optimize its computations according to the specific input, resulting in improved efficiency and reduced computational demand. The RL-based dynamic vision transformer showed improved performance over the static network while being significantly efficient in the inference phase.

- **How will the technique generalize across different datasets and model sizes in the context of a vision problem?** The generalizability of the optimized dynamic neural network was assessed on various datasets (CIFAR-10, CIFAR-100, and Tiny-ImageNet), ensuring that the benefits of layer skipping were not limited to specific datasets but could be applied across different datasets. In each of the dataset categories, the technique achieved better performance. The technique is also shown to be effective on models with different depths and widths, including transfer learning. Moreover, the method has been shown to be effective in transfer learning scenarios.

- **How does the proposed system improve efficiency in terms of both time and space?** The impact of layer skipping on the execution time of the model was thoroughly investigated and compared to the baseline static transformer models. Both in computational time and memory footprint, it achieves about 53%, averaged over different settings. Especially on higher batch sizes, the model achieves up to 600% inference speed increase when compared to the static one. Similarly, as the layers are activated sparsely, the working memory will only increase linearly with the number of activated layers. As such, on average, 53% memory footprint can be expected across different settings. This model then can be pruned even more to become efficient through the post-training RL-NSGA technique. This analysis provided valuable insights into the computational advantages associated with the layer-skipping approach.

5.1 Recommendation

- **Exploration of Different RL Algorithms:** While this research employs a particular RL algorithm for training the layer-skipping agent, it would be beneficial to explore and compare the performance of different RL algorithms. Alternative algorithms, such as Proximal Policy Optimization (PPO) may offer improved training stability, faster convergence, or better exploration-exploitation trade-offs even though it incurs more computational cost. Conducting comparative studies with these algorithms can provide insights into the most suitable RL approach for training the layer-skipping agent in the dynamic Transformer.

- **Investigation of Adaptive Layer Scaling:** In this research, the dynamic ViT dynamically skips layers based on the RL agent's decisions. However, exploring the possibility of adapting the layer size or scaling the number of attention heads based on the input's relevance could be an interesting direction for future work. By adjusting the capacity of individual layers or attention heads, the dynamic ViT could further optimize computational efficiency by allocating more resources to critical parts of the input and reducing redundancy in less informative sections.

- **Developing Specialized Non-Zero Kernel:** In the current implementation, the widely-used *nonzero* function from popular deep learning frameworks is utilized to extract inputs that don't need to be processed in the following layer. However, it has been observed that this function introduces noticeable performance overhead in smaller models. To mitigate this cost, one potential solution is the utilization of a specialized hardware-based kernel that can directly operate on the GPU. By leveraging such a dedicated kernel, the performance impact associated with the *nonzero* function can be minimized, resulting in improved efficiency for small models.

5.2 Future Works

- **Robustness analysis:** Future work can focus on evaluating the robustness of the proposed layer-skipping dynamic ViT network. Robustness analysis involves studying the model's performance under various perturbations, such as adversarial attacks or input variations. By subjecting the model to different challenging scenarios, researchers can assess its ability to maintain performance while dynamically skipping layers. This analysis will provide insights into the model's resilience and help identify potential vulnerabilities or areas for improvement.

- **Dynamic attention mechanisms:** While the proposed dynamic ViT network focuses on layer skipping as a means to reduce computational requirements, future work can investigate the integration of dynamic attention mechanisms within the model. Attention mechanisms have played a crucial role in the success of transformer architectures, allowing them to capture contextual information effectively. By introducing dynamic attention, the model can adaptively allocate attention to different regions or tokens within an input, further optimizing the computational resources and improving performance. Exploring techniques such as sparse attention, adaptive attention, or learned attention routing can enhance the efficiency and effectiveness of the dynamic ViT network in capturing relevant information while reducing computational costs.

- **Explainablity:** While the dynamic ViT architecture presents significant improvements in computational efficiency and performance, understanding the decision-making process of the RL agent can provide valuable insights and build trust in its choices. Exploring techniques such as attention mechanisms or visualization methods that highlight the importance of specific layers or attention heads can help interpret the agent's behavior.

References

[1] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[2] Y. Le and X. S. Yang, "Tiny imagenet visual recognition challenge," 2015.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *ArXiv*, vol. abs/2010.11929, 2021.

[4] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *Sn Computer Science*, vol. 2, 2021.

[5] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, vol. 19, pp. 305–307, 2017.

[6] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," *ArXiv*, vol. abs/2212.04356, 2022.

[7] D. Min, D. B. Lee, E. Yang, and S. J. Hwang, "Meta-stylespeech : Multi-speaker adaptive text-to-speech generation," in *International Conference on Machine Learning*, 2021.

[8] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," *ArXiv*, vol. abs/2102.12092, 2021.

[9] J. M. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Zídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. A. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu,

P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, pp. 583 – 589, 2021.

[10] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. J. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *ArXiv*, vol. abs/2005.14165, 2020.

[12] S. Hooker, "The hardware lottery," *Communications of the ACM*, vol. 64, pp. 58 – 65, 2020.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[14] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European Conference on Computer Vision*, 2014.

[15] C. Crawl, "Common crawl dataset," 2023. [Online]. Available: https://commoncrawl.org/

[16] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J. Nie, and J. rong Wen, "A survey of large language models," *ArXiv*, vol. abs/2303.18223, 2023.

[17] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *North American Chapter of the Association for Computational Linguistics*, 2018.

[18] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *ArXiv*, vol. abs/1905.11946, 2019.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[20] K. Lu, A. Grover, P. Abbeel, and I. Mordatch, "Pretrained transformers as universal computation engines," *ArXiv*, vol. abs/2103.05247, 2021.

[21] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," in *Neural Information Processing Systems*, 2019.

[22] I. Padhi, Y. Schiff, I. Melnyk, M. Rigotti, Y. Mroueh, P. L. Dognin, J. Ross, R. Nair, and E. Altman, "Tabular transformers for modeling multivariate time series," *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3565–3569, 2020.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[24] J. Chung, Çaglar Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *ArXiv*, vol. abs/1412.3555, 2014.

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

[26] OpenAI, "Gpt-4 technical report," *ArXiv*, vol. abs/2303.08774, 2023.

[27] A. Gupta, D. Chugh, Anjum, and R. Katarya, "Automated news summarization using transformers," *ArXiv*, vol. abs/2108.01064, 2021.

[28] J. Chun, "Sentimentarcs: A novel method for self-supervised sentiment analysis of time series shows sota transformers can struggle finding narrative arcs," *ArXiv*, vol. abs/2110.09454, 2021.

[29] C. Raffel, N. M. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *ArXiv*, vol. abs/1910.10683, 2019.

[30] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International Conference on Machine Learning*. PMLR, 2021, pp. 10 347–10 357.

[31] Y. Wang, J. Xu, and Y. Sun, "End-to-end transformer based model for image captioning," in *AAAI Conference on Artificial Intelligence*, 2022.

[32] J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millican, M. Reynolds, R. Ring, E. Rutherford, S. Cabi, T. Han, Z. Gong, S. Samangooei, M. Monteiro, J. Menick, S. Borgeaud, A. Brock, A. Nematzadeh, S. Sharifzadeh, M. Binkowski, R. Barreira, O. Vinyals, A. Zisserman, and K. Simonyan, "Flamingo: a visual language model for few-shot learning," *ArXiv*, vol. abs/2204.14198, 2022.

[33] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," in *International Conference on Machine Learning*, 2021.

[34] A. Maharana, Q. Tran, F. Dernoncourt, S. Yoon, T. Bui, W. Chang, and M. Bansal, "Multimodal intent discovery from livestream videos," in *Findings of the Association for Computational Linguistics: NAACL 2022*. Seattle, United States: Association for Computational Linguistics, Jul. 2022, pp. 476–489. [Online]. Available: https://aclanthology.org/2022.findings-naacl.36

[35] A. Urooj, A. Mazaheri, N. Da vitoria lobo, and M. Shah, "MMFT-BERT: Multimodal Fusion Transformer with BERT Encodings for Visual Question Answering," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 4648–4660. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.417

[36] D. Serdyuk, O. Braga, and O. Siohan, "Transformer-based video front-ends for audio-visual speech recognition," in *Interspeech*, 2022.

[37] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. J. Colwell, and A. Weller, "Rethinking attention with performers," *ArXiv*, vol. abs/2009.14794, 2020.

[38] A. S. Luccioni, S. Viguier, and A.-L. Ligozat, "Estimating the carbon footprint of bloom, a 176b parameter language model," *ArXiv*, vol. abs/2211.02001, 2022.

[39] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 3645–3650. [Online]. Available: https://aclanthology.org/P19-1355

[40] A. Lacoste, A. S. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *ArXiv*, vol. abs/1910.09700, 2019.

[41] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. B. Rao, P. Barnes, Y. Tay, N. M. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. C. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. García, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai,

M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Díaz, O. Firat, M. Catasta, J. Wei, K. S. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *ArXiv*, vol. abs/2204.02311, 2022.

[42] L. B. Heguerte, A. Bugeau, and L. Lannelongue, "How to estimate carbon footprint when training deep learning models? a guide and review," 2023.

[43] M. Brundage, S. Avin, J. Clark, H. Toner, P. Eckersley, B. Garfinkel, A. Dafoe, P. Scharre, T. Zeitzoff, B. Filar, H. Anderson, H. Roff, G. C. Allen, J. Steinhardt, C. Flynn, S. Ó. hÉigeartaigh, S. Beard, H. Belfield, S. Farquhar, C. Lyle, R. Crootof, O. Evans, M. Page, J. J. Bryson, R. V. Yampolskiy, and D. Amodei, "The malicious use of artificial intelligence: Forecasting, prevention, and mitigation," *ArXiv*, vol. abs/1802.07228, 2018.

[44] A. Hagerty and I. Rubinov, "Global ai ethics: A review of the social impacts and ethical implications of artificial intelligence," *ArXiv*, vol. abs/1907.07892, 2019.

[45] T. Gries and W. A. Naudé, "Artificial intelligence, jobs, inequality and productivity: Does aggregate demand matter?" *Labor: Human Capital eJournal*, 2018.

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[47] M. Zhu, Y. Tang, and K. Han, "Vision transformer pruning," 2021.

[48] Q. Zhang, S. Zuo, C. Liang, A. Bukharin, P. He, W. Chen, and T. Zhao, "Platon: Pruning large transformer models with upper confidence bound of weight importance," in *International Conference on Machine Learning*. PMLR, 2022, pp. 26 809–26 823.

[49] Z. Yuan, C. Xue, Y. Chen, Q. Wu, and G. Sun, "Ptq4vit: Post-training quantization framework for vision transformers," *ArXiv*, vol. abs/2111.12293, 2021.

[50] Z. Liu, Y. Wang, K. Han, S. Ma, and W. Gao, "Post-training quantization for vision transformer," in *Neural Information Processing Systems*, 2021.

[51] Y. Ding, H. Qin, Q.-Y. Yan, Z. Chai, J. Liu, X. Wei, and X. Liu, "Towards accurate post-training quantization for vision transformer," *Proceedings of the 30th ACM International Conference on Multimedia*, 2022.

[52] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, "Scatterbrain: Unifying sparse and low-rank attention," in *Neural Information Processing Systems*, 2021.

[53] S. Cahyawijaya, "Greenformers: Improving computation and memory efficiency in transformer models via low-rank approximation," *ArXiv*, vol. abs/2108.10808, 2021.

[54] A. Kumar, "Vision transformer compression with structured pruning and low rank approximation," *ArXiv*, vol. abs/2203.13444, 2022.

[55] J. Dass, S. Wu, H. Shi, C. Li, Z. Ye, Z. Wang, and Y. Lin, "Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention," *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 415–428, 2022.

[56] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *ArXiv*, vol. abs/1503.02531, 2015.

[57] Y. Rao, W. Zhao, B. Liu, J. Lu, J. Zhou, and C.-J. Hsieh, "Dynamicvit: Efficient vision transformers with dynamic token sparsification," in *Neural Information Processing Systems*, 2021.

[58] C. Wei, B. Duke, R. Jiang, P. Aarabi, G. W. Taylor, and F. Shkurti, "Sparsifiner: Learning sparse instance-dependent attention for efficient vision transformers," *ArXiv*, vol. abs/2303.13755, 2023.

[59] J. Qiu, H. Ma, O. Levy, S. Yih, S. Wang, and J. Tang, "Blockwise self-attention for long document understanding," *ArXiv*, vol. abs/1911.02972, 2019.

[60] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *ArXiv*, vol. abs/1904.10509, 2019.

[61] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and L. Kaiser, "Universal transformers," *ArXiv*, vol. abs/1807.03819, 2018.

[62] M. Elbayad, J. Gu, E. Grave, and M. Auli, "Depth-adaptive transformer," *ArXiv*, vol. abs/1910.10073, 2020.

[63] Y. Liu, F. Meng, J. Zhou, Y. Chen, and J. Xu, "Explicitly modeling adaptive depths for transformer," *ArXiv*, vol. abs/2004.13542, 2020.

[64] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, pp. 7436–7456, 2021.

[65] S. Laskaridis, A. Kouris, and N. D. Lane, "Adaptive inference through early-exit networks: Design, challenges and directions," *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, 2021.

[66] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS*, 1989.

[67] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[68] X. Wang, F. Yu, Z.-Y. Dou, and J. Gonzalez, "Skipnet: Learning dynamic routing in convolutional networks," *ArXiv*, vol. abs/1711.09485, 2017.

[69] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in *International Conference on Machine Learning*, 2017.

[70] Z. Liu, H. Mao, C. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," 2022.

[71] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[72] M. Minsky and S. Papert, "Perceptrons - an introduction to computational geometry," 1969.

[73] K. P. Murphy, "Machine learning - a probabilistic perspective," in *Adaptive computation and machine learning series*, 2012.

[74] G. V. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989.

[75] H. T. Siegelmann and E. Sontag, "On the computational power of neural nets," in *Annual Conference Computational Learning Theory*, 1992.

[76] J. W. Siegel and J. Xu, "Approximation rates for neural networks with general activation functions," *Neural networks : the official journal of the International Neural Network Society*, vol. 128, pp. 313–321, 2019.

[77] M. Raghu, B. Poole, J. M. Kleinberg, S. Ganguli, and J. N. Sohl-Dickstein, "On the expressive power of deep neural networks," *ArXiv*, vol. abs/1606.05336, 2016.

[78] K. Fukushima, "Cognitron: A self-organizing multilayered neural network," *Biological Cybernetics*, vol. 20, pp. 121–136, 1975.

[79] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[80] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer, 2014, pp. 818–833.

[81] A. Babenko, A. Slesarev, A. Chigorin, and V. Lempitsky, "Neural codes for image retrieval," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 584–599.

[82] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, vol. 1, pp. 339–356, 1988.

[83] Y. Bengio, P. Y. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5 2, pp. 157–66, 1994.

[84] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural Computation*, vol. 12, pp. 2451–2471, 2000.

[85] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *ArXiv*, vol. abs/1409.3215, 2014.

[86] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1764–1772. [Online]. Available: https://proceedings.mlr.press/v32/graves14.html

[87] Y. Hua, Z. Zhao, R. Li, X. Chen, Z. Liu, and H. Zhang, "Deep learning with long short-term memory for time series prediction," *IEEE Communications Magazine*, vol. 57, pp. 114–119, 2018.

[88] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[89] A. Graves, "Adaptive computation time for recurrent neural networks," *ArXiv*, vol. abs/1603.08983, 2016.

[90] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals, "Listen, attend and spell," *ArXiv*, vol. abs/1508.01211, 2015.

[91] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. F. Diamos, E. Elsen, R. J. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Ng, "Deep speech: Scaling up end-to-end speech recognition," *ArXiv*, vol. abs/1412.5567, 2014.

[92] S. Ö. Arik, M. Chrzanowski, A. Coates, G. F. Diamos, A. Gibiansky, Y. Kang, X. Li, J. Miller, A. Ng, J. Raiman, S. Sengupta, and M. Shoeybi, "Deep voice: Real-time neural text-to-speech," in *International Conference on Machine Learning*, 2017.

[93] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. R. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *ArXiv*, vol. abs/1609.08144, 2016.

[94] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. Dauphin, "Convolutional sequence to sequence learning," in *International Conference on Machine Learning*, 2017.

[95] P. Kouris, G. Alexandridis, and A. Stafylopatis, "Abstractive text summarization based on deep learning and semantic content generalization," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 5082–5092. [Online]. Available: https://aclanthology.org/P19-1501

[96] S. Karita, N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. Yalta, R. Yamamoto, X. fei Wang, S. Watanabe, T. Yoshimura, and W. Zhang, "A comparative study on transformer vs rnn in speech applications," *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pp. 449–456, 2019.

[97] Y. Hao, L. Dong, F. Wei, and K. Xu, "Self-attention attribution: Interpreting information interactions inside transformer," in *AAAI Conference on Artificial Intelligence*, 2020.

[98] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations*, 2013.

[99] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," in *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp, Eds., vol. 13. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf

[100] J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *ArXiv*, vol. abs/1607.06450, 2016.

[101] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, 2016.

[102] S. Cai, Y. Shu, W. Wang, and B. C. Ooi, "Isbnet: Instance-aware selective branching network," *ArXiv*, vol. abs/1905.04849, 2019.

[103] Y. Han, Z. Yuan, Y. Pu, C. Xue, S. Song, G. Sun, and G. Huang, "Latency-aware spatial-wise dynamic networks," *ArXiv*, vol. abs/2210.06223, 2022.

[104] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv: Learning*, 2016.

[105] B. Liu, M. Wang, H. Foroosh, M. F. Tappen, and M. Pensky, "Sparse convolutional neural networks," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814, 2015.

[106] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, pp. 241:1–241:124, 2021.

[107] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *ArXiv*, vol. abs/1802.06367, 2018.

[108] S. Jaszczur, A. Chowdhery, A. Mohiuddin, L. Kaiser, W. Gajewski, H. Michalewski, and J. Kanerva, "Sparse is enough in scaling transformers," in *Neural Information Processing Systems*, 2021.

[109] W. Fedus, B. Zoph, and N. M. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *ArXiv*, vol. abs/2101.03961, 2021.

[110] Y. Nan, H. Zhang, Y. Zeng, J. Zheng, and Y. Ge, "Faster and accurate green pepper detection using nsga-ii-based pruned yolov5l in the field environment," *Comput. Electron. Agric.*, vol. 205, p. 107563, 2023.

[111] X. Wang, F. Yu, Z.-Y. Dou, and J. Gonzalez, "Skipnet: Learning dynamic routing in convolutional networks," *ArXiv*, vol. abs/1711.09485, 2018.

[112] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and L. Kaiser, "Universal transformers," *ArXiv*, vol. abs/1807.03819, 2019.

[113] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 16, pp. 285–286, 2005.

[114] B. J. A. Kröse, "Learning from delayed rewards," *Robotics Auton. Syst.*, vol. 15, pp. 233–235, 1995.

[115] R. S. Sutton, D. A. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *NIPS*, 1999.

[116] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016.

[117] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *ArXiv*, vol. abs/1707.06347, 2017.

[118] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[119] C. Browne, E. J. Powley, D. Whitehouse, S. M. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, pp. 1–43, 2012.

[120] Y. Ban, Y. Yan, A. Banerjee, and J. He, "Ee-net: Exploitation-exploration neural networks in contextual bandits," *ArXiv*, vol. abs/2110.03177, 2021.

[121] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, 2002.

[122] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

[123] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *CoRR*, vol. abs/1506.02438, 2015.

[124] O. Kramer, "Genetic algorithm essentials," in *Studies in Computational Intelligence*, 2017.

[125] K. Deb, "Multi-objective optimization using evolutionary algorithms," in *Wiley-Interscience series in systems and optimization*, 2001.

[126] M. Ghaderian and F. Veysi, "Multi-objective optimization of energy efficiency and thermal comfort in an existing office building using nsga-ii with fitness approximation: A case study," *Journal of building engineering*, vol. 41, p. 102440, 2021.

[127] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: Nsga-ii," in *Parallel Problem Solving from Nature*, 2000.

[128] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, pp. 577–601, 2014.

[129] Q. Li, B. Zhang, and X. Chu, "Eapruning: Evolutionary pruning for vision transformers and cnns," in *British Machine Vision Conference*, 2022.

[130] W. Hong, G. Li, S. Liu, P. Yang, and K. Tang, "Multi-objective evolutionary optimization for hardware-aware neural network pruning," *Fundamental Research*, 2022.

[131] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE Trans. Evol. Comput.*, vol. 3, pp. 257–271, 1999.

[132] C. Audet, J. Bigeon, D. Cartier, S. Le Digabel, and L. Salomon, "Performance indicators in multiobjective optimization," *European Journal of Operational Research*, vol. 292, no. 2, pp. 397–422, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221720309620

[133] N. Riquelme, C. von Lücken, and B. Barán, "Performance metrics in multi-objective optimization," *2015 Latin American Computing Conference (CLEI)*, pp. 1–11, 2015.

[134] R. Mishra, H. P. Gupta, and T. Dutta, "A survey on deep neural network compression: Challenges, overview, and solutions," *ArXiv*, vol. abs/2010.03954, 2020.

[135] L. Yu and W. Xiang, "X-pruner: explainable pruning for vision transformers," *ArXiv*, vol. abs/2303.04935, 2023.

[136] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *CoRR*, vol. abs/1511.06530, 2015.

[137] N. M. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *ArXiv*, vol. abs/1701.06538, 2017.

[138] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv: Learning*, 2016.

[139] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2017.

[140] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.

[141] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 1398–1406, 2017.

[142] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv: Learning*, 2018.

[143] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *ArXiv*, vol. abs/1810.05270, 2018.

[144] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *International Joint Conference on Artificial Intelligence*, 2018.

[145] K. Azarian, Y. Bhalgat, J. Lee, and T. Blankevoort, "Learned threshold pruning," *ArXiv*, vol. abs/2003.00075, 2020.

[146] W. Kwon, S. Kim, M. W. Mahoney, J. Hassoun, K. Keutzer, and A. Gholami, "A fast post-training pruning framework for transformers," *ArXiv*, vol. abs/2204.09656, 2022.

[147] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," *ArXiv*, vol. abs/2301.00774, 2023.

[148] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *arXiv: Computer Vision and Pattern Recognition*, 2015.

[149] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *ArXiv*, vol. abs/1802.04680, 2018.

[150] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantized nets: A deeper understanding," in *NIPS*, 2017.

[151] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *ArXiv*, vol. abs/1710.03740, 2017.

[152] Y. Bondarenko, M. Nagel, and T. Blankevoort, "Understanding and overcoming the challenges of efficient transformer quantization," in *Conference on Empirical Methods in Natural Language Processing*, 2021.

[153] G. Yang, D. Lo, R. D. Mullins, and Y. Zhao, "Dynamic stashing quantization for efficient transformer training," *ArXiv*, vol. abs/2303.05295, 2023.

[154] J. Schmidhuber, "Deep learning: Our miraculous year 1990-1991," *ArXiv*, vol. abs/2005.05744, 2020.

[155] Y. Liu, J. Cao, B. Li, W. Hu, J.-F. Ding, and L. Li, "Cross-architecture knowledge distillation," *ArXiv*, vol. abs/2207.05273, 2022.

[156] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *CoRR*, vol. abs/1412.6550, 2014.

[157] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," *ArXiv*, vol. abs/1612.03928, 2016.

[158] L. Beyer, X. Zhai, A. Royer, L. Markeeva, R. Anil, and A. Kolesnikov, "Knowledge distillation: A good teacher is patient and consistent," *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10 915–10 924, 2021.

[159] S. Hahn and H. Choi, "Self-knowledge distillation in natural language processing," in *Recent Advances in Natural Language Processing*, 2019.

[160] X. Ren, R. Shi, and F. Li, "Distill bert to traditional models in chinese machine reading comprehension (student abstract)," in *AAAI Conference on Artificial Intelligence*, 2020.

[161] Z. Yang, C. Zhang, W. Zhang, J. Jin, and D. Chen, "Essence knowledge distillation for speech recognition," *ArXiv*, vol. abs/1906.10834, 2019.

[162] J. Yoon, H. Lee, H. Y. Kim, W. I. Cho, and N. S. Kim, "Tutornet: Towards flexible knowledge distillation for end-to-end speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 1626–1638, 2020.

[163] X. Yang, Q. Li, C. Zhang, and P. C. Woodland, "Knowledge distillation from multiple foundation models for end-to-end speech recognition," *ArXiv*, vol. abs/2303.10917, 2023.

[164] K. Wang, F. Yang, and J. van de Weijer, "Attention distillation: self-supervised vision transformer students need more guidance," in *British Machine Vision Conference*, 2022.

[165] D. Jia, K. Han, Y. Wang, Y. Tang, J. Guo, C. Zhang, and D. Tao, "Efficient vision transformers via fine-grained manifold distillation," *ArXiv*, vol. abs/2107.01378, 2021.

[166] H. Prasantha, H. Shashidhara, and K. Balasubramanya Murthy, "Image compression using svd," in *International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007)*, vol. 3, 2007, pp. 143–145.

[167] S. R. Kamalakara, A. F. Locatelli, B. Venkitesh, J. Ba, Y. Gal, and A. N. Gomez, "Exploring low rank training of deep neural networks," *ArXiv*, vol. abs/2209.13569, 2022.

[168] G. I. Winata, S. Cahyawijaya, Z. Lin, Z. Liu, and P. Fung, "Lightweight and efficient end-to-end speech recognition using low-rank transformer," *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6144–6148, 2019.

[169] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *ArXiv*, vol. abs/1909.11942, 2019.

[170] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *ArXiv*, vol. abs/2001.04451, 2020.

[171] S. Hooker, N. Moorosi, G. Clark, S. Bengio, and E. L. Denton, "Characterising bias in compressed models," *ArXiv*, vol. abs/2010.03058, 2020.

[172] M. Pietroń and M. Wielgosz, "Retrain or not retrain? - efficient pruning methods of deep cnn networks," *Computational Science – ICCS 2020*, vol. 12139, pp. 452 – 463, 2020.

[173] R. Dong, Y. Mao, and J. Zhang, "Resource-constrained edge ai with early exit prediction," *J. Commun. Inf. Networks*, vol. 7, pp. 122–134, 2022.

[174] F. Ilhan, L. Liu, K.-H. Chow, W. Wei, Y. Wu, M. Lee, R. R. Kompella, H. Latapie, and G. Liu, "Eenet: Learning to early exit for adaptive inference," *ArXiv*, vol. abs/2301.07099, 2023.

[175] M. McGill and P. Perona, "Deciding how to decide: Dynamic routing in artificial neural networks," in *International Conference on Machine Learning*, 2017.

[176] K. Wu, J. Zhang, H. Peng, M. Liu, B. Xiao, J. Fu, and L. Yuan, "Tinyvit: Fast pretraining distillation for small vision transformers," *ArXiv*, vol. abs/2207.10666, 2022.

[177] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *ArXiv*, vol. abs/2006.04768, 2020.

[178] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[179] Y. Tay, M. Dehghani, S. Abnar, H. W. Chung, W. Fedus, J. Rao, S. Narang, V. Q. Tran, D. Yogatama, and D. Metzler, "Scaling laws vs model architectures: How does inductive bias influence scaling?" *ArXiv*, vol. abs/2207.10551, 2022.

[180] G. Kim and K. Cho, "Length-adaptive transformer: Train once with length drop, use anytime with search," in *Annual Meeting of the Association for Computational Linguistics*, 2020.

[181] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, "Design science research process: A model for producing and presenting information systems research," *ArXiv*, vol. abs/2006.02763, 2020.

[182] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 9992–10 002, 2021.

[183] S. H. Lee, S. Lee, and B. C. Song, "Vision transformer for small-size datasets," *ArXiv*, vol. abs/2112.13492, 2021.

[184] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation policies from data," *ArXiv*, vol. abs/1805.09501, 2018.

[185] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, "Random erasing data augmentation," *ArXiv*, vol. abs/1708.04896, 2017.

[186] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *ArXiv*, vol. abs/1611.01578, 2017.

[187] C. F. Hayes, R. Rădulescu, E. Bargiacchi, J. Källström, M. Macfarlane, M. Reymond, T. Verstraeten, L. M. Zintgraf, R. Dazeley, F. Heintz, E. Howley, A. A. Irissappane, P. Mannion, A. Nowé, G. Ramos, M. Restelli, P. Vamplew, and D. M. Roijers, "A practical guide to multi-objective reinforcement learning and planning," *Autonomous Agents and Multi-Agent Systems*, vol. 36, no. 1, apr 2022. [Online]. Available: https://doi.org/10.1007/s10458-022-09552-y

[188] S. Russel and P. Norvig, ""artificial intelligence – a modern approach", second edition, pearson education, 2003." 2015.

[189] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *ArXiv*, vol. abs/1711.05101, 2017.

[190] X. Zhi, V. Babbar, P. W. Sun, F. Silavong, R. Shi, and S. Moran, "Lightweight parameter pruning for energy-efficient deep learning: A binarized gating module approach," 2023.

[191] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of neural network representations revisited," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3519–3529.