Symbolic Regression and Sequence Modelling with Conditional and Dynamic Language Models

by

Mojtaba Valipour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Christopher J. Pal
Professor, École Polytechnique de Montréal,
University of Montreal & Mila

Supervisor: Ali Ghodsi
Professor, Dept. of Statistics and Actuarial Science,
University of Waterloo

Internal Member: Ming Li
Professor, Dept. of Computer Science,
University of Waterloo

Internal-External Member: Mohammad Kohandel
Professor, Dept. of Applied Mathematics,
University of Waterloo

Other Member(s): Wenhu Chen
Assistant Professor, Dept. of Computer Science,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is a compilation of research efforts aimed at advancing the field of deep learning and artificial intelligence, specifically focusing on symbolic regression and dynamic training methodologies for neural networks. Each chapter is based on a peer-reviewed paper or a preprint version, as listed below, showcasing innovative contributions to the academic community.

Chapter 1: SymbolicGPT is based on:

> ”Valipour, M., You, B., Panju, M., & Ghodsi, A. (2021). Symbolicgpt: A generative transformer model for symbolic regression. arXiv preprint arXiv:2106.14131.”

This paper was also presented at the Neurips 2022 - ENLSP workshop.

Chapter 2: DyLoRA is based on:

> ”Valipour, M., Rezagholizadeh, M., Kobyzev, I., & Ghodsi, A. (2023, May). DyLoRA: Parameter-Efficient Tuning of Pre-trained Models using Dynamic Search-Free Low-Rank Adaptation. In Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics (pp. 3274-3287).”

Chapter 3: SortedNet is based on:

> ”Valipour, M., Rezagholizadeh, M., Rajabzadeh, H., Tahaei, M., Chen, B., & Ghodsi, A. (2023). Sortednet, a place for every network and every network in its place: Towards a generalized solution for training many-in-one neural networks. arXiv preprint arXiv:2309.00255.”

This paper was also presented at the Neurips 2023 ENLSP workshop.

Chapter 4: Sorted LLaMA is based on:

> ”Kavehzadeh, P., Valipour, M., Tahaei, M., Ghodsi, A., Chen, B., & Rezagholizadeh, M. (2024, March). Sorted LLaMA: Unlocking the Potential of Intermediate Layers of Large Language Models for Dynamic Inference. In Findings of the Association for Computational Linguistics: EACL 2024 (pp. 2129-2145).”

Abstract

In an era where the boundaries of machine learning are continuously being pushed, this thesis presents two more advancements in the field of deep learning and artificial intelligence, with a focus on symbolic regression and dynamic training methodologies for neural networks. The first major contribution, SymbolicGPT, introduces a novel approach to symbolic regression using a transformer-based language model. This model significantly outperforms traditional methods by leveraging the strengths of probabilistic language models for improved accuracy and efficiency. The second theme of this thesis revolves around dynamic training methodologies, aimed at enhancing the adaptability and computational efficiency of neural networks under varying constraints. Within this framework, we introduce DyLoRA and SortedNet as key innovations. DyLoRA offers a dynamic, search-free low-rank adaptation technique, enabling models to adjust their complexity on-the-fly without extensive retraining. SortedNet proposes a generalized framework for embedding multiple neural network architectures within a single model, facilitating efficient model selection and adaptation. Extending SortedNet, SortedLLama applies these principles to large language models, demonstrating efficient dynamic inference capabilities.

Acknowledgments

My heart is filled with gratitude, reaching out to each soul whose presence has woven threads of joy into the fabric of my life. Blessed beyond measure, I cherish the moments shared with you, savoring every second as a precious gift. In the company of you all, every moment was a dance, a song, a burst of laughter under the wide skies of our shared journey.

Dedication

To those I love, this is dedicated.

To my mother, who always motivated me to achieve more and make her proud, and to my father, who always supported me.

To my brothers and sisters, who taught me a lot by sharing their own life experiences.

To friends who have contributed to the enrichment of my life.

To all those who have loved me without hesitation, without expectations and treated me more like a soul mate than another ordinary person in their life, giving me a chance to grow up, learn and cherish my time with them.

To all who better understand what I mean.

To my colleagues and mentors, who accelerated my research with their constructive feedback and fruitful discussions.

And to my supervisor, who patiently tolerated my mistakes and encouraged me to become a better person.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of deep learning and neural networks has revolutionized a wide range of fields. Language models - especially deep transformers such as GPT [10] - have made significant advances in linguistics and natural language processing. These advancements have extended the range of problems that can be solved by language models, including those that appear unrelated to language.

However, many of these problems still remain challenging. For example, how can we make GPT conditional over an extra order-invariant embedding? Or how can we use these transformers in a limited budget setting when using the full capacity of the model is not viable? These questions inspired us to think deeply about how we can make large language models more flexible. Therefore, we came up with several new methods, and we demonstrated the capacity and limits of our novel proposed methods in different settings.

This thesis is a step towards better large language models that are more flexible, can solve a wider range of problems, and they can be used in practical settings. To summarize, this thesis presents two advancements in the field of deep learning. The first contribution, SymbolicGPT, introduces a novel transformer language model tailored for the task of symbolic regression. We created a new synthetic dataset with our proposed framework for scalable data generation, we demonstrated the possibility and effectiveness of pre-training models for symbolic regression and we compared our method with the traditional methods.

The second theme of this thesis focus on the efficient and dynamic language models, aiming to enhance the adaptability and computational efficiency of neural networks during inference. Within this framework, we introduce DyLoRA, SortedNet and SortedLlama. DyLoRA propose a novel dynamic, search-free low-rank adaptation technique that enable models to adjust their ranks on-the-fly without training overhead. SortedNet proposes a generalized scalable framework for training many-in-one models simultaneously. This facilitates efficient model selection and offers flexibility during inference. And finally, we extend SortedNet to Sorted Fine-Tuning (SoFT) for making pre-trained generative large language models many-in-one.

The remaining part of this chapter provides an overview of the structure of this thesis and the contributions that we had in each part.

1.1 Thesis Overview

1.1.1 Chapter 1: SymbolicGPT

One of the intriguing problems that is challenging for Large Language Models is symbolic regression, which involves finding a mathematical equation that fits a dataset. Traditional approaches, primarily based on genetic programming [3], involve searching the vast space of mathematical expressions to find an optimal formula. These methods are, however, computationally intensive since they need to be retrained for each new dataset.

In this context, we introduce a paradigm shift by treating symbolic regression as a language modeling task. It's feasible to employ deep language models for symbolic mathematics because mathematical expressions adhere to their own grammar, just as human language does. By leveraging this idea, SymbolicGPT transforms the symbolic regression problem into a captioning pre-training task. As opposed to training a model for each dataset, SymbolicGPT trains a singular model to caption datasets with their underlying equations. The contributions of this work are as follows:

- A novel scalable generative pre-training transformer model to solve the task of Symbolic Regression
- We also proposed a method for synthetic equation generation to build the required pre-training dataset

1.1.2 Chapter 2: DyLoRA

Models are growing, especially pre-trained ones, which require dynamic, efficient, and adaptable training necessary. While fine-tuning is effective, it often involves significant computational cost. To address this, we propose DyLoRA, a dynamic low-rank adapter technique that overcomes the limitations of fixed-size LoRA blocks. This method allows dynamic, search-free models that are faster and more adaptive than their traditional counterparts. Our contributions can be summarized as:

- We proposed a novel algorithm that makes LoRA dynamic at inference time without adding extra overhead.
- We demonstrated that our proposed method is a search-free LoRA that make the user free of searching for the optimal rank.

1.1.3 Chapter 3: SortedNet

Moreover, the increasing size of deep neural networks necessitates models that can adapt to dynamic conditions, especially given varying computational resources and application requirements. While deep neural networks inherently possess modularity, current training methods aren't fully exploiting it. Our solution, inspired by previous works, introduces nested models with sorted training for dynamic inference. This approach harnesses the modularity of neural networks, allowing for efficient dynamic inference across various architectures without modifying the original model. Contributions of this chapter are:

- We introduced a novel many-in-one solution to configure sub-models in a sorted-manner and training them all at once
- Our proposed method outperforms state-of-the-art methods to train many-in-one neural networks
- We also demonstrated the performance of this approach in pre-trained language models.

1.1.4 Chapter 4: SortedLLama

Lastly, the rise of large language models has transformed the NLP landscape. While these models are powerful, their deployment can be resource-intensive. We extend the SortedNet approach to generative NLP tasks, enabling dynamic inference of large language models without pre-training. By focusing on the potential of intermediate layers, our approach offers a more efficient and flexible solution for deploying these models in real-world scenarios. Our contributions in this work are as follows:

- We extended SortedNet training mechanism for the Meta LLama [99] and shows the effectiveness of this approach
- We proposed novel analysis scheme to understand the limitations and opportunities
- We conducted extensive experiments to evaluate the performance of sub-models in the generative tasks

1.1.5 Appendices

In this part, we briefly mention what other topics have been covered in the appendix. Appendix A covers the details of a dropout [92] inspired method to make the SymbolicGPT conditional, a novel model-agnostics conditional generative model for symbolic regression, and more experiments. Appendix B covers time complexity analysis, more details regarding the hyperparameters of the experimental settings that we chose, and more experiments in the setting of DyLoRA for GPT [82]. Appendix C covers the difference between sorted and nested architectures, an extensive literature review, a theoretical analysis for the performance bound of SortedNet models, and more experiment details including hyperparameters, the effect of gradient accumulation on SortedNet performance, answer whether we can extend SortedNet mechanism to more complex dimensions and a synthetic experiment to understand the impact of sorting information better. And finally appendix D discusses the computation overhead of Sorted Fine-Tuning and provides additional experiments and analysis and details about the experimental settings.

To summarize, this introduction gives an overview of the innovative methods we propose for this thesis. These methods aim to improve the current transformers and make better conditional and dynamic models, optimizing large language model deployments.

Chapter 2

SymbolicGPT: A Generative Transformer Model for Symbolic Regression

2.1 Abstract

Symbolic regression is the task of identifying a mathematical expression that best fits a provided dataset of input and output values. Due to the richness of the space of mathematical expressions, symbolic regression is generally a challenging problem. While conventional approaches based on genetic evolution algorithms have been used for decades, deep learning-based methods are relatively new and an active research area. In this work, we present SymbolicGPT, a novel transformer-based language model for symbolic regression[1]. This model exploits the advantages of probabilistic language models like GPT, including strength in performance and flexibility. Through comprehensive experiments, we show that our model performs strongly compared to competing models with respect to the accuracy, running time, and data efficiency.

2.2 Introduction

Deep learning and neural networks have earned an esteemed reputation for being capable tools for solving a wide variety of problems over countless application domains. Notably, deep language models have made an enormous impact in the field of linguistics and natural language processing. With the advances in technology like Generative Pre-trained Transformers, or GPT

---

[1] Code, datasets and results are available at github.com/mojivalipour/symbolicgpt

[81], the scope of problems now accessible to automated methods continues to grow. It is particularly interesting when language models are used for tasks that, at first glance, do not seem to have any relationship with language at all.

Symbolic regression, the problem of finding a mathematical equation to fit a set of data, is one such task. The objective of symbolic regression is to obtain a closed-form symbolic mathematical expression to describe the relationship between specified predictor and response variables, where the mathematical expression is allowed to be flexible without being restricted to a particular structure or family. More precisely, the goal in symbolic regression is to recover a mathematical function f in terms of the input variables $\boldsymbol{x} = [x_1 \dots x_d]^\top$, given a set of data point vectors of the form $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$, such that $f(\boldsymbol{x}_i) = y_i$ for all i. Here, $x_1, \dots, x_d, y_i$ are scalars and $\boldsymbol{x}_i \in \mathbb{R}^d$.

By not imposing any structural constraints on the shape of the desired equation, symbolic regression is a much more difficult task compared to other kinds of regression, such as linear regression or multinomial regression, as the search space of candidate expressions is so much larger.

The most common approach for symbolic regression is based on genetic programming, where numerous candidate parse trees are generated, evaluated, combined, and mutated in an evolutionary way until a tree is produced that models an expression that fits the dataset up to a required accuracy level. In essence, it is a search strategy over the vast space of mathematical expressions, seeking the formula that would optimize an objective function.

In this typical framework, which applies not only to genetic methods but also many deep-learning-based approaches to symbolic regression, the goal is to identify a mathematical expression that most optimally fits a single input dataset. This dataset is the basis over which all the training occurs. Consequently, when presented with any new dataset (as a fresh instance of the task of symbolic regression), the entire training procedure must begin again from scratch.

In this work, we explore an alternative approach to symbolic regression by considering it as a task in language modeling. Symbolic mathematics behaves as a language in its own, with well-formed mathematical expressions treated as valid “sentences” in this language. As with human language, symbolic equations follow their own grammar, and an equation cannot be generated at random. It is natural, therefore, to consider using deep language models to address tasks involving symbolic mathematics.

We can frame the regression problem as an exercise in captioning. Each instance takes input in the form of a cloud of points in $\mathbb{R}^{d+1}$, with each point consisting of d components corresponding to $\boldsymbol{x}$ and a single component for the associated y value. The instance returns a statement in the language of symbolic mathematics to describe the point set. By training a model to correctly "caption" datasets with the equations underlying them, we obtain a system for performing symbolic regression quickly and accurately.

Based on this idea, we present SymbolicGPT, a method that makes use of deep language models for symbolic regression. SymbolicGPT employs a framework that represents a major shift from the way symbolic regression is conventionally performed. We move the task of symbolic regression from being a strictly quantitative problem into a language one. Effectively, we propose a system that not only learns the language of symbolic mathematics, but also the underlying relationship between point clouds and mathematical expressions that define them.

As part of SymbolicGPT, we can use a network similar to a point-net model [80] to represent the input point cloud in an order-invariant way. This allows us to obtain vector embeddings of the entire input dataset for symbolic regression instances without depending on the number of points in the dataset or the order in which they are given.

A major advantage of SymbolicGPT is that we are no longer training a model to learn an equation for an individual dataset in each instance of symbolic regression. Instead, we train a single language model once, and use that trained model to rapidly solve instances of symbolic regression as individual captioning tasks. We will show that SymbolicGPT provides competent performance in accurately reconstructing mathematical equations to fit numerical datasets, presenting a new frontier for language models and a novel direction for approaching symbolic regression.

2.3 Related Work

Traditionally, the problem of symbolic regression has been tackled with methods based on genetic algorithms [65, 3, 90, 66, 111]. In this framework, the task is seen as a search optimization problem where symbolic expressions are candidates and the expression with the greatest fitness, or fitting accuracy on the training data, is obtained through a process of mutation and evolution. Although this approach has shown success in practice, it is computationally expensive, highly randomized, requires instance-based training, and struggles with learning equations containing many variables and constants.

More recently, newer approaches to symbolic regression have arisen that make use of neural networks. The EQL (Equation Learner) model [64, 86] is an example of performing symbolic regression by training a neural network that represents a symbolic expression. This method, and others based on it [14, 50], take advantage of advances in deep learning as an alternative to genetic approaches. However, they still approach symbolic regression as an instance-based problem, training a model from scratch given every new input dataset for a regression task.

A recent study [5] presents a novel, language-based method for handling symbolic regression as a machine translation task, similar to the approach used by [54] for performing symbolic integration and solving differential equations. Given an input dataset, the algorithm treats the input as a text string and passes it through a trained sequence-to-sequence LSTM to produce an output text string that is parsed as the desired symbolic expression. Although this method overcomes the cost of per-instance training, its interpretation of the input dataset as a textual string limits its usability, as the input data must follow specific constraints, such as fitting a one-dimensional mesh of fixed size. Consequently, this method can only be used in one-dimensional space. However, in most problems, more than one variable is involved and we need to find a multivariate function.

A recent extension of this work [6] uses the set transformer encoder [55] instead of LSTMs to resolve the scalability issues and address the permutation invariancy of the input points properly. In this work, we also propose a scalable method that removes such limitations on the structure of input data. This can be applied easily to symbolic regression problems in high-dimensional spaces and when many variables are involved.

Another active area of research is to use deep reinforcement learning methods to tackle this problem [50, 76]. The method presented by Petersen et. al. [76] uses a hybrid approach between traditional genetic algorithms and deep learning methods. Here, the authors use deep RNNs to generate samples of candidate skeletons. As an example, if the function was $f(x) = x^2 + 1$, the corresponding skeleton would be $C_1 x^2 + C_2$. As in [52], numerical optimization is then used to optimize for the constants of each candidate skeleton. A reinforcement learning algorithm is applied to train the RNN to generate better skeletons at every iteration. However, this method still relies on the iterative nature of traditional genetic algorithms as well as numerical optimization. This results in a computationally intensive process to generate a prediction for each equation.

2.4 Method

Our model for symbolic regression, SymbolicGPT, consists of three main stages: obtaining an order-invariant embedding of the input dataset using our T-net network [80], obtaining a skeleton equation using a GPT language model [82], and optimizing constant values to fill in the equation skeleton. In addition to discussing each of these steps, we also present the method for generating our equation datasets.

2.4.1 Equation Generation

To train our language model, we need a large dataset of solved instances of symbolic regression. This dataset is a collection of input-label pairs where each input is in the form of a numerical dataset, itself a set of input and output pairs $\{(\boldsymbol{x}, y)\}$, and the corresponding label is a string encoding the symbolic expression governing the relationship between variables in the numerical dataset.

To ensure that the language model is able to generalize to unseen equations, having good training data is key. It is necessary to train the model over a wide, diverse set of training equations to prevent the language model from overfitting.

There are a number of different ways to randomly sample symbolic mathematical expressions. One approach, as used in [8], is to consider symbolic expressions as constructed by rules in a context-free grammar, and randomly sampling from rules until reaching a string containing only terminal values. Another approach, taken in [54], uses parse-tree representations of symbolic formulas, presenting a method that samples uniformly from all trees of n nodes and then filling in nodes with valid operators or variable values.

For our training dataset, we use an approach similar to the latter, where we start with a blank parse tree and then "decorate" the nodes with choices of operators and variables. In contrast with [54], we do not constrain our parse trees by the number of nodes, but by the number of levels. This enables more control over the maximum level of complexity in the equations used in our training set, as the number of levels in the parse tree corresponds to the number of potential function nesting, a measure of how complex an equation can be.

We begin by fixing k, the maximum number of levels in the parse tree for the equations we wish to encounter in our training set. We also begin with a pre-specified number of variables, d, and a pre-selected set of operators, $P = \{u_1, \ldots, u_m\}$, that are allowed to appear in any training equation. Then, for each data-equation pair in our training set, we generate a perfectly balanced binary tree of depth k, having $2^{k-1}-1$ internal nodes and 2^{k-1} leaf nodes. These nodes originally start off empty to form the template of a symbolic expression.

The template is filled in by randomly selecting valid choices to occupy each node in the parse tree. For leaf nodes, each node is randomly assigned with a variable from the set $\{x_1, \ldots, x_d\}$. For interior nodes, operators from the set P are randomly chosen. Once filled in, the parse tree can naturally be interpreted as a symbolic expression. For nodes filled in by binary operators, both of their child nodes are used as input; in the case of unary operators, only the left child is used as input, and the right child is ignored. Importantly, the unary operator "id($\cdot$)", which returns its input argument unchanged, is included in P, which effectively allows for equations with shallower or unbalanced parse trees to still be represented using this template.

Additionally, to ensure that the equations generated are not all too complex, we introduce "terminal" nodes in which children of the terminal nodes are discarded. This ensures that we obtain a diverse set of equations within the training set.

As a final step for the equation generation procedure, constants are incorporated into the equation by inserting them at nodes in the parse tree. Given a specified value $r \in [0, 1]$ and constant bounds c_{min} and c_{max}, for each node in the tree, a random real-valued constant is selected between c_{min} and c_{max} and, with probability r, is inserted as a multiplicative factor the subtree rooted at that node. Similarly, a second random constant is selected between c_{min} and c_{max} and, with probability r, is inserted as an additive bias to the subtree rooted at that node. By varying the constant ratio r, the equations can be customized to include many constants, few constants, or none at all.

Once an equation is generated, an input dataset for symbolic regression can be produced by evaluating the symbolic expression at n different vectors $\boldsymbol{x}$ randomly sampled from some region of interest in $\mathbb{R}^d$. The label value for the symbolic regression instance would be the symbolic expression. This process can be repeated many times to construct the training set by which our SymbolicGPT model will learn how to perform symbolic regression.

2.4.2 Order-Invariant Embedding

Once the training set of input data and output equations is generated, it is used to train our model for translating numerical datasets into equation strings.

The first step in our system is to convert the input dataset $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n} \subset \mathbb{R}^{d+1}$ into a single vector embedding $\boldsymbol{w}_D \in \mathbb{R}^e$. For the conversion to be useful, it must have two properties. First, it should not strictly depend on the number of points in the input dataset, n. In practice, the datasets provided as input to a symbolic regression solver may have varying sizes, and we do not want our method to be restricted to cases with a fixed number of input points.

Second, the conversion method should not be sensitive to the order in which the points of the dataset are given. The input to a symbolic regression instance is a collection of data points, rather than a sequence, and the optimal symbolic expression to fit the dataset should not depend on the order in which the points are listed. Thus, the vector embedding of the dataset should be similarly order-invariant.

Our approach for converting the dataset D into a vector embedding is to use a network similar to T-net, a kind of deep network that makes use of a global max-pooling layer to provide order-invariance over its arbitrarily-sized input [80]. Our T-net takes as input the dataset D, consisting of n data points over d variables, represented in matrix format as $X \in \mathbb{R}^{n \times (d+1)}$, where n can be any number and d, the number of allowable variables, is fixed in advance. Any symbolic regression instance with fewer than d variables can be padded with 0 values, bringing the total number of variables up to d.

The matrix X is first normalized using a learnable normalization layer to regulate extreme values from the input. The normalized input points are then passed through three stages of MLP networks. Within each stage, each of the n rows of X are passed individually, albeit in parallel, through a single fully connected layer, where weights are shared between the networks for all points for that stage. The first stage results in n points encoded in e-dimensional space; the second stage takes them into $2e$ dimensions, and the output after the third stage are n points having $4e$ dimensions each.

The next layer in the T-net is a global max pool, which reduces the $n \times 4e$ output of the previous step down to a $1 \times 4e$-dimensional vector. The max-pooling eliminates the dependence on both n and the order of the input points, achieving both goals needed for our vector embedding. Finally, the output of the global max-pool is passed through two more fully connected layers, resulting in a single output vector $\boldsymbol{w}_D$, an e-dimensional embedding of the input dataset. The overall structure of the T-net is shown in the left part of Figure 2.1.

2.4.3 Generative Model Architecture

The main component of SymbolicGPT is the deep network for producing symbolic equations, as implemented using a GPT-based language model [81, 82, 10]. This framework takes in two pieces of input: the order-invariant embedding of the point cloud $\boldsymbol{w}_D$ as produced by the T-net, representing the input dataset, and a sequence of tokens, $X_{(eq)}$, used to initialize the output formula string. In the typical regression case where no information is provided about the output symbolic expression in advance, this token sequence would be the singleton Start-of-Sequence token $\langle SOS \rangle$, although in general it can be any desired prefix of the output equation. The input token sequence is tokenized at a character level and encoded as the matrix W_t using a trainable embedding as part of the GPT model.

The first step in the GPT model is to combine the two inputs $\boldsymbol{w}_D$ and W_t together, along with the positional embedding matrix W_p. Based on empirical support, we chose to obtain the combined embedding by taking the sum $W_p + W_D + X_{eq} W_t$, where W_D is the dataset representation vector $\boldsymbol{w}_D$ expanded to fit a matrix matching the dimensions of the other embeddings.

The combined vector is then passed through $l = 8$ successive transformer blocks, using the standard format of GPT models [82]. Each transformer block is a sequential combination of a masked multi-head self-attention layer and a pointwise feed-forward network, with all blocks feeding into a central residual pathway, similar to ResNets [32].

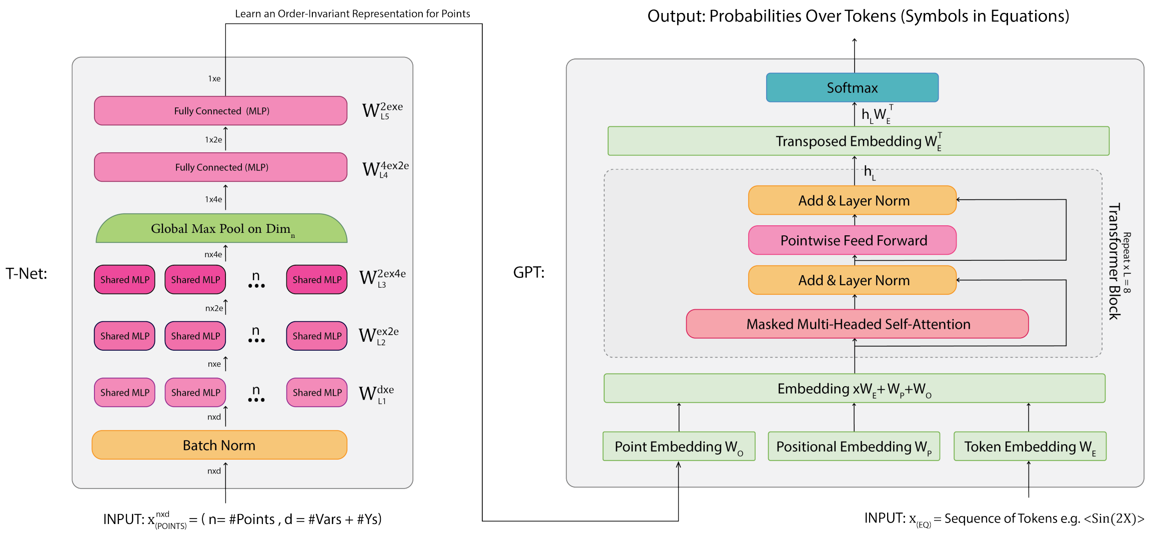


Figure 2.1: The architecture of SymbolicGPT. The left box illustrates the structure of our order-invariant T-net for obtaining a vector representation of the input dataset, and the right box shows the structure of the GPT language model for producing symbolic equation skeletons.

After l layers of the transformer block, the resulting output vector $\boldsymbol{h}$ is passed through a final decoder in the form of a linear projection into a vanilla softmax classifier. The projection uses the transposed token embedding matrix $W_t^\top$ to map the hidden state vector back into the space of tokens for symbolic expressions. The result of the softmax is a probability vector over tokens in the symbolic equation, which can be sampled to produce the best equation to describe the input dataset. We use top-k sampling with $k = 40$ for our experiments.

Although the symbolic equation used to generate the data can contain constant values, we do not train the GPT model to recover these values exactly. Instead, constant values in the equation are masked by $\langle C \rangle$ tokens during the training phase, and the output of the GPT model is a “skeleton equation” which leaves these placeholders in the output string. This is because it is unnecessary to burden the language model with the additional task of learning precise constant values, as this can be easily handled as a separate step.

To train our model, we used the standard unsupervised language model’s cost function as mentioned in the original GPT paper [81].

2.4.4 Learning Constants

Once the trained GPT model predicts a skeleton equation, we learn values of constants to decorate the skeleton as a post-processing step. This division of tasks is a common approach for string-based regression methods [52, 5].

To learn the values of constants in the symbolic equation, we employ BFGS optimization [28] using an implementation from SciPy [107]. The learned constant values then replace the $\langle C \rangle$ placeholder tokens in the skeleton equation, resulting in the final symbolic expression to represent the given symbolic regression task.

2.4.5 Evaluation Metric

In the literature, it is a common practice to use mean squared error as a metric for measuring the predictive accuracy of an equation. For data following equations with large values, however, this can be problematic, as the residuals can grow very large even when the predicted equation is very close to the true underlying one. To resolve this issue, we normalize the residuals using a factor of $\|\boldsymbol{y} + \epsilon\|_2$, where ϵ is used to avoid division by zero and $\|\cdot\|_2$ is the Euclidean norm. Then the normalized mean squared error, MSE_N, is given by

$$MSE_N(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{i=1}^{n} \frac{(y_i - \hat{y}_i)^2}{\|\boldsymbol{y} + \epsilon\|_2}$$

2.4.6 Strengths and Advantages

Our method exhibits the following strengths and advantages.

One-Time Training

In contrast with most approaches for symbolic regression, our method does not start training from scratch given every new problem instance. All of the model training is performed as a one-time procedure that takes place before the GPT transformer is ever used. Thus, SymbolicGPT enjoys all of the benefits of allowing a pretrained model, similar to popular frameworks like

BERT [20], which can make use of massive neural networks because the model can be trained offline in advance.

After the model is trained, every instance of symbolic regression can be solved rapidly as a problem in inference.

Scalability

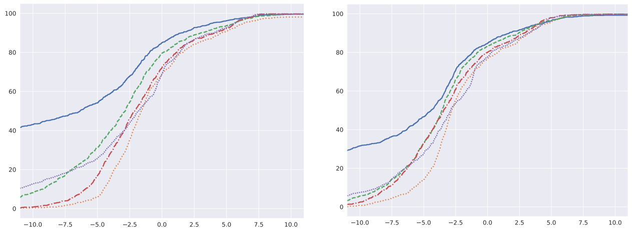
Our approach addresses two of the main problems with traditional methods. First, our model is able to scale to multiple variables. Iterative methods that choose the best candidate equation at each iteration struggle as the dimension d of the inputs increase since the search space of functions grow exponentially with respect to d. Traditional methods that rely on improving the model in an iterative fashion will struggle as the search space of functions grows exponentially with respect to d. By passing in the data points directly as inputs, the model is able to infer the dimension and produce equations accordingly. Second, our model is able to generate equation predictions rapidly compared to existing methods that train from scratch for each regression instance. We show empirically that SymbolicGPT produces superior results using significantly less computation time in the next section.

2.5 Experiments and Results

To test our model, we implemented SymbolicGPT and trained it in a number of different settings, which we detail below. In all cases, we trained SymbolicGPT over 4 epochs using a batch size of 128. The embedding size for the T-net vector representation is $e = 512$, and the maximum equation output length was capped at 200 tokens.

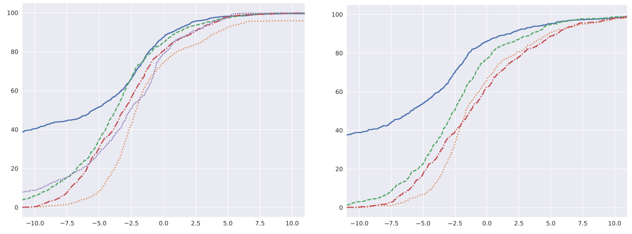
Training and inference for the SymbolicGPT model were performed using an Intel(R) Core (TM) i9-9900K CPU @ 3.60GHz with a single NVIDIA GeForce RTX 2080 11 GB GPU and 32.0 GB Ram. It is noteworthy that our performance scores were achieved using only a single GPU, and scaling up is expected to improve training and inference times even further.

Our experimental framework consists of a large-scale comparison test where we test our model on 1000 different, randomly generated instances of symbolic regression and evaluate performance based on MSE_N. We repeat this test on five different settings, based on the choice of the dimension d: datasets with one input variable, two variables, three variables, and in two different test sets, a random selection between one to nine variables. This last pair of tests will be referred to as the “general” experiments. The first general experiment, along with the three



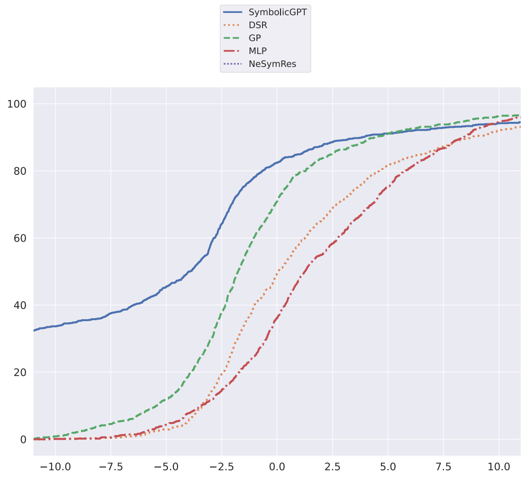
(a) 1 variable

(c) 3 variables



(b) 2 variables

(d) 1-9 vars (General)

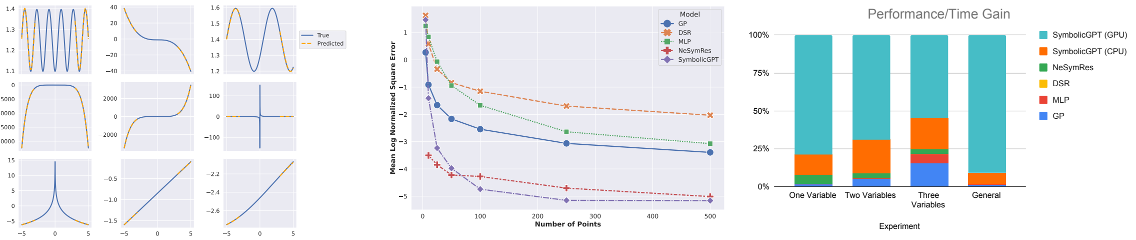


(e) 1 to 9 variables: This experiment includes all of the benchmark equations in both AI Feynman [101] and Nguyen benchmarks [103]. We used these benchmarks because it is a common practice in the literature.

Figure 2.2: Cumulative $\log MSE_N$ over all methods and experiments. Each curve shows the proportion of test cases that attained an error score less than every given threshold. SymbolicGPT finds better fitting equations for more test cases than any other method tested, as well as more highly accurate equations (with $\log MSE_N < -10$).

experiments before it, use equations that come from the same distribution that we used to generate the training set. To ensure that the test set aligns with the training set in terms of constants and data points, and to evaluate the model on real equations, we generated skeleton equations for the second "general" experiment from the 12 Nguyen template equations that involve up to 3 variables and the 98 AI Feynman equations that include equations of up to 9 variables.

In each experimental setting, SymbolicGPT was trained using a minimum of 10,000 randomly generated symbolic regression instances belonging to the associated dimensional configuration, each consisting of an input dataset and an equation label. A further 1000 dataset-equation pairs were generated as the validation set, and 1000 new dataset-equation pairs were generated for the test set using different seeds. The training and validation datasets used values of $\boldsymbol{x} \in [-3.0, 3.0]^d$, and test datasets took values of $\boldsymbol{x} \in ([-5, -3] \cup [3, 5])^d$. The datasets of



(a) (b) (c)

Figure 2.3: a) Graphical representations of selected equations of one input variable. The solid blue curves are the graphs of the true underlying equations; the orange dotted curves are the predicted functions as generated by SymbolicGPT. b) The effect of the number of data points on model performance for the three variable experiment. c) Average relative performance gain over running times (in seconds) for an instance of symbolic regression during each of the four experiments, the more the better. All measurements were conducted on the same CPU. We also reported the performance of our method on a single GPU.

True equation	Predicted by SymbolicGPT equation	Test error
$-1.67 \times 1/2 \times x_1 \times x_2^2 + 0.81$	$C \times x_1 \times x_2^2/2 + C$	4.01e-13
$-0.39 \times x_1 \times (-2.04 \times x_2/(2 \times \pi) + 1.75) - 1.73$	$C \times x_1 \times (C \times x_2/(2 \times \pi) + C) + C$	1.53e-12
$1.82 \times \sqrt{0.71 \times x_1 + 0.72} - 1.26$	$C \times \sqrt{C \times \lvert C \times x_1 + C\rvert} + C$	5.07e-12
$0.68 \times x_1^3 - 0.64 \times x_1^2 - 1.53 \times x_1 + 0.78$	$C \times x_1^3 + C \times x_1^2 + C \times x_1 + C$	8.251e-11
$\frac{-0.84 \times x_3 \times x_4 \times x_5}{(0.32 \times x_2 - 0.91 \times x_1 + 0.62)^2 + (0.75 \times x_6 + 0.35 \times x_7 + 1.10)^2 + (-0.64 \times x_8 + 0.67 \times x_9)^2 - 0.07} - 0.34$	$\frac{C \times x_3 \times x_4 \times x_5}{(C \times x_2 + C \times x_1 + C)^2 + (C \times x_6 + C \times x_7 + C)^2 + (C \times x_8 + C \times x_9)^2 + C} + C$	3.52e-10
$2.07 \times x_1^5 - 1.77 \times x_1^4 - 0.15 \times x_1^3 + 1.53 \times x1^2 + 0.94 \times x1 - 1.79$	$C \times x_1^2 + C \times x_1^5 + C \times x_1^4 + C \times x_1^3 + C \times x_1^2 + C \times x_1 + C$	9.80e-08
$0.05 \times x_1 \times (0.04 \times x_2/(2 \times \pi) + 0.05) + 1.44$	$C \times \sqrt{\times abs(C \times \cos(C \times \cos(C \times x_2)))} + C$	9.47e-06
$1.42 \times \sin(1.05 \times x_1 - 0.20) + 0.32 \times \sin(-1.11 \times x_1 + 0.80 \times x_1^2) - 0.51$	$C \times \sin(C \times x_1 + C) + C \times \sin(C \times x_1 + C \times x_1^2) + C$	0.01
$-0.79 \times \sin(0.43 \times x_1 - 0.45) \times \cos(0.35 \times x_2 + 0.81) + 0.67$	$C \times x_1 + C \times \cos(C \times x_2 + C) + C$	0.05
$1.14 \times \log(0.63 \times x_1 - 1.01) + 2.09 \times \log(-0.64 \times x_1^2 + 0.28) + 0.59$	$C \times \log(C \times x_1 + C) + C \times \log(C \times x_1^2 + C) + C$	0.06
$1.11 \times \frac{\exp(0.37 \times x_1^2 + 1.25)}{\sqrt{1.29 \times x_2 + 0.58}} + 1.67$	$C \times \sqrt{C \times abs(C \times \exp(C \times x_1)/x_2)} + C$	251.47

Table 2.1: A selection of examples showing the general model's performance on benchmark equations. Both successful and unsuccessful examples were provided to demonstrate the overall performance of the proposed method.

one, two, and three variables contained 30, 200, and 500 points, respectively. The number of points for each instance in the general dataset with $d \in \{1, 2, ..., 9\}$ was a randomly selected integer between 20 and 250.

The parse tree templates, as described in the 'Equation Generation' step, contained a maximum depth of $k = 4$ levels and allowable operators coming from the set

$$\begin{aligned} P = &\{\text{id}(\cdot), \text{add}(\cdot,\cdot), \text{mul}(\cdot,\cdot), \sin(\cdot), \text{pow}(\cdot,\cdot), \\ &\cos(\cdot), \text{sqrt}(\cdot), \text{exp}(\cdot), \text{div}(\cdot,\cdot), \text{sub}(\cdot,\cdot), \\ &\log(\cdot), \arcsin(\cdot)\} \end{aligned}$$

Constant values selected from the interval $[-2.1, 2.1]$ were randomly inserted using a constant ratio $r = 0.5$.

We compared our methods with four existing models for nonlinear regression:

1. Neural Symbolic Regression that Scales (**NeSymReS**): We selected [6] to represent one of the most recent developments in deep learning methods for symbolic regression. We trained NeSymRes on our generated dataset and at inference time generate one candidate equation for BFGS constant optimization. Additionally, we use the architecture configurations provided in their official implementation.
2. Deep Symbolic Regression (**DSR**): DSR [76] is a reinforcement learning based approach to symbolic regression. It is effective for simple equations but suffers scalability issues due to a constant optimization step that is computationally expensive for larger configurations.
3. Genetic Programming (**GP**): We chose Python's GPLearn package to represent genetic evolution algorithms for symbolic regression, using a model with a population size of 1000 and 10 generations.
4. Neural Network (**MLP**): We use a standard Multilayer Perceptron to act as a non-symbolic, nonlinear regressor to use as a baseline for comparison, as implemented in the Python package Scikit-Learn [74].

For each method, we evaluated its performance on 1000 test instances of symbolic regression in each of the four experiment settings, using MSE_N as the fitness metric. We summarized the results in the cumulative distribution plots of Figure A.3, showing the proportion of the test cases that attained error less than any given threshold value. Methods corresponding to curves positioned higher in the plot achieved higher accuracy on more test equations, and hence are better regressors. However, the most important region of the plot is the far left side, as the number of test cases that achieved the lowest possible error is an indication of how often the method would find a highly accurate fitting equation. As in the NeSymRes paper [6], we report NeSymRes for up to three variables.

Some visualized examples of predictions generated by SymbolicGPT are presented in Figure 2.3b. Table 2.1 also shows predicted equations generated by the general model for the benchmark datasets. Despite the fact that the model recovered the correct template for some datasets, BFGS was unable to optimize the constants across all datasets. It is also evident from the examples that the model is not always successful in recovering every possible combination of constants or variables in an equation template.

The average gain (performance/time), are shown in figure 2.3c. To make a fair comparison between running times, all experiments were performed using the same computer specifications. The results show that SymbolicGPT significantly provide better performance gain to solve an instance of symbolic regression compared with other methods.

To gauge the data efficiency of each methods, we also ran each algorithms on symbolic regression instances with the varying number of input data points. The results of this experiment are shown in Figure 2.3a. As expected, more training points results in better performance across all methods ; however, SymbolicGPT achieves lower error comparing to all other methods.

2.6 Conclusion

In this work, we have presented a method that pushes the boundaries of language models and approaches the problem of symbolic regression from a new and powerful direction. We have employed language models in a novel way and with a novel approach, combining them with symbolic mathematics and order-invariant representations of point clouds. Our approach eliminates the per-instance computation expense of most regression methods, and resolves the input restrictions imposed by other language-based regression models. Moreover, our method is fast, scalable, and performs competently on several kinds of symbolic regression problems when compared with existing approaches.

In future work, we would like to explore the limits of SymbolicGPT to measure the extent of its scalability. Our experiments showed the competency of our model, but did not establish the capacity of the SymbolicGPT framework. Although we only used one GPU in all of our experiments, it is possible to train powerful language models for symbolic regression with more computational power, larger datasets, and more sophisticated models like GPT3 [10].

There is also room to improve in situations where exact equation recovery is required, a problem whose current bottleneck is the constant optimization step in our algorithm. Another area to explore further is the effect of different sampling strategies for the learned equations from the generative language model.

2.7 Limitations

To process the additional input, we must modify the model's architecture accordingly. Adding new nodes to the input layer of the model would cover this newly added input, for example. A drawback of this approach is that it cannot be used to manage different "$\mathbf{c}$" with different dimensions. This means we have to use padding, which is inefficient in large dimensions. Additionally, to ensure both inputs ("$\mathbf{c}$" and "$\mathbf{x}$") contribute equally, we need to map them to a common space. Otherwise, the model will be biased toward the bigger input. Different problems require different dimensions for the additional input, and it is unclear what is the most optimal choice. In addition, we will limit the effect of the condition in the input layer, so it is the model's responsibility not to forget and to learn what the condition's effect is, making learning even harder.

Chapter 3

DyLoRA: Parameter Efficient Tuning of Pre-trained Models using Dynamic Search-Free Low Rank Adaptation

3.1 Abstract

With the ever-growing size of pretrained models (PMs), fine-tuning them has become more expensive and resource-hungry. As a remedy, low-rank adapters (LoRA) keep the main pretrained weights of the model frozen and just introduce some learnable truncated SVD modules (so-called LoRA blocks) to the model. While LoRA blocks are parameter-efficient, they suffer from two major problems: first, the size of these blocks is fixed and cannot be modified after training (for example, if we need to change the rank of LoRA blocks, then we need to re-train them from scratch); second, optimizing their rank requires an exhaustive search and effort. In this work, we introduce a dynamic low-rank adaptation (DyLoRA) technique to address these two problems together. Our DyLoRA method trains LoRA blocks for a range of ranks instead of a single rank by sorting the representation learned by the adapter module at different ranks during training. We evaluate our solution on different natural language understanding (GLUE benchmark) and language generation tasks (E2E, DART and WebNLG) using different pretrained models such as RoBERTa and GPT with different sizes. Our results show that we can train dynamic search-free models with DyLoRA at least 4 to 7 times (depending to the task) faster than LoRA without significantly compromising performance. Moreover, our models can perform consistently well on a much larger range of ranks compared to LoRA. [1]

---

[1] github.com/huawei-noah/KD-NLP/tree/main/DyLoRA

3.2 Introduction

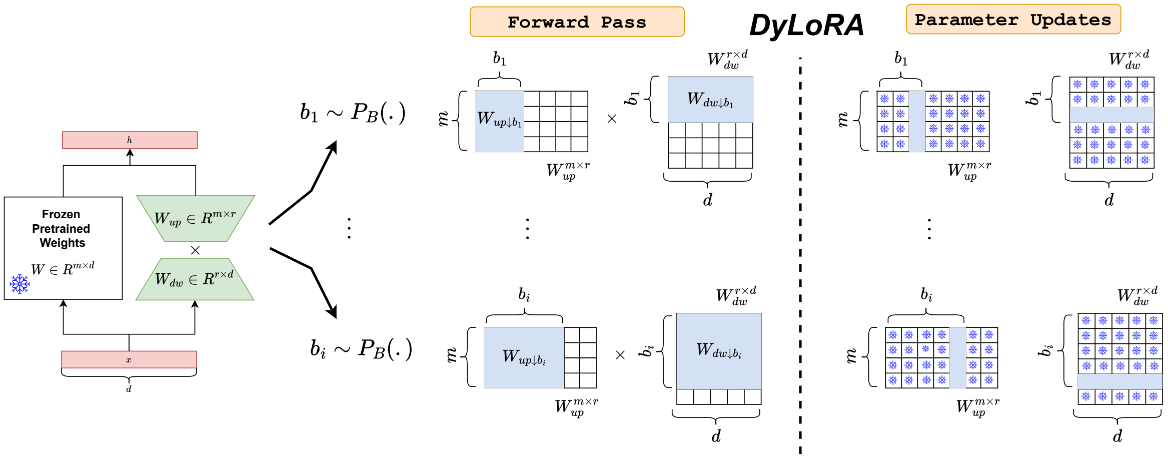


Figure 3.1: DyLoRA: The overall diagram of our proposed method. In each iteration, we sample from a pre-defined random distribution which will help us to truncate the up-projection and down-projection matrices in the LoRA [42] objective.

Pre-training/fine-tuning has become a popular paradigm for solving many tasks in natural language processing (NLP) [20, 61, 9] and Computer Vision [91, 32, 40, 7, 15, 23]. pretrained models (PMs) such as pretrained language models (PLMs) [20, 9], and pretrained visual-language models [62, 60, 93, 116] have advanced a lot in recent years. With the ever-growing size of these pretrained models, fine-tuning them on downstream tasks becomes more expensive. Moreover, as the ratio of the number of parameters of models with respect to the labeled data increases, the fine-tuning process will be more prone to overfitting [48]. There are two categories of solutions: first, model compression [45, 16]; second, parameter-efficient tuning (PET) [38, 48, 63].

There are many different model compression techniques in the literature for Transformer-based models such as matrix factorization [68, 96], pruning [113], quantization [97, 79], and knowledge distillation [33, 59, 45, 73, 83]. There are also different types of PET techniques in the literature such as low-rank adapters [109, 48, 39, 43], and prompt-based techniques [56].

Although model compression solutions are well-established in recent years in the literature, applying them to large language models can be very costly, because compression techniques usually need to train (or fine-tune) the original large model. A case in point is knowledge distillation which relies on fine-tuning a large teacher model or even pre-training the student model as suggested in [46]. Moreover, using compression techniques usually leads to degrading the model performance. PETs can be alternatives to the compression methods, especially when we would like to use the full capacity of the large pretrained models with light training efforts (such as the *language-model-as-a-service* scenario [94]). Among PET techniques, low-rank adapters have received much attention because, in contrast to prompt-tuning techniques, low-rank adapters do not add to the sequence length, get trained faster, and perform better [48]. Even though there are several low-rank adaptation techniques in the literature, such as Adapter [39], Compacter [48], and LoRA [43]; they all suffer from two major common problems: first, it is not clear how to select the size of their rank (while their performance is very sensitive to this rank selection); second, their training is static which means that if a low-rank model is trained based on a particular rank size, it will not work well in other rank values (i.e. for any other rank value we need to train a separate model).

This paper proposes a dynamic low-rank adapter technique (DyLoRA) to address these two problems. Without loss of generality, we focus on LoRA[42] and train LoRA blocks for a range of ranks instead of a single rank by sorting out the representation learned at different ranks during training. While our model is more flexible, it can outperform LoRA in a much wider range of ranks without adding to the training time. Moreover, our technique does not need extra training for searching across ranks. We summarize our contributions in the following:

- Dynamic LoRA: On top of LoRA, we developed a new algorithm (DyLoRA) that makes it dynamic at inference time without incurring extra costs.
- Search-free LoRA: We demonstrate that by making a negligible compromise in performance, it is possible to avoid the costly search process of choosing the optimal rank for LoRA.

3.3 Related Work

This section reviews low-rank adaptation techniques for parameter-efficient tuning and potential existing solutions to make these techniques dynamic and search-free.

It has been shown in [1] that for classification tasks such as natural language understanding (NLU), PLMs have a low intrinsic dimension. This observation motivates the use of low-rank adapters for parameter-efficient tuning. There are several low-rank adapters in the literature such as LoRA [43], Adapter [39], Compacter [48], and Parallel Adapter (PA) [30]. LoRA is a low-rank up-projection/down-projection transformation without any non-linearity applied in parallel to key and value attention matrices. The main benefit of LoRA is that the adapter module, after training, can be integrated into the original weight matrices of the model, which in turn can lead to a very efficient inference time. Adapters also have a low-rank up-projection/down-projection transformation with an intermediate non-linearity. The Adapter module is applied in series with the feed-forward network (FFN). Having the adaptor module in-line with other blocks in the model can increase the inference time of the model. PA is a faster version of the Adapter, which can be applied in parallel with the FFN block. The compactor is a more memory-efficient version of the Adapter, which deploys the sum of Kronecker products to reconstruct each up-projection and down-projection matrices. All these low-rank adapters suffer from two major issues: first, finding the best rank requires heavy exhaustive training and search; second, the tuned adapter module works well only with a particular rank.

While there have been some efforts in the literature towards dynamic networks such as DynaBERT [37] and GradMax [25], to the best of our knowledge, this problem for factorized networks and low-rank adapters is still open. DRONE [16] propose a technique for data-aware low-rank model compression however their approach is not search-free, and also, it is not dynamic. DynaBERT introduces a two-stage method to train width and depth-wise dynamic networks. However, DynaBERT requires a fine-tuned teacher model on the task to train its sub-networks which makes it unsuitable for PET techniques. GradMax is a technique that gradually adds to the neurons of a network without touching the already trained neurons. But it is unclear how GradMax can be deployed to alleviate the rank-search problem in low-rank adapters. [113] propose a structured pruning technique called factorized low-rank pruning (FLOP). FLOP decomposes weight matrices of a network into the sum of rank-1 components, which are regularized during training to gain sparsity. It is worth mentioning that FLOP aims at compressing the main model, and even if it can be used for finding a good rank in the lower-rank representation of full-weight matrices, the final low-rank model will not be dynamic (i.e. it is trained well only for one rank and not a range of ranks, same as LoRA.). In this paper, we propose a new methodology for training low-rank modules for multiple ranks simultaneously rather than training a single-rank adapter at a time (without changing the training budget). Inspired by the idea of *nested dropout* [84], we pursue ordering the representations of the bottleneck in the low-rank adapter modules with a new recipe. To the best of our knowledge, this is the first time that the concept of ordering representations has been deployed in training PLMs.

3.4 Background

3.4.1 Nested Dropout

Inspired by the dropout [34], nested drop-out [84] is a stochastic regularization technique that targets enforcing ordered representations in training auto-encoders. The nested dropout, adds an implicit bias (which does not exist in dropout) to favor order in training. For example, in dropout, we can randomly drop any nodes or units in the network, but in nested dropout, if we randomly select k^{th} unit, then we keep all the units indexed from 1 to k and drop the units with indices larger than k. Therefore, nested dropout tends toward accommodating more important information in lower indices while learning representations.

Following the notations of [84], nested dropout assumes an auto-encoder mapping of N training examples $\{y_i\}_{i=1}^{N} \in Y$, $Y \subset \mathbb{R}^D$ to their corresponding representations $\{x_i\}_{i=1}^{N} \in X$, $X \subset \mathbb{R}^K$ using the function $f_\theta : Y \to X$ with parameters θ; and then decoding these representations using another function $g_\psi : X \to Y$ with parameters ψ to reconstruct the inputs. The reconstruction loss can be defined as follows:

$$C(\theta, \psi) = \sum_{i=1}^{N} ||y_i - g_\psi(f_\theta(y_i))||^2. \tag{3.1}$$

Suppose we want to randomly drop some units in our representation vector x. In this regard, we sample a random variable $b \sim p_B(.), b \in \{1, 2, ..., K\}$ from a pre-defined categorical distribution $p_B(.)$ and truncate the functions f_θ and g_ψ to keep their corresponding units indexed from 1 to b and dropping $b + 1$ to K indices. Let's define the b-truncated version of the vector x as $x_{\downarrow b}$ and the b-truncated version of the functions f_θ and g_ψ as $f_{\theta\downarrow b}$ and $g_{\psi\downarrow b}$ respectively. In this case, the reconstruction loss is redefined for the b-truncated model as follows:

$$\begin{aligned} &C(\theta, \psi) = \mathbb{E}_{p_B}[C_{\downarrow b}(\theta, \psi)] = \sum_{b=1}^{K} p_B(b) C_{\downarrow b}(\theta, \psi) \\ &\text{where} \\ &C_{\downarrow b}(\theta, \psi) = \sum_{i=1}^{N} ||y_i - g_{\psi\downarrow b}(f_{\theta\downarrow b}(y_i))||^2. \end{aligned} \tag{3.2}$$

In the final stage, the parameters of this model can be obtained by solving the following optimization problem.

$$(\theta^*, \psi^*) = \underset{\theta,\psi}{\text{argmin}}\, C(\theta, \psi). \tag{3.3}$$

While our work in this paper is inspired by the feature of ordering information suggested in nested dropout, we can distinguish our work from nested dropout in several aspects:

1. The nested dropout technique is used to add order information to a vector representation; however, we are adding order information to the low-rank matrix decomposition to make it work across a range of ranks instead of a single rank.

2. Our training algorithm differs from nested dropout in the choice of the distribution function $p_B(.)$, and we propose a more efficient individual loss for each truncated matrix compared to the linear summation loss (check equations 2 and 11 in the original paper [84]) in nested dropout. The original proposal for the nested dropout was to use a batch with mixed truncated examples. To enhance efficiency and resolve suboptimality, we propose to fix truncation in the entire batch as part of our approach.

3.4.2 LoRA: Low-rank Adapters

In LoRA [42], some pretrained weights of dense layers of PLMs are summed with parallel linear low-rank adapter modules. During fine-tuning, the original pretrained weights are kept frozen; LoRA modules can be updated instead. For example, let's assume that $W_0 \in \mathbb{R}^{m \times d}$ is a pretrained weight matrix in the network which is accompanied by a LoRA module $\Delta W = W_{up} W_{dw}$ where $W_{up} \in \mathbb{R}^{m \times r}$, $W_{dw} \in \mathbb{R}^{r \times d}$, and $r \ll min(m, d)$. Then, the output of this layer can be obtained as

$$h = W_0 x + \Delta W x = W_0 x + \frac{\alpha}{r} W_{up} W_{dw} x. \tag{3.4}$$

Bear in mind that the W_{up} matrix is initialized as a zero matrix, and the W_{dw} matrix is initialized as a zero-mean Gaussian distribution where α is a constant scale hyper-parameter.

In LoRA, the rank r is a hyperparameter that should be tuned for each task. Moreover, LoRA is a *static* low-rank adapter that works only with a particular size of r, which has been trained on it.

3.5 Our Method: DyLoRA

In this section, we introduce our solution to get dynamic low-rank adapters that can be trained and deployed well on a range of ranks instead of a single particular rank (with a fixed training budget). This flexibility can free us from searching for the best ranks by training the model multiple times.

Without loss of generality, we explain our solution on top of LoRA as one of the prominent low-rank adapter techniques in the literature. In each LoRA module, we have an up-projection ($W_{up} \in \mathbb{R}^{m \times r}$) and a down-projection matrix ($W_{dw} \in \mathbb{R}^{r \times d}$). Let's assume that we would like to train the LoRA module to operate in the range of $r \in \text{Range}[r_{min}, r_{max}]$ where r_{min} and r_{max} can be treated as new hyper-parameters. To make the LoRA module work in a range of ranks instead of a single rank, we need to ensure that increasing or decreasing the rank will not significantly hamper the model's performance. One way to implement such behavior would be by sorting the information content of different ranks in the training process of LoRA modules. In this regard, at each training step, we sample $b \sim p_B(.), b \in \{r_{min}, r_{min}+1, ..., r_{max}\}$ form a pre-defined categorical distribution (which has support in Range$[r_{min}, r_{max}]$) and truncate W_{dw} and W_{up} matrices accordingly.

$$\begin{aligned} W_{dw\downarrow b} &= W_{dw}[1:b,:] \\ W_{up\downarrow b} &= W_{up}[:,1:b] \end{aligned} \tag{3.5}$$

$W_{dw\downarrow b}$ and $W_{up\downarrow b}$ are b-truncated versions of W_{dw} and W_{up} respectively (see Fig. 4.1 for the visualization). Moreover, let's define W_{dw}^{b} as the bth row of W_{dw}; W_{up}^{b} corresponds to the bth column of W_{up}.

$$\begin{aligned} W_{dw}^{b} &= W_{dw}[b,:] \\ W_{up}^{b} &= W_{up}[:,b] \end{aligned} \tag{3.6}$$

Then, the forward pass of this truncated LoRA module during training will be calculated as follows:

$$h = W_0 x + \frac{\alpha}{b} W_{up\downarrow b} W_{dw\downarrow b} x \tag{3.7}$$

For simplicity, let's assume that we have only one LoRA module in the network (the one which is described in Eq. 3.7). Let's first consider the regular static loss function ($\mathcal{L}^{\mathcal{S}}$) of the network $f(x; W_{dw}, W_{up})$ with W_{dw} and W_{up} tunable parameters for N given input-output pairs $(\mathbf{x}, \mathbf{y}) = (x_i, y_i)_{i=1}^{N}$:

$$\begin{aligned} &\min_{W_{dw}, W_{up}} \mathcal{L}^{\mathcal{S}}(\mathbf{x}, \mathbf{y}; W_{dw}, W_{up}) \triangleq \\ &\sum_{i=1}^{N} l(f(x_i; W_{dw}, W_{up}), y_i). \end{aligned} \tag{3.8}$$

where $l(f, \mathbf{y})$ is a loss function that measures the divergence of network predictions compared with the target labels. Then, let's extend the training loss to make the network dynamic considering the b-truncation process. We can define our dynamic loss function $\mathcal{L}^{\mathcal{DY}}$ as follows.

$$\mathcal{L}_{\downarrow b}^{\mathcal{DY}} = \sum_{i=1}^{N} l(f(x_i; W_{dw\downarrow b}, W_{up\downarrow b}), y_i). \tag{3.9}$$

Bear in mind that, our loss function has a major difference from the nested dropout loss, which makes it more efficient. The nested dropout loss is in the form of

$$\sum_{b=r_{min}}^{r_{max}} p_B(b)\mathcal{L}^{\mathcal{DY}}_{\downarrow b}(\mathbf{x}, \mathbf{y}; W_{dw\downarrow b}, W_{up\downarrow b}) \tag{3.10}$$

which requires to sum the loss over the entire possible range of ranks and it is computationally expensive. To overcome this computational restriction, we replace it by optimizing the model parameters for each target rank individually at each time step. We show that this scheme quite works well.

The other difference with nested dropout is that in the parameter update phase, we add a new mode (so-called *frozen*) as a hyper-parameter to our training. This new mode suggests to only update the bth corresponding row and column sampled in the truncation phase (i.e. a single row or column will be updated at a time to prevent the learning parameters from being forgotten at previous time steps.). With a minor performance cost, this approach can improve the efficiency of our algorithm even further.

$$\begin{aligned} W^b_{dw} &\leftarrow W^b_{dw} - \eta\nabla_{W^b_{dw}}\mathcal{L}^{\mathcal{DY}}_{\downarrow b} \\ W^b_{up} &\leftarrow W^b_{up} - \eta\nabla_{W^b_{up}}\mathcal{L}^{\mathcal{DY}}_{\downarrow b} \end{aligned} \tag{3.11}$$

Table 3.4 shows the impact of only updating ”b” versus updating the columns and rows from 1 to b. The summary of our technique is described in Algorithm 2.

3.6 Experiments

In this section, we describe the experiments used to evaluate our DyLoRA model on both natural language understanding (NLU) and natural language generation (NLG) tasks. To be fair with the original LoRA method, we try to keep the setting of our experiments similar to the LoRA paper [42]. Therefore similarly, we chose the pretrained RoBERTa [61] base model as the backbone of the LoRA and DyLoRA experiments for the GLUE benchmark (Development Set), and GPT-Medium for the NLG tasks. For our experiments, we did not use any hyper-parameter tuning, nor did we search the validation epochs, nor did we use MLNI trick (use the MLNI checkpoint instead of the pretrained weights) to enhance the model's performance. More details about the hyperparameters is available in Table B.1 in Appendix C.4.2. In total, we conducted more than 200 experiments and evaluated more than 1600 models, details of which can be found in the attachments.

Algorithm 1 DyLoRA - Training

```
Require:
    $r \in$Range[$r_{min}$,$r_{max}$]; $i$: the number of training iterations; $\alpha$: a scaling factor; $p_B$: probability distribution function for rank selection; $X \in \mathbb{R}^{d \times n}$ : all input features to LoRA; $W_0 \in \mathbb{R}^{m \times d}$ the original frozen pretrained weight matrix
Require: $W_{dw} \in \mathbb{R}^{r \times d}$; $W_{up} \in \mathbb{R}^{m \times r}$, FROZEN: whether to keep the lower ranks frozen when updating the higher ranks
    while t $<$ $i$ do:
        Forward:
        // sample a specific rank, during test is given
        $b \sim p_B(.)$
        // truncate down-projection matrix
        $W_{dw\downarrow b} = W_{dw}$[:$b$,:]
        $W^b_{dw} = W_{dw}$[$b$,:]
        // truncate up-projection matrix
        $W_{up\downarrow b} = W_{up}$[:,:$b$]
        $W^b_{up} = W_{up}$[:,$b$]
        // calculate the LoRA output
        $h = W_0 X + \frac{\alpha}{b} W_{up\downarrow b} W_{dw\downarrow b} X$
        Backward:
        if FROZEN then
                // only update the unique parameters of the selected rank
                $W^b_{dw} \leftarrow W^b_{dw} - \eta \nabla_{W^b_{dw}} \mathcal{L}^{\mathcal{DY}}_{\downarrow b}$
                $W^b_{up} \leftarrow W^b_{up} - \eta \nabla_{W^b_{up}} \mathcal{L}^{\mathcal{DY}}_{\downarrow b}$
        else
                $W_{dw\downarrow b} \leftarrow W_{dw\downarrow b} - \eta \nabla_{W^b_{dw\downarrow b}} \mathcal{L}^{\mathcal{DY}}_{\downarrow b}$
                $W_{up\downarrow b} \leftarrow W_{up\downarrow b} - \eta \nabla_{W^b_{up\downarrow b}} \mathcal{L}^{\mathcal{DY}}_{\downarrow b}$
        end if
    end while
```

3.6.1 Baselines

- **Fine Tune**: To show a relative upper bound for the performance of our proposed method, we fine-tuned all the parameters in the model. Even though we have a large number of trainable parameters, this can help us better understand how higher-rank models perform.

- **LoRA**: As a baseline to DyLoRA, we employed the original LoRA model with their tuned

Model: RoBERTa-Base						
Task	**Rank=1**	**Rank=2**	**Rank=4**	**Rank=8**	**Rank=16**	**Rank=32**
QQP (Accuracy)	<u>89.14</u>	89.96	90.33	90.69	90.95	**91.02**
SST-2 (Accuracy)	<u>93.58</u>	94.15	94.38	**94.84**	94.27	94.5
MRPC (Accuracy)	87.25	87.75	88.24	87.25	<u>86.76</u>	**89.22**
CoLA (Mathews)	61.84	<u>57.78</u>	61.57	**63.81**	63.07	62.82

Table 3.1: The effect of the rank of the low-rank adaptation matrix over the performance of the model. In this experiment, all the other hyperparameters are fixed, and we only changed the rank of the LoRA model. In this search space, <u>Underline</u> shows the minimum performance rank, and the **bold** number shows the maximum performance rank.

hyperparameters [42]. As a result, most of the experiments have been conducted in a favorable manner for LoRA.

- **FLOP**: Due to its flexibility, **F**actorized **Lo**w Rank **P**runing (FLOP) [113] can be applied to any matrix multiplication and, therefore, can be used to avoid the search in our problem. However, this baseline lacks the dynamic properties of DyLoRA. We used it to show regularization-based techniques' performance and pros and cons.

3.6.2 LoRA rank selection problem

There is no clear guidance on how to determine the rank for the LoRA algorithm. It is evident in the LoRA paper [42] that the performance of models varies a lot with different ranks (e.g. check Tables 15, and 18 in the LoRA paper), and does not indicate any clear trend. We also observe the same problem in the GLUE benchmark. We may argue that theoretically, the rank with the best performance is always the highest. High ranks, however, introduce additional parameters into the adaptive process and this might be undesirable. In practice, as demonstrated in Table 3.1, the most effective rank differs depending on the task. For example, based on the MRPC results, the rank with the lowest performance is 16 while the rank with the highest performance is 32. This is different from SST-2, in which rank 1 is the least performing rank and rank 8 is the most effective rank. Many factors can contribute to this difference, including but not limited to the size of the dataset, hyperparameter selections, hardware configurations and the optimization.

	Accuracy	**Accuracy**	**F1**	**Mathews**	**Accuracy**	**Accuracy**	**Accuracy**	**Pearson**	
Model	**MNLI**	**SST-2**	**MRPC**	**CoLA**	**QNLI**	**QQP**	**RTE**	**STS-B**	**Avg**
Rank = 1									
LoRA	$34.60_{\pm 3.69}$	$69.61_{\pm 7.99}$	$83.47_{\pm 3.90}$	$25.57_{\pm 9.71}$	$53.00_{\pm 2.95}$	$44.30_{\pm 7.50}$	$57.55_{\pm 5.51}$	$76.07_{\pm 6.06}$	54.90
DyLoRA (Frozen)	$85.36_{\pm 0.26}$	$93.51_{\pm 0.49}$	$90.75_{\pm 0.70}$	$56.95_{\pm 1.54}$	$91.70_{\pm 0.28}$	$87.87_{\pm 0.17}$	$66.79_{\pm 8.54}$	$89.95_{\pm 0.24}$	82.86
DyLoRA	$85.59_{\pm 0.07}$	$93.23_{\pm 0.63}$	$91.58_{\pm 0.69}$	$57.93_{\pm 2.12}$	$91.95_{\pm 0.14}$	$88.37_{\pm 0.15}$	$74.80_{\pm 1.48}$	$90.30_{\pm 0.13}$	**84.22**
Rank = 2									
LoRA	$40.53_{\pm 6.17}$	$82.75_{\pm 5.08}$	$88.00_{\pm 1.81}$	$43.30_{\pm 4.67}$	$63.42_{\pm 2.99}$	$59.21_{\pm 6.13}$	$68.88_{\pm 1.26}$	$85.51_{\pm 1.94}$	66.45
DyLoRA (Frozen)	$85.74_{\pm 0.28}$	$93.76_{\pm 0.52}$	$91.09_{\pm 0.45}$	$56.88_{\pm 2.09}$	$92.03_{\pm 0.22}$	$88.21_{\pm 0.07}$	$63.90_{\pm 12.85}$	$90.25_{\pm 0.15}$	82.73
DyLoRA	$86.02_{\pm 0.06}$	$93.81_{\pm 0.30}$	$91.66_{\pm 0.46}$	$59.91_{\pm 1.88}$	$92.39_{\pm 0.25}$	$89.33_{\pm 0.05}$	$76.03_{\pm 1.61}$	$90.60_{\pm 0.09}$	**84.97**
Rank = 3									
LoRA	$58.95_{\pm 6.02}$	$90.00_{\pm 1.27}$	$89.66_{\pm 1.25}$	$56.78_{\pm 1.88}$	$79.26_{\pm 4.80}$	$72.58_{\pm 4.09}$	$72.49_{\pm 2.30}$	$88.80_{\pm 0.29}$	76.07
DyLoRA (Frozen)	$85.78_{\pm 0.25}$	$93.76_{\pm 0.26}$	$91.78_{\pm 0.89}$	$58.86_{\pm 0.32}$	$92.17_{\pm 0.18}$	$88.40_{\pm 0.0}$	$70.90_{\pm 6.14}$	$90.50_{\pm 0.29}$	84.02
DyLoRA	$86.70_{\pm 0.09}$	$94.11_{\pm 0.33}$	$91.56_{\pm 0.86}$	$60.97_{\pm 2.01}$	$92.77_{\pm 0.21}$	$89.76_{\pm 0.07}$	$77.11_{\pm 2.97}$	$90.69_{\pm 0.14}$	**85.46**
Rank = 4									
LoRA	$72.10_{\pm 5.25}$	$91.56_{\pm 0.34}$	$89.62_{\pm 0.92}$	$58.53_{\pm 3.93}$	$85.09_{\pm 1.20}$	$80.78_{\pm 3.73}$	$73.07_{\pm 2.29}$	$89.28_{\pm 0.72}$	80.00
DyLoRA (Frozen)	$85.93_{\pm 0.19}$	$93.85_{\pm 0.33}$	$91.28_{\pm 0.71}$	$59.25_{\pm 1.05}$	$92.27_{\pm 0.16}$	$88.52_{\pm 0.08}$	$71.12_{\pm 2.46}$	$90.53_{\pm 0.18}$	84.10
DyLoRA	$86.82_{\pm 0.04}$	$94.40_{\pm 0.13}$	$92.06_{\pm 0.46}$	$59.81_{\pm 1.71}$	$92.91_{\pm 0.31}$	$89.80_{\pm 0.10}$	$77.40_{\pm 2.72}$	$90.86_{\pm 0.06}$	**85.53**
Rank = 5									
LoRA	$78.61_{\pm 3.97}$	$92.82_{\pm 0.46}$	$90.75_{\pm 0.96}$	$60.37_{\pm 3.10}$	$88.97_{\pm 0.90}$	$85.26_{\pm 1.56}$	$73.21_{\pm 2.17}$	$89.90_{\pm 0.30}$	82.49
DyLoRA (Frozen)	$85.95_{\pm 0.17}$	$93.78_{\pm 0.26}$	$91.28_{\pm 0.64}$	$59.41_{\pm 1.30}$	$92.30_{\pm 0.17}$	$88.56_{\pm 0.09}$	$71.48_{\pm 2.92}$	$90.60_{\pm 0.20}$	84.17
DyLoRA	$87.00_{\pm 0.10}$	$94.29_{\pm 0.41}$	$91.73_{\pm 0.60}$	$60.52_{\pm 1.07}$	$93.01_{\pm 0.28}$	$90.04_{\pm 0.10}$	$76.90_{\pm 2.11}$	$90.97_{\pm 0.20}$	**85.56**
Rank = 6									
LoRA	$83.02_{\pm 1.59}$	$93.49_{\pm 0.88}$	$91.28_{\pm 0.63}$	$61.94_{\pm 2.27}$	$90.32_{\pm 0.76}$	$87.54_{\pm 1.51}$	$76.68_{\pm 1.16}$	$90.12_{\pm 0.12}$	84.30
DyLoRA (Frozen)	$85.98_{\pm 0.16}$	$93.76_{\pm 0.46}$	$91.12_{\pm 0.43}$	$58.95_{\pm 1.10}$	$92.46_{\pm 0.14}$	$88.68_{\pm 0.13}$	$72.64_{\pm 2.44}$	$90.64_{\pm 0.23}$	84.28
DyLoRA	$86.97_{\pm 0.20}$	$94.27_{\pm 0.37}$	$91.44_{\pm 0.64}$	$60.16_{\pm 1.70}$	$93.01_{\pm 0.21}$	$90.07_{\pm 0.14}$	$77.33_{\pm 1.66}$	$91.03_{\pm 0.20}$	**85.53**
Rank = 7									
LoRA	$85.44_{\pm 0.78}$	$93.62_{\pm 0.35}$	$91.27_{\pm 0.73}$	$62.19_{\pm 2.66}$	$91.88_{\pm 0.23}$	$89.51_{\pm 0.30}$	$75.52_{\pm 1.41}$	$90.35_{\pm 0.24}$	84.97
DyLoRA (Frozen)	$86.08_{\pm 0.14}$	$93.97_{\pm 0.17}$	$91.02_{\pm 0.70}$	$58.76_{\pm 0.94}$	$92.30_{\pm 0.10}$	$88.77_{\pm 0.06}$	$73.50_{\pm 1.67}$	$90.68_{\pm 0.15}$	84.38
DyLoRA	$86.82_{\pm 0.10}$	$94.27_{\pm 0.33}$	$91.38_{\pm 0.59}$	$59.51_{\pm 1.75}$	$92.99_{\pm 0.26}$	$90.04_{\pm 0.06}$	$77.91_{\pm 1.58}$	$91.07_{\pm 0.19}$	**85.50**
Rank = 8									
LoRA	$86.82_{\pm 0.18}$	$94.01_{\pm 0.30}$	$91.48_{\pm 0.73}$	$62.08_{\pm 1.37}$	$92.39_{\pm 0.39}$	$90.42_{\pm 0.02}$	$74.51_{\pm 0.41}$	$90.48_{\pm 0.24}$	85.27
DyLoRA (Frozen)	$86.10_{\pm 0.04}$	$93.69_{\pm 0.41}$	$91.19_{\pm 0.79}$	$58.52_{\pm 0.95}$	$92.47_{\pm 0.18}$	$88.82_{\pm 0.06}$	$73.29_{\pm 2.49}$	$90.68_{\pm 0.14}$	84.35
DyLoRA	$86.76_{\pm 0.13}$	$94.36_{\pm 0.38}$	$91.38_{\pm 0.83}$	$59.51_{\pm 1.84}$	$93.00_{\pm 0.32}$	$89.91_{\pm 0.08}$	$77.55_{\pm 0.59}$	$91.05_{\pm 0.19}$	**85.44**
Best (Rank)									
LoRA	87.03(8)	94.50(6)	92.25(7)	**66.05**(7)	92.81(8)	**90.45**(8)	77.98(6)	90.87(8)	86.49
DyLoRA (Frozen)	86.18(7)	94.50**(2)**	**92.93(3)**	61.57(5)	92.70**(6)**	88.88(8)	75.81(7)	90.89**(6)**	85.43
DyLoRA	**87.17(6)**	**94.72**(7)	92.79(8)	63.32**(3)**	**93.56**(8)	90.17**(6)**	**80.14(4)**	**91.36**(7)	**86.66**
Full Rank									
Fine Tune*	**87.6**	**94.8**	90.2	63.6	92.8	**91.9**	78.7	91.2	86.4

Table 3.2: In this table, the task is to find a low-rank adaptation matrix that works with different ranks at inference time given a fixed budget (training time).

3.6.3 Dynamic low rank adaptation

For example, suppose we have a neural network that we wish to deploy on various devices with different configurations. The use of higher ranks may pose a problem for very sensitive devices

		Accuracy	F1	Accuracy	Pearson	
Model (Rank)	**Trainable Params**	**SST-2**	**MRPC**	**QNLI**	**STS-B**	**AVERAGE**
Fine Tune*	125M	**94.8**	90.2	92.8	**91.2**	92.25
FLOP*	80M	92.09	88.61	89.05	88.18	89.48
LoRA (1)	**0.628M**	93.58	91.93	91.98	90.85	92.09
Maximum Rank: $r_{max} = 8$						
DyLoRA (1)	**0.628M**	$93.23_{\pm 0.63}$	$91.58_{\pm 0.69}$	$91.95_{\pm 0.14}$	$90.30_{\pm 0.13}$	91.77
DyLoRA (8)	0.887M	$94.36_{\pm 0.38}$	$91.38_{\pm 0.83}$	$93.00_{\pm 0.32}$	$91.05_{\pm 0.19}$	**92.45**

Table 3.3: This table compares DyLoRA with compression-based algorithms. As indicated by *, we reported "Fine Tune" and FLOP from their original papers, [61] and [113]. To the best of our knowledge, experiments were conducted under the same experimental setting. We count all the trainable parameters including classifier, unlike LoRA paper [42] which they count only LoRA specific parameters.

Maximum Rank: $r_{max} = 8$								
$b \sim P_B$: *Distribution*	*Updated Parameters*	Accuracy **SST-2**	F1 **MRPC**	Mathews **CoLA**	Accuracy **QNLI**	Accuracy **RTE**	Pearson **STS-B**	**AVERAGE**
		Rank=8						
Geometric (p=0.15)	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	$93.97_{\pm 0.33}$	$90.84_{\pm 1.15}$	$58.95_{\pm 1.95}$	$92.74_{\pm 0.13}$	$74.80_{\pm 0.90}$	$90.66_{\pm 0.15}$	83.66
	W_{dw}^{b},W_{up}^{b}	$93.60_{\pm 0.24}$	$90.50_{\pm 0.42}$	$58.19_{\pm 1.17}$	$92.26_{\pm 0.12}$	$71.91_{\pm 1.74}$	$90.20_{\pm 0.36}$	82.78
Uniform	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	$94.36_{\pm 0.38}$	$91.38_{\pm 0.83}$	$59.51_{\pm 1.84}$	$93.00_{\pm 0.32}$	$77.55_{\pm 0.59}$	$91.05_{\pm 0.19}$	**84.47**
	W_{dw}^{b},W_{up}^{b}	$93.69_{\pm 0.41}$	$91.19_{\pm 0.79}$	$58.52_{\pm 0.95}$	$92.47_{\pm 0.18}$	$73.29_{\pm 2.49}$	$90.68_{\pm 0.14}$	83.31
		Rank=1						
Geometric (p=0.15)	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	$93.53_{\pm 0.47}$	$91.36_{\pm 0.72}$	$59.43_{\pm 1.12}$	$92.24_{\pm 0.08}$	$73.65_{\pm 3.55}$	$90.33_{\pm 0.14}$	**83.42**
	W_{dw}^{b},W_{up}^{b}	$93.58_{\pm 0.26}$	$90.81_{\pm 0.83}$	$58.55_{\pm 1.13}$	$92.27_{\pm 0.28}$	$68.52_{\pm 11.88}$	$90.60_{\pm 0.31}$	82.39
Uniform	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	$93.23_{\pm 0.63}$	$91.58_{\pm 0.69}$	$57.93_{\pm 2.12}$	$91.95_{\pm 0.14}$	$74.80_{\pm 1.48}$	$90.30_{\pm 0.13}$	83.30
	W_{dw}^{b},W_{up}^{b}	$93.51_{\pm 0.49}$	$90.75_{\pm 0.70}$	$56.95_{\pm 1.54}$	$91.70_{\pm 0.28}$	$66.79_{\pm 8.54}$	$89.95_{\pm 0.24}$	81.61

Table 3.4: Ablation Study - In this experiment, our goal is to demonstrate how the introduced distribution can affect the performance of DyLoRA.

as they have a greater number of parameters. Therefore, we must either train several models with different configurations or find the most optimal rank. The cost associated with this is significant, as even in the setting of LoRA, we are required to find the best rank for each task and each device. Using DyLoRA, however, one needs to train one model per task and, as our method is adaptive at inference time, we can deploy it according to our needs. In Table 3.2, we demonstrate the dynamic properties of DyLoRA. In order to ensure a fair comparison, all LoRA and DyLoRA models in this table have the same model size, we used the same code and evaluation process, and all models were trained to the same extent. In LoRA, we lose performance when performing inferences for the lower ranks. This occurs because the model has been trained only for rank 8 during training. In DyLoRA, we preserve a high level of performance for lower ranks while competing well with LoRA on rank 8.

Model	**Time**	**SST-2** (r)	**MRPC** (r)
	Maximum Rank: $r_{max} = 64$		
LoRA (Search)	$7x$	**95.3**(64)	89.71(64)
DyLoRA (Frozen)	**1x**	94.38(**7**)	**89.95**(**34**)
	Maximum Rank: $r_{max} = 32$		
LoRA (Search)	$6x$	**94.84**(32)	88.73(16)
DyLoRA (Frozen)	**1x**	94.38(**7**)	**89.71**(**5**)

Table 3.5: In this table, the search space of rank is larger compared to the previous experiment and the goal is to find the most optimal rank for the low-rank adaptation of a pre-rained RoBERTa-Base. For LoRA (Search), we ran experiments for ranks=1,2,4,8,16,32,64 and we reported the best results. In the Exhaustive Search, one has to search all the ranks from 1 to 64, which means it will cost 64 times more than our proposed method. The lower the rank the better, and the higher the performance is the better.

Model (Method)	**Updated Params**	**Trainable Params**	E2E NLG Challenge				
			BLEU	**NIST**	**MET**	**ROUGE-L**	**CIDEr**
			Rank=1				
GPT-2 M (LoRA)		0.09M	3.38	1.18	9.23	18.79	0.12
GPT-2 M (DyLoRA)	W^b_{dw},W^b_{up}	0.09M	$67.92_{\pm 0.20}$	$8.65_{\pm 0.06}$	$44.91_{\pm 0.38}$	$69.07_{\pm 0.32}$	$2.38_{\pm 0.04}$
GPT-2 M (DyLoRA)	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	0.09M	$68.86_{\pm 0.55}$	$8.72_{\pm 0.04}$	$45.81_{\pm 0.40}$	$70.33_{\pm 0.64}$	$2.43_{\pm 0.04}$
			Rank=2				
GPT-2 M (LoRA)		0.19M	46.99	6.39	34.19	56.10	1.27
GPT-2 M (DyLoRA)	W^b_{dw},W^b_{up}	0.19M	$68.81_{\pm 0.49}$	$8.75_{\pm 0.02}$	$45.23_{\pm 0.22}$	$69.81_{\pm 0.30}$	$2.41_{\pm 0.01}$
GPT-2 M (DyLoRA)	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	0.19M	$68.97_{\pm 1.03}$	$8.75_{\pm 0.07}$	$45.88_{\pm 0.55}$	$70.07_{\pm 0.86}$	$2.43_{\pm 0.04}$
			Rank=3				
GPT-2 M (LoRA)		0.29M	63.68	8.46	42.37	65.84	2.24
GPT-2 M (DyLoRA)	W^b_{dw},W^b_{up}	0.29M	$68.41_{\pm 1.00}$	$8.69_{\pm 0.10}$	$45.31_{\pm 0.64}$	$69.75_{\pm 0.69}$	$2.42_{\pm 0.02}$
GPT-2 M (DyLoRA)	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	0.29M	$69.33_{\pm 0.26}$	$8.76_{\pm 0.05}$	$46.19_{\pm 0.22}$	$70.56_{\pm 0.43}$	$2.46_{\pm 0.01}$
			Rank=4				
GPT-2 M (LoRA)		0.39M	69.88	8.81	46.81	72.10	2.53
GPT-2 M (DyLoRA)	W^b_{dw},W^b_{up}	0.39M	$68.36_{\pm 0.41}$	$8.70_{\pm 0.02}$	$45.46_{\pm 0.56}$	$69.91_{\pm 0.50}$	$2.43_{\pm 0.01}$
GPT-2 M (DyLoRA)	$W_{dw\downarrow b}$,$W_{up\downarrow b}$	0.39M	$69.19_{\pm 0.43}$	$8.75_{\pm 0.03}$	$46.26_{\pm 0.47}$	$70.78_{\pm 0.63}$	$2.46_{\pm 0.02}$
			Fine-Tune				
GPT-2 M (FT)*		354M	68.2	8.62	46.2	71.0	2.5

Table 3.6: For all metrics, higher is better. Rows with * have been reported based on the LoRA paper. Unlike [42], we included the classifier number of parameters in our trainable parameters count.

3.6.4 Search-free low rank adaptation

The process of selecting a particular rank can be expensive as previously mentioned. In Table 3.5, we present an experiment that illustrates the costs associated with such a search for LoRA and DyLoRA. As an example, if one naively wanted to search the entire range of ranks (for example, 64 in the experiment), then they would have to train and evaluate 64 distinct models in order to determine the proper rank. It becomes even more expensive if one searches the entire rank space. In the case of uniform search, this cost is less, yet still more expensive (7 times in the experiment) than our proposed method. Therefore, for LoRA (Search), we ran experiments for ranks=1,2,4,8,16,32,64 and we reported the best results. The results demonstrate that our proposed method performs competitively at a much lower cost.

3.6.5 Robustness of DyLoRA

As illustrated in Table 3.2, DyLoRA is quite robust to randomness and can produce consistently good results due to stable convergence.

3.6.6 Regularization and Pruning

An alternative method of avoiding the search problem is using regularization/pruning techniques to determine the intrinsic rank of the weight matrix. In this way, we can reduce the number of parameters of the original matrices; however, we will not have a dynamic model during inference. To illustrate the difference between such methods and DyLoRA, we reported the performance of one of these models, FLOP [113], in Table 3.3. FLOP utilizes low-rank factorization to create new matrices representing the original weight matrix. Thus, they will have fewer total parameters but require more trainable parameters to reach a comparable performance to DyLoRA.

3.6.7 Generative Tasks

In this experiment, we evaluate the performance of our model on different natural language generation (NLG) tasks such as the E2E NLG Challenge [69], DART [67] and WebNLG [29]. The results of the E2E task are shown in Table 3.6 and due to the space limit, the results of the other two tasks are demonstrated in Appendix B.3. The generative tasks demonstrate a similar pattern as the NLU task, showing that our model is able to work well at wider range of ranks compared to LoRA.

3.6.8 Ablation study

In this subsection, we investigate the impact of two design choices in DyLoRA: first, the new distribution P_B hyper-parameter in our technique; second, the impact of updating W^b_{dw} and W^b_{up} parameters instead of the entire $W_{dw\downarrow b}$ and $W_{up\downarrow b}$. The distribution P_B changes the relative importance of the different ranks during the training process. To examine the impact of the chosen distribution on DyLoRA's performance, we used two distributions, geometric and uniform. As shown in Table 3.4, the geometric distribution provides a much better method for optimizing the lower ranks, since it pays much more attention to the lower ranks during training, and uniform distribution will give better performance over all ranks. We chose to use uniform distribution in most of our experiments to avoid adding another hyperparameter, which is a requirement of the geometric distribution. Moreover, we demonstrate that it is possible to ensure that the optimization of rank b will not negatively affect the performance of the lower ranks (1 to $b-1$), while performing reasonably well. As mentioned, this can be accomplished by only updating the unique parameters associated with rank r that do not overlap with lower ranks.

In addition, in Table 3.7, we demonstrate the result of using our individual loss (Eq. 3.9) vs. the nested dropout original objective function in an equal setting. As shown, our proposed objective function is both effective and efficient. Furthermore, it is important to note that the summation loss is not scalable when many ranks are involved. We also discussed the time complexity of LoRA and DyLoRA in Appendix B.1.

Maximum Rank: $r_{max} = 8$		
Loss	**Training Time**	**CoLA**
$\mathcal{L}^{\mathcal{DY}}_{\downarrow b}$	**645.82s**	52.64
$\sum p_B(b)\mathcal{L}^{\mathcal{DY}}_{\downarrow b}$	1175.69s	**54.12**

Table 3.7: This experiment shows the impact of choosing individual loss vs. summation loss functions on our training. The average performance across all possible ranks (1,2,...,8) is reported. For summation loss to be computationally more feasible, smaller epochs were chosen. A total of seven GPUs were used in this experiment.

3.7 Conclusion

In this paper, we presented our solution DyLoRA to address two problems in low-rank adapters regarding rank selection and making them dynamic. We showed that DyLoRA can select the rank without requiring multiple re-training and is able to make LoRA dynamic at inference time.

As a result, we can avoid the process of searching for the most optimal ranks for many real-life scenarios. It has been demonstrated that DyLoRA performance is comparable with LoRA, yet we can support a wider range of ranks without adding additional time and effort.

Limitations

According to LoRA [42], a proper choice of the scalar can improve the results. In order to determine what is the best choice, further investigation is required. Despite our demonstration that uniform distribution can be as effective as specific geometric distribution, further investigation is necessary to evaluate the effect of different distributions on different downstream tasks. As shown in this paper, our algorithm works over a wide range of ranks, but further research is needed to understand the impact of choosing a particular range.

Chapter 4

SortedNet, a Place for Every Network and Every Network in its Place: Towards a Generalized Solution for Training Many-in-One Neural Networks

4.1 Abstract

Deep neural networks (DNNs) must cater to a variety of users with different performance needs and budgets, leading to the costly practice of training, storing, and maintaining numerous specific models. There are solutions in the literature to deal with single dynamic or many-in-one models instead of many individual networks; however, they usually suffer from heavy model search requirements, being architecture-specific, working only on a limited number of dimensions (e.g. depth only or width only) or sub-models. To address these problems, we propose SortedNet, a generalized and scalable training solution to harness the inherent modularity of DNNs. Thanks to a generalized nested architecture (which we refer to as *sorted* architecture in this paper) with shared parameters and its novel update scheme combining random sub-model sampling and gradient accumulation, SortedNet enables the training of numerous sub-models simultaneously, simplifies dynamic model selection and deployment during inference, and reduces the model storage requirement significantly. The versatility and scalability of SortedNet are validated through various architectures and tasks including LLaMA, BERT, RoBERTa (NLP tasks), ResNet and MobileNet (image classification) demonstrating its superiority over existing dynamic training methods. SortedNet is able to train up to 160 sub-models at once, achieving at

least 96% of the original model's performance.

4.2 Introduction

"For every minute spent organizing, an hour is earned." - Benjamin Franklin.

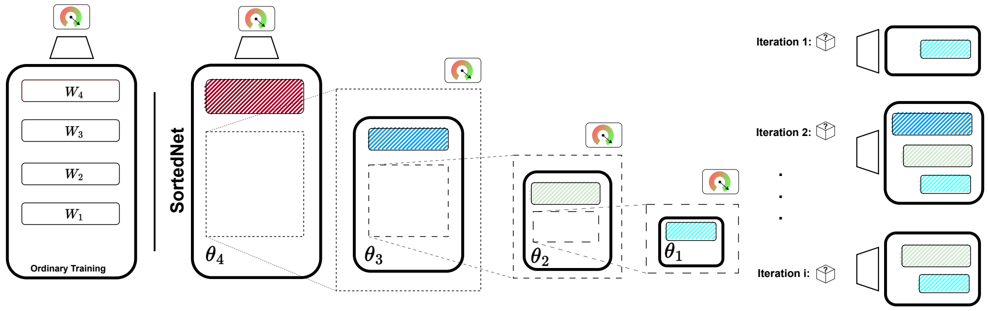


Figure 4.1: The overall diagram of our SortedNet training approach. First, we need to define the pool of sub-models of interest including the main model as well. During training, at each iteration, we sample from the pool of sub-models (given a pre-defined random distribution) to be trained for the target loss function (for one step).

Deep neural networks (DNNs) are increasingly gaining interest and becoming more popular [88]. This popularity translates to the increasing demand and requirements from the users which should be met by these models. People pre-train or fine-tune more models with various sizes to address the performance and computational needs of their tasks and target devices (with different memory and computational power whether deployed in the cloud or on edge devices). However, developing, storing, maintaining, and deploying many individual models for diverse set of users can be very difficult and costly [22]. Moreover, in the era of gigantic pre-trained models [20, 61] and large language models [9, 17] the computational demands can vary significantly from task to task. Therefore, there is a growing demand for models which can adapt themselves to the dynamic conditions, while conventional neural network would fail to address such cases [117, 119].

On the other hand, DNNs demonstrate modular architectures along various dimensions, like layers and blocks across depth, and neurons, channels and attention heads along width. This inherent modularity enables the extraction of sub-models with similar shapes to the original model. However, this modularity has not been deployed in regular training methods, and consequently, the performance of the sub-models falls short compared to the main model. Hence, the challenge lies in harnessing the full potential of modularity in deep neural networks, allowing for the efficient utilization of sub-models to enhance their performance and enable their practical deployment in real-world scenarios.

Instead of training individual models, we can leverage sub-models of DNNs and train them together with the main models to obtain many-in-one networks with sub-models that can be used for different tasks. There are variety of approaches in the literature for training sub-models [12, 117, 37]. These techniques while effective have certain shortcomings: often use a sophisticated training process combined with knowledge distillation (which needs to train a separate teacher model) [37], require architecture modification [70], work for specific architectures only [11], cannot handle more than a very small number of sub-models, need heavy model search (e.g. neural architecture search) during training or inference [12], involve redundant sub-model optimization [26], or show poor performance for the main model or sub-models [117].

To address these problems, we propose SortedNet, a generalized and scalable training solution to harness the inherent modularity of DNNs across various dimensions. As the name of our method implies, it chooses the sub-models in a sorted manner (a generalized version of nested architectures) within the main model to avoid heavy search during or after training. In contrast to nested models in which smaller sub-models are always totally encapsulated by larger sub-models, our generalized sorted version relaxes the nested constraint but ties the origin of sub-models to the origin of the main model across any target dimension (for more details see Appendix C.1). This sorted configuration with shared parameters enforces regular order and consistency in the knowledge learned by sub-models. One option to sort the sub-models is based on their computation and accuracy requirements which will enable us to extract our desired sub-models without requiring extensive search at the test time. The use of a predefined sorting order ensures that each targeted sub-model possesses a unique computation overhead, effectively removing optimization of redundant sub-models from training.

To train the sorted sub-models, we propose a novel updating scheme that combines random sampling of sub-models with gradient accumulation. We tried the SortedNet solution successfully on various architectures and tasks such as the decoder-based LLaMA (13B) large language models [99] on the GSM8K [18] mathematical reasoning task, encoder-based BERT [20] and RoBERTa [61] on the set of GLUE [108] language understanding tasks, ResNet [31] and MobileNet [87] on the CIFAR-10 image classification task. Our comprehensive empirical studies across different architectures, tasks and dynamicity along various dimensions ranging from width and depth to attention head and embedding layer show the superiority and generalizabilty of our proposed method over state of the art dynamic training methods. Moreover, SortedNet offers several benefits, including minimal storage requirements and dynamic inference capability (i.e. switching between various computation budgets) during inference.

To summarize, the main contributions of this paper are:

- Introducing a many-in-one solution to configure sub-models in a sorted manner and training them simultaneously with some unique aspects such as scalability (training many sub-models), generality (CNN, Transformers, depth, width), and search-free (no need for search during training or inference among sub-models) and maintaining competitive performance of the main model.
- Outperforming state-of-the-art methods in dynamic training on CIFAR10 [53]. Furthermore, scaling the number of sub-models to 160 and achieving at least 96% of the original model's performance showcases the efficacy of our single round training method.
- Demonstrating the effectiveness of the proposed method on Large pre-trained language models by dynamic training of the BERT, RoBERTa and LLaMA models.

4.3 Related Work

Table 4.1: Comparison of different existing related work and distinguishing our solution

Method	**Sub-Models: Config.(#)**	**Performance**	**Anytime**	**Search-Free**	**# of Trainable Params**	**Target Dim.**	**Architecture**
Early Exit [117]	Sorted (Few)	Low	✓	✓	$\|\theta\|$	Depth	Transformer
Layer Drop [26]	Random (Many)	Low	✗	✗	$\|\theta\|$	Depth	Transformer
DynaBERT [37]	Sorted (Few)	High	✗	✗	$2\|\theta\|$	Depth & Width	Transformer
Once for All [11]	Nested (Many)	High	✗	✗	$\|\theta\|$ or $2\|\theta\|$	General	CNN
LCS [70]	Arbitrary (Many)	High	✓	✓	$\|\theta\|$ or $2\|\theta\|$	General	CNN
Slimmable [119]	Sorted (Few)	Moderate	✓	✓	$\|\theta\|$	Width	CNN
MatFormer [22]	Sorted (Few)	High	✗	✓	$\|\theta\|$	Width	Transformer
SortedNet (Ours)	Sorted (**Many**)	**High**	✓	✓	$\|\theta\|$	**General**	CNN & Transformer

In this section, we briefly review the most relevant existing works to our SortedNet idea. A summary of these solutions and how they are different from each other can be found in Table 4.1. For more details, please refer to appendix C.2.

Slimmable Networks [120] Slimmable networks is a width adjustable training method. It was proposed particularly for CNN architectures and thus, careful consideration of the batch normalization module for various width sizes is necessary. In contrast to slimmable networks, our SortedNet covers more architectures and works in both depth and width dimensions.

Early Exit [117] is one of the most popular baseline techniques which adds a classifier to intermediate layers of an already trained model. The parameters of the main model are frozen and the classifiers are updated in a separate fine-tuning process. While this solution is relatively straightforward, the performance of the sub-models lags significantly behind that of the main model.

Dayna-BERT [37] presents a dynamic compression method for pre-trained BERT models, enabling flexible adjustments in model size, both in depth and width, during inference. DynaBERT is different from us in the follow aspects: first, in DynaBERT, only a very few sub-models are functional; second, DynaBERT requires an already trained teacher model and utilizes knowledge distillation (KD); third, DynaBERT needs search to find an optimal sub-model; last, DynaBERT is architecture dependent.

Layer-drop [26] is a structured dropout training which allows layer pruning at the inference time. Similar to DynaBERT, it is applied to pre-trained language models; however, in contrast to DynaBERT, Layer-drop only targets the depth of neural networks and not their width.

Once-for-All (OFA) [12] targets efficient inference across different devices. It first trains a network which supports many sub-models with varying latency/accuracy characteristics ; it then searches among the feasible sub-models according to the accuracy and latency requirements of their target device. OFA is different from our solution in: first, it has a progressive training nature in contrast to our stochastic or summation loss; second, it needs teacher and KD; third, it requires a separate neural architecture search (NAS) at the inference time; fourth, OFA is for CNN-based models; last, it does not have any particular assumption for configuring sub-models (see Fig. C.2 for more details).

Learning Compressible Subspace (LCS) [70] is an adaptive compression technique based on training compressible subspace of neural networks. While LCS does not require any re-training at the inference time, this solution has some other limitations including: first, it needs double memory at the training time; second, the choices of initial weights and the compression function are unclear and arbitrary (left as a hyper-parameter); third, it is only tried on CNNs; forth, similar to Layer-drop, the search space of sub-models is huge which makes the training sub-optimal.

MatFormer [22] is a pre-training only many-in-one solution based on summation loss for Transformer-based models. MatFormer works only along the width dimension of the FFN block in Transformers and cannot handle more than a very few number of sub-models.

4.4 Proposed Method

4.4.1 A Generalized and Scalable View

In the related work section, we have discussed several approaches concerning the training of many-in-one networks. These approaches differ in terms of their target architecture, training loss, number of training parameters, the configuration of the sub-models (random, nested, or sorted), the number of trained sub-models, and reliance on search or re-training before deployment. Our SortedNet method can be viewed as a simple, general, and scalable version of these existing solutions. These benefits have mostly resulted from the sorted configuration of sub-models with their shared parameters and our stochastic training. To the best of our knowledge, this is the first work that has scaled training of sorted sub-models to various dimensions and different architecture types.

In this subsection, we present SortedNet, a generalized and scalable method for training many-in-one neural networks. In order to train many-in-one networks, we need to specify a few design choices: first, how to form the sub-models and their configurations; second, what are the target architectures; and third, how to train the sub-models along with the main model.

Designing the sub-models SortedNet imposes an inductive bias on training based on the assumption that the parameters of sub-models have a concentric architecture tied to the origin along each dimension (which we refer to as a *sorted architecture*). This sorted configuration with shared parameters enforces a regular order and consistency in the knowledge learned by sub-models (see Fig. 4.1).

Let's consider a many-in-one neural network $f(x;\theta(n))$ with the parameters $\theta(n)$ and the input x which is comprised of n sub-models $f(x;\theta(i))|_{i=0}^{n-1}$, where $\theta(i)$ represents the weights of the ith sub-model. We define a universal set which contains all unique sub-models: $\Theta = \{\theta(0), \theta(1), ..., \theta(n)\}$.

Setting up an order Suppose that we would like to target $D = \{Dim_1, Dim_2, ..., Dim_K\}$ many-in-one dimensions in our model. Then, let's start with $\Theta = \varnothing$ and build the sub-models iteratively. In this regard, at each iteration t during training, we have sampling and truncation procedures along any of the targeted dimensions:

$$\begin{aligned} &\theta_t^* = \cap_{j=1}^{|D|} \theta_{Dim_j \downarrow b_j^t}(n) \quad \text{where } b_j^t \sim P_{B_j} \\ &\text{IF } \theta_t^* \notin \Theta : \Theta \leftarrow \Theta \cup \{\theta_t^*\} \end{aligned} \tag{4.1}$$

where $Dim_j \downarrow b_j^t$ indicates that we have truncated $\theta(n)$ along the Dim_j dimension from the index 1 up to the index b_j^t at the iteration t. b_j^t is sampled from a distribution P_{B_j} with the support set of $B_j = \{1, 2, ..., d_j\}$ to form the ith sub-model. d_j refers to the maximum index of the jth dimension. This iterative process will be done during training and the set of n unique sub-models Θ will be built.

To illustrate the process better, let's see a simple case such as BERT$_{base}$ where we want to make a many-in-one network across the width and depth dimensions, $D = \{\text{Depth}, \text{Width}\}$. In this case, we have 12 layers and a hidden dimension size of 768. Suppose that $Depth$ corresponds to $j = 1$ and $Width$ corresponds to $j = 2$ in Eq. 4.1. For simplicity, let's use a discrete uniform distribution for sampling indices across these two dimensions. To create the first sub-model ($i = 1$), we need to sample b_1^1 uniformly from the set of natural numbers in the range of 1 to 12: $B_1 = \{1, 2, ..., 12\}$; and we need to sample b_2^1 from the range of 1 to 768: $B_2 = \{1, 2, 3, ..., 768\}$. Bear in mind that we can even choose a subset of B_1 and B_2 as the support set for sampling probability distribution. After these two samplings, we will have two truncated sets of parameters: $\theta_{Depth\downarrow b_1^1}$ and $\theta_{Width\downarrow b_2^1}$. The intersection of these two truncated parameters will give us the first sub-model: $\theta_1 = \theta_{Depth\downarrow b_1^1} \cap \theta_{Width\downarrow b_2^1}$.

Training Paradigm Regular training of neural networks concerns improving the performance of the whole model and usually this training is not aware of the performance of the sub-models. In fact, in this scenario, if we extract and deploy the sub-models of the trained large model on a target task, we would experience a significant drop in the performance of these sub-models compared with the main model. However in SortedNet, we propose a training method that allows for training sub-models together with the main model in a stochastic way. The SortedNet paradigm leads to the following benefits:

- Search-free sub-model extraction: after training, by importance sorting of sub-models the best sub-model for a given budget can be selected without the need for search.
- Anytime: Each smaller sub-model is a subset of a larger one which makes switching between different sub-models efficient. This leads to an important feature of our SortedNet which is so-called *anytime* that is a network which can produce its output at any stage of its computation.
- Memory efficient: we train a many-in-one network where sub-models are all part of a single checkpoint, which minimizes storage requirement.

For efficiency purposes, in our training, the last layer, e.g. the classification layer, is shared between all sub-models; alternatively, we can add a separate classification layer to each sub-model. For simplicity and efficiency, we chose the former i.e. use a shared classification layer.

Algorithm 2 SortedNet

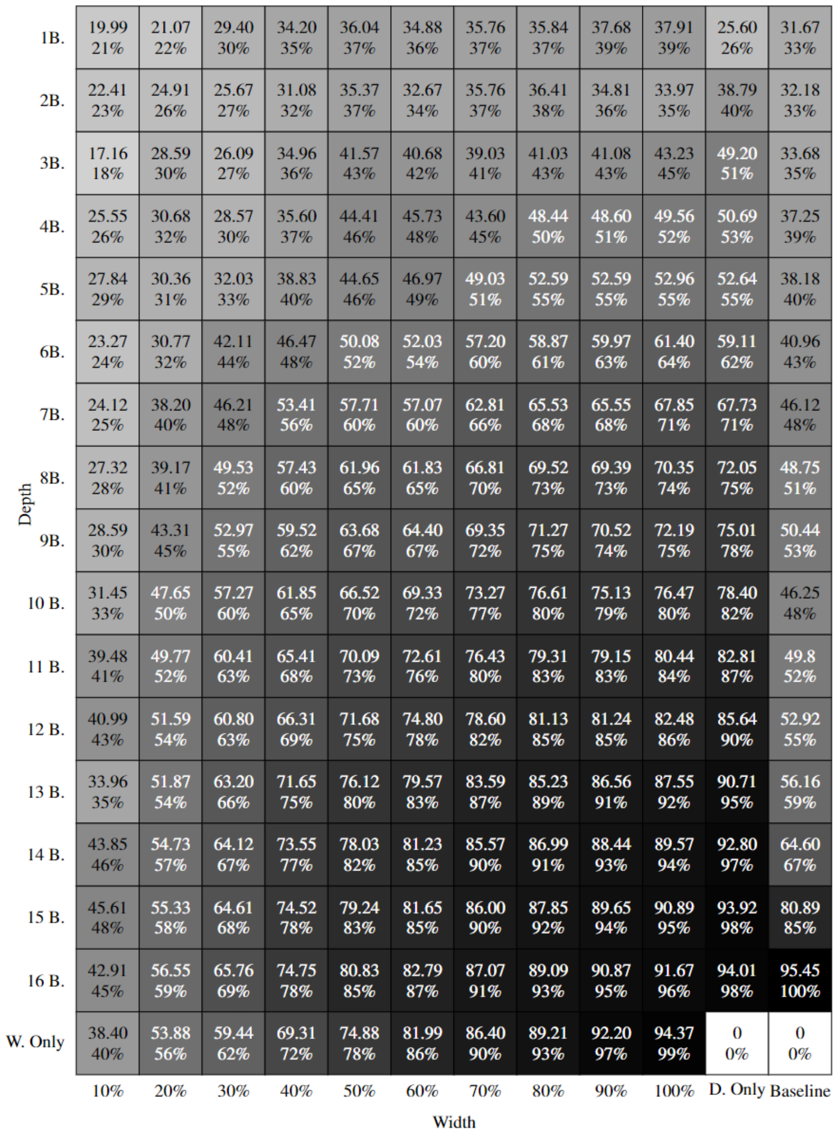
```
Require: Define the target dimensions for sorting $D = \{Dim_1, Dim_2, ..., Dim_K\}$, for each given dimension we need the set of target indices: $B_j = \{1, 2, ..., d_j\}$; the discrete sampling distribution $P_{B_j}$ for each dimension needs to be decided;
    $iter$: the number of training iterations; $g_{acc}$: Gradient Accumulation Steps; $\eta$: Learning Rate, $\mathcal{L}$: Loss Function;
    repeat
        Building Sub-models:
        // sample indices across all target dimensions
        $b_j^t \sim P_{B_j}$ for $j \in \{1, 2, ..., |D|\}$
        // truncate the parameters $\theta(n)$ along each dimension
        $\theta(t) = \cap_{j=1}^{|D|} \theta_{Dim_j \downarrow b_j^t}(n)$
        // alternatively one can provide a list of settings and randomly sample from a single distribution
        Forward & Backward:
        $\min_{\theta_t^*} \mathcal{L} \triangleq \mathrm{L}(y, f(x, \theta_t^*))$
        if $t \mod g_{acc} == 0$ then
            // update the parameters
            $\theta_t^* \leftarrow \theta_t^* - \eta \nabla_{\theta_t^*} \mathcal{L}$
        end if
    until t $< iter$
```

4.4.2 SortedNet Algorithm

In this subsection, we describe our proposed training algorithm. For training a SortedNet with n sub-models, at each iteration during training, a random index needs to be sampled from a predefined distribution: $b_j^i \sim P_{B_j}$. After finding the target sub-model θ_t^* at each iteration, we can use one of the following objectives to update the parameters of the selected sub-model:

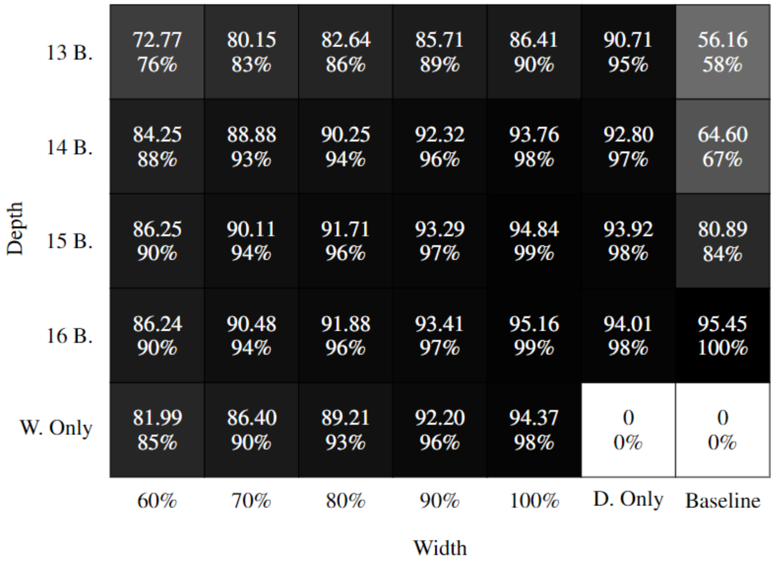
- (Stochastic Loss) Only train the selected sub-model $f(x, \theta_t^*)$:
  $\min_{\theta_t^*} \mathcal{L} \triangleq \mathrm{L}(y, f(x, \theta_t^*))$ where L is the loss function for training the model on a given task (e.g. L can be a regular cross entropy loss) and y refers to the ground-truth labels.

- (Stochastic Summation) Train the sub-model $f(x, \theta_t^*)$ and all its targeted sub-models along each dimension. Let's assume that $\Theta^{\perp}(\theta_t^*)$ is the universal set for all targeted sub-models of θ_t^*. Then the loss function can be defined as:



	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	D. Only	Baseline
1B.	19.99 21%	21.07 22%	29.40 30%	34.20 35%	36.04 37%	34.88 36%	35.76 37%	35.84 37%	37.68 39%	37.91 39%	25.60 26%	31.67 33%
2B.	22.41 23%	24.91 26%	25.67 27%	31.08 32%	35.37 37%	32.67 34%	35.76 37%	36.41 38%	34.81 36%	33.97 35%	38.79 40%	32.18 33%
3B.	17.16 18%	28.59 30%	26.09 27%	34.96 36%	41.57 43%	40.68 42%	39.03 41%	41.03 43%	41.08 43%	43.23 45%	49.20 51%	33.68 35%
4B.	25.55 26%	30.68 32%	28.57 30%	35.60 37%	44.41 46%	45.73 48%	43.60 45%	48.44 50%	48.60 51%	49.56 52%	50.69 53%	37.25 39%
5B.	27.84 29%	30.36 31%	32.03 33%	38.83 40%	44.65 46%	46.97 49%	49.03 51%	52.59 55%	52.59 55%	52.96 55%	52.64 55%	38.18 40%
6B.	23.27 24%	30.77 32%	42.11 44%	46.47 48%	50.08 52%	52.03 54%	57.20 60%	58.87 61%	59.97 63%	61.40 64%	59.11 62%	40.96 43%
7B.	24.12 25%	38.20 40%	46.21 48%	53.41 56%	57.71 60%	57.07 60%	62.81 66%	65.53 68%	65.55 68%	67.85 71%	67.73 71%	46.12 48%
8B.	27.32 28%	39.17 41%	49.53 52%	57.43 60%	61.96 65%	61.83 65%	66.81 70%	69.52 73%	69.39 73%	70.35 74%	72.05 75%	48.75 51%
9B.	28.59 30%	43.31 45%	52.97 55%	59.52 62%	63.68 67%	64.40 67%	69.35 72%	71.27 75%	70.52 74%	72.19 75%	75.01 78%	50.44 53%
10 B.	31.45 33%	47.65 50%	57.27 60%	61.85 65%	66.52 70%	69.33 72%	73.27 77%	76.61 80%	75.13 79%	76.47 80%	78.40 82%	46.25 48%
11 B.	39.48 41%	49.77 52%	60.41 63%	65.41 68%	70.09 73%	72.61 76%	76.43 80%	79.31 83%	79.15 83%	80.44 84%	82.81 87%	49.8 52%
12 B.	40.99 43%	51.59 54%	60.80 63%	66.31 69%	71.68 75%	74.80 78%	78.60 82%	81.13 85%	81.24 85%	82.48 86%	85.64 90%	52.92 55%
13 B.	33.96 35%	51.87 54%	63.20 66%	71.65 75%	76.12 80%	79.57 83%	83.59 87%	85.23 89%	86.56 91%	87.55 92%	90.71 95%	56.16 59%
14 B.	43.85 46%	54.73 57%	64.12 67%	73.55 77%	78.03 82%	81.23 85%	85.57 90%	86.99 91%	88.44 93%	89.57 94%	92.80 97%	64.60 67%
15 B.	45.61 48%	55.33 58%	64.61 68%	74.52 78%	79.24 83%	81.65 85%	86.00 90%	87.85 92%	89.65 94%	90.89 95%	93.92 98%	80.89 85%
16 B.	42.91 45%	56.55 59%	65.76 69%	74.75 78%	80.83 85%	82.79 87%	87.07 91%	89.09 93%	90.87 95%	91.67 96%	94.01 98%	95.45 100%
W. Only	38.40 40%	53.88 56%	59.44 62%	69.31 72%	74.88 78%	81.99 86%	86.40 90%	89.21 93%	92.20 97%	94.37 99%	0 0%	0 0%

Figure 4.2: CIFAR10 classification accuracy (and recovery percentage) for Sorted-Net (160 Models) and the baseline. In each cell, we reported the performance of the sub-model (top) and the relative performance of the model (in percentage) with respect to the baseline largest model performance (bottom). W. Only: Sorting only the widths, D. Only: Sorting only the depth. More black the better.



Depth	60%	70%	80%	90%	100%	D. Only	Baseline
13 B.	72.77 76%	80.15 83%	82.64 86%	85.71 89%	86.41 90%	90.71 95%	56.16 58%
14 B.	84.25 88%	88.88 93%	90.25 94%	92.32 96%	93.76 98%	92.80 97%	64.60 67%
15 B.	86.25 90%	90.11 94%	91.71 96%	93.29 97%	94.84 99%	93.92 98%	80.89 84%
16 B.	86.24 90%	90.48 94%	91.88 96%	93.41 97%	95.16 99%	94.01 98%	95.45 100%
W. Only	81.99 85%	86.40 90%	89.21 93%	92.20 96%	94.37 98%	0 0%	0 0%

Figure 4.3: CIFAR10 classification performance for the best-performing subset of sub-models trained by SortedNet from scratch. More black the better.

$$\min_{\Theta^{\perp}(\theta_t^*)} \mathcal{L} \triangleq \sum\nolimits_{\theta\in\Theta^{\perp}(\theta_t^*)} \mathrm{L}(y, f(x,\theta))$$

This way, one sub-model or a subset of sub-models are updated in each iteration. Alternatively, one can choose to train all sub-models at each iteration, which is costly on a large scale.

Table 4.2: Comparing the performance of state-of- the-art methods with Sorted-Net over CIFAR10 in terms of test accuracy.

Network	Width	FLOPs	NS-IN	LCS-p-IN	SortedNet-IN	NS-BN	LCS-p-BN (aka US)	SortedNet-BN
cpreresnet20 [31] (CIFAR10)	100%	301M	88.67±2.1	87.61±2.3	**89.14**±2.1	79.84±5.0	65.87±2.1	**85.24**±2.3
	75%	209M	87.86±1.6	85.73±2.2	**88.46**±2.1	78.59±3.4	**85.67**±1.4	85.29±3.2
	50%	97M	84.46±2.1	81.54±5.3	**85.51**±2.2	69.44±4.0	65.58±3.1	**70.98**±4.3
	25%	59M	75.42±2.2	**76.17**±1.2	75.10±2.6	10.96±3.4	**15.78**±3.5	12.59±3.2
avg.	-	-	84.10	82.76	**84.55**	59.70	58.22	**63.52**

4.4.3 Why Does SortedNet Work?

In Appendix C.3, we provide theoretical justification for parameter convergence of the sub-models in identically trained scenarios and also provide the performance bound between the trained sub-models and their similar corresponding network trained independently.

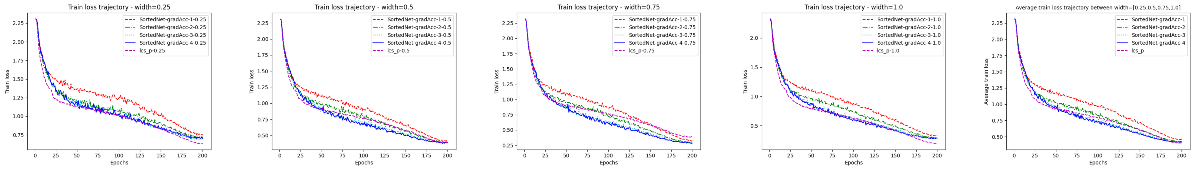


Figure 4.4: Comparing the training loss trajectory of SortedNet on CIFAR10 for different gradient accumulation values with LCS_p. Each subfigure demonstrates the results in different widths. The rightmost subfigure reports the average across the widths. The underlying network (cPreResNet20) and hyperparameters are fixed.

Convergence Suppose $\hat{f}$ is a sub-model of a larger network, and f is an identical model architecture trained independently. We aim to understand the relationship between the parameters of these two networks, θ for $\hat{f}$ and ϕ for f, as they are trained under identical conditions. Assuming that the gradients of the loss functions for $\hat{f}$ and f, are L-Lipschitz continuous, and the learning rate is η, we show that

$$\|\theta_{t+1} - \phi_{t+1}\| \leq (1 + \eta L)\|\theta_t - \phi_t\|. \tag{4.2}$$

This indicates that the difference in the parameters of $\hat{f}$ and f is governed by the Lipschitz constant L and the learning rate η, suggesting that the parameters should remain close throughout the training process, especially when the difference between the gradients of the loss functions of the two networks is negligible.

Performance Bound Moreover, we would like to find a performance bound between a trained sub-model (with optimized parameters θ^*) and its corresponding individual model (with optimized parameters ϕ^*). Let's assume that $\phi^* = \theta^* + \Delta\theta$. We show in Appendix C.3 that the deviation $\Delta f = f(x; \phi^*) - \hat{f}(x; \theta^*)$ in the function value from its optimal value due to a parameter perturbation $\Delta\theta$ is bounded by $\frac{1}{2}L\|\Delta\theta\|^2$ under the assumption of L-Lipschitz continuity of the gradient.

$$\Delta f \approx \frac{1}{2}\Delta\theta^T H(x; \theta^*)\Delta\theta \leq \frac{1}{2}L\|\Delta\theta\|^2 \tag{4.3}$$

This result implies that the function value's deviation grows at most quadratically with the size of the parameter perturbation.

4.5 Experiments

In this section, we conduct a set of experiments to show the effectiveness and importance of our solution. The details of the hyper-parameters for each experiment can be found in Appendix C.4.2.

Table 4.3: A comparison of the performance of different sub-models with and without the SortedNet. The model's performance will improve if we have more budgets and calculate the representation of deeper layers.

	Acc.	Acc.	F1	Mathews Corr.	Acc.	Acc.	Acc.	Pearson		
Model	**MNLI**	**SST-2**	**MRPC**	**CoLA**	**QNLI**	**QQP**	**RTE**	**STS-B**	**AVG**	**AVG w/o ours**
Sorted-RoBERTa (1L)	60.07	70.76	81.22	0.00	59.64	77.80	47.65	9.36	**50.81**	40.33
Sorted-RoBERTa (2L)	71.98	80.28	81.22	0.00	81.49	87.09	47.29	70.37	**64.97**	40.86
Sorted-RoBERTa (4L)	76.74	80.50	81.22	0.00	85.21	88.82	46.93	75.07	**66.81**	41.06
Sorted-RoBERTa (4L)	79.13	84.75	81.22	44.51	86.60	90.11	49.10	84.94	**75.04**	42.95
Sorted-RoBERTa (5L)	81.14	89.91	81.22	48.41	87.88	90.86	55.96	88.22	**77.95**	43.80
Sorted-RoBERTa (6L)	82.21	92.09	86.67	53.41	88.83	91.12	67.87	89.09	**81.41**	46.13
Sorted-RoBERTa (7L)	82.99	92.78	89.13	56.42	89.29	**91.29**	73.29	89.58	**83.10**	44.80
Sorted-RoBERTa (8L)	83.33	93.23	89.78	57.22	89.40	91.29	75.09	89.67	**83.63**	55.17
Sorted-RoBERTa (9L)	83.39	92.66	89.66	58.69	89.40	91.25	**77.26**	89.72	**84.00**	61.36
Sorted-RoBERTa (10L)	**87.42**	93.12	**91.64**	**61.21**	**91.87**	91.19	74.01	**89.74**	**85.02**	54.30
Sorted-RoBERTa (11L)	87.34	**93.35**	91.45	60.72	91.74	91.17	74.01	89.72	**84.94**	77.48
Sorted-RoBERTa (12L)	83.35	92.89	90.81	59.20	89.44	91.28	76.53	89.77	84.16	**86.13**
avg.	79.26	87.93	86.09	41.25	85.50	89.45	64.26	79.61	**76.67**	52.86

4.5.1 Is SortedNet Scalable?

To show that our proposed method is scalable, we designed an experiment that tries to train 160 different models across multiple dimensions (width and depth) all at once. As a baseline, we trained the largest network (a MobileNetV2), and reported the best performance of the model. Because the performance of the model was poor for all the other sub-models (less than 12%), we trained the classifier layer for 5 more epochs before evaluating each sub-model for the baseline and reported the best performance. As the results suggests in Figure 4.2, our method was able to capture the maximum performance for many of these sub-models in a zero-shot manner. In each cell, we reported the performance of the sub-model on top and the recovery percentage of the model with respect to the largest model (in this example, 95.45). Despite sharing the weights across all models, sharing the classifier and zero-shot evaluation, the proposed method preserved up to 96% of the performance of the largest model which is highly encouraging. Further training of the classifier for our proposed method will lead to even better performance as shown in appendix C.4.4 (between ∼2 to 15% improvement for different sub-models). In addition, we also tried to sort the depth and width using proposed method individually, which is reported in the Figure 4.2 as D. Only, and W. Only, respectively. Across width, SortedNet successfully preserved up to 99% of the largest network's performance.

4.5.2 Can we find the best sub-models using SortedNet?

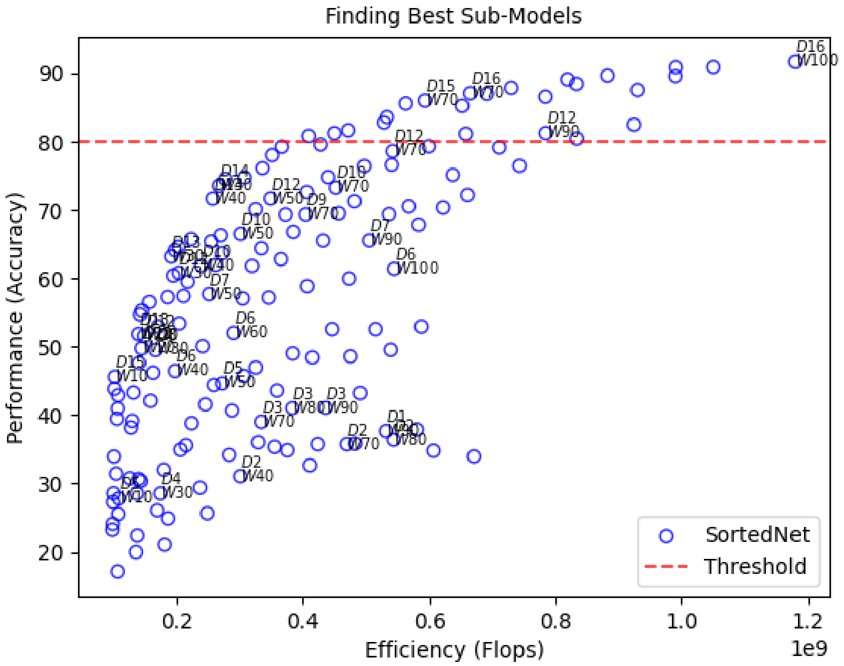


Figure 4.5: Finding best sub-models automatically using a desired threshold bar to eliminate the worst performing models.

As shown in Figure 4.3, based on the performance of the models in the previous experiment shown in Figure 4.2, we selected a subset of best-performing networks (width $>$ 60% and depth $>$ 13 blocks), and retrained the network from scratch using SortedNet to show the success rate of our proposed method. As shown, SortedNet successfully preserved up to 99% of the performance of the ordinary training of the largest network.

We can also make this selection process fully automated by sorting the performance of all sub-models after evaluation and filtering out a subset of best-performing models that perform better than a desired threshold. As it can be seen in Figure 4.5, there is a set of sub-models which perform better than 80%. To better understand the pattern, we annotated some of the points using "^D_W" as template which shows for each model the corresponding width and depth.

4.5.3 Can we generalize SortedNet?

In another experiment, as shown in Table 4.2, we demonstrate the superiority of our stochastic approach compared to the state-of-the-art methods such as LCS (shown as LCS_p in the table) [70], Slimmable Neural Network (NS) [119], and Universally Slimmable Networks (US) [118]. To make the comparisons fair, we equalized the number of gradient updates for all models. We also tried to remove the impact of architecture design such as the choice of the normalization layers. Therefore, we tried to compare methods by different layer normalization techniques such as BatchNorm [44] and InstanceNorm [102]. In addition, we ensure that complementary methods such as Knowledge Distillation have no impact on the results as these methods can be applied and improve the results independent of the method. As shown in the table, SortedNet demonstrates a superior average performance compared to other methods, indicating its generalization across various settings such as different norms. It is worth noting that we realized the unexpected nature of the LCS-p-BN results in Table 4.2. However, these results are in line with the original LCS paper's observations [70] (see Figure 3 of the LCS paper). The LCS authors [70] also hypothesized that this drop caused by inaccurate batch norm statistics. To address this, they suggested an architectural adjustment to GroupNorm. Our SortedNet approach, on the other hand, remains unaffected by this issue, thus requiring no such modifications.

4.5.4 Extending Sorted Net to Pre-trained Language Models

In this experiment, the goal is to apply SortedNet for a pre-trained transformer model and evaluate the performance on the GLUE benchmark [108]. As the baseline, we chose RoBERTa [61] to demonstrate the flexibility of our algorithm. In Table 4.3, we sorted all the layers of RoBERTa-base. As the results demonstrate, our proposed method in average perform better than the baseline by a significant margin ($\sim$ 23%). However, the largest model has a small drop in performance (less than 2%). It is interesting that the transformer architecture can preserve the performance of sub-models up to some extent without additional training. However, our algorithm improves the performance of these sub-models between 10 to 40% approximately. A more complex setting (sorting across Bert models), has been investigated in appendix C.4.5.

Table 4.4: Speed-up in inference time on GSM8K benchmark by utilizing Speculative Decoding and Adaptive Early-Exit Techniques over SortedNet models.

SortedNet Efficient Decoding

	Stochastic Loss			**Summation Loss**		
			Auto-regressive Decoding			
Model	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**
Layer 40 (full)	96.41	23.95	-	86.10	25.24	-
			Speculative Decoding			
Draft Model	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**
Layer 12	58.86 (1.63×)	22.28	0.35	63.68 (1.35×)	16.52	0.40
Layer 16	60.25 (1.60×)	23.42	0.24	63.89 (1.34×)	20.92	0.30
Layer 20	65.92 (1.46×)	25.09	0.16	69.61 (1.23×)	21.98	0.22
			Instance-Aware Dynamic Inference			
Model	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**
Layer 12:40	46.02 (2.09×)	20.69	-	50.89 (1.69×)	24.63	-
			Adaptive Speculative Decoding			
Layer 12:40	47.93 (2.01×)	23.27	0.06	65.19 (1.32×)	24.79	0.07
Layer 12:24	48.66 (1.98×)	24.33	0.14	63.58 (1.35×)	25.17	0.19

4.5.5 Extending SortedNet to Decoder-based Large Language Models

To further show the scalability and generalizablity of SortedNet in more practical scenarios, we fine-tuned a LLaMA-13b [99] on GSM8K [18], which is one of the challenging mathematical reasoning tasks. We chose the first 12, 16, 20, 24, 28, 32, 36, and 40 layers of LLaMA to build our submodels. To equalize the number of updates, we trained the model based on our stochastic loss 8 times more than the summation loss, as we have 8 models and each forward pass in stochastic loss is 1/8 of the summation loss. In table 5.2, we reported the performance of a subset of submodels and speedup gain that one can achieve using different sampling techinques such as Autoregressive decoding, speculative decoding [57], and Instance-Aware Dynamic Inference which is a confidence-based early-exiting of the Sorted Models. As shown, combining SortedNet and speculative decoding can improve the time per token efficiency up to 2.09 times faster than using auto-regressive for the full size model.

4.5.6 Ablation Study

Convergence (Training Time) Analysis Being sorted and randomly selecting one sub-model at the time from a predefined set of the sub-models empowers SortedNet with a higher convergence rate and a faster training time. Figure 4.4 empirically certifies this claim and compares the

Table 4.5: Effect of gradient accumulation on SortedNet-IN performance while fixing the number of parameter updates. The underlying network and dataset are cPreResNet20 and CIFAR10, respectively. Numb. Updates refers to the number of calls to optimize.step()

Grad. Accum.	Num. Updates	Epochs	Accuracy @ Width				Avg.
			100%	75%	50%	25%	
$g_{acc} = 1$	200	200	84.94	84.92	82.54	71.03	80.85
$g_{acc} = 2$	200	400	86.69	86.68	84.40	72.36	82.53
$g_{acc} = 3$	200	600	87.37	87.50	84.57	73.00	83.11
$g_{acc} = 4$	200	800	87.93	87.40	84.27	76.23	83.95

training convergence of SortedNet against LCP_p, which, to the best of our knowledge, LCP_p stands as the most recent state-of-the-art method. As LCS_p uses summation loss over four sub-models in every training steps and to have a fair comparison, we therefore report the performance of SortedNet in different values of gradient accumulation (g_{acc}), where $g_{acc} = 4$ provides a fair comparison with LCS_p. As shown in the figure, SortedNet with $g_{acc} = 4$ converges either faster or competitive across different sub-models. Moreover, SortedNet does not require any for-loop in its implementation; thus tailoring parallel computation and resulting in faster running time. We empirically investigate this feature and found that in the same settings, SortedNet runs at least one third faster than LCS_p (details in Appendix C.4.3).

The impact of gradient accumulation The goal of this experiment is to examine the impact of gradient accumulation (g_{acc}) on the performance of SortedNet within an equal number of parameters updates. Table 4.5 presents the results obtained in terms of accuracies for 4 different gradient accumulation values. To ensure an equal number of updates, the maximum number of epochs is adjusted for each scenario, e.g. $g_{acc} = k$ receives k times more epochs than $g_{acc} = 1$. As the results explains, increasing gradient accumulation values results in a higher performance for SortedNet. This observation can be attributed to the increase in training stochasticity when gradient accumulation is raised. Consequently, each sub-model in SortedNet contributes more equally to the updating of weight parameters, leading to a faster convergence rate. More details are provided in Appendix C.4.1.

In addition, we highlighted the details of each experiment hyperparameters in appendix C.4.2 and further analysis has been provided in appendix C.4.6 to better understand the behavior of sortedNet methodology.

Conclusion

In summary, this paper proposes a new approach for training dynamic neural networks that leverages the modularity of deep neural networks to efficiently switch between sub-models during inference. Our method sorts sub-models based on their computation/accuracy and trains them using an efficient updating scheme that randomly samples sub-models while accumulating gradients. The stochastic nature of our proposed method is helping our algorithm to generalize better and avoid greedy choices to robustly optimize many networks at once. We demonstrate through extensive experiments that our method outperforms previous dynamic training methods and yields more accurate sub-models across various architectures and tasks. The sorted architecture of the dynamic model proposed in this work aligns with sample efficient inference by allowing easier samples to exit the inference process at intermediate layers. Exploring this direction could be an interesting area for future work. One other research direction which we will leave for future is to keep the gap between submodels of the SortedNet and the corresponding individual models trained regularly, as lowest as possible during each step.

Limitations

It is good to note that our proposed method might be sensitive to the randomness as the chosen trajectory at the moment is random uniform. Further research is necessary to investigate the effect of choosing more optimal strategies for choosing the next model at each iteration. One can further analyse the performance of progressively training submodels and ensure that the models converged properly at the ultimate training steps.

Chapter 5

Sorted LLaMA: Unlocking the Potential of Intermediate Layers of Large Language Models for Dynamic Inference Using Sorted Fine-Tuning (SoFT)

5.1 Abstract

Large language models (LLMs) have revolutionized natural language processing (NLP) by excelling at understanding and generating human-like text. However, their widespread deployment can be prohibitively expensive. SortedNet is a recent training technique for enabling dynamic inference by leveraging the modularity in networks and sorting sub-models based on computation/accuracy in a nested manner. We extend SortedNet to generative NLP tasks, making large language models dynamic without any Pre-Training and by only replacing Standard Fine-Tuning (SFT) with Sorted Fine-Tuning (SoFT). Our approach boosts model efficiency, eliminating the need for multiple models for various scenarios during inference. We show that this approach can unlock the power of intermediate layers of transformers in generating the target output. Our sub-models remain integral components of the original model, minimizing storage requirements and transition costs between different computational/latency budgets. The efficacy of our proposed method was demonstrated by applying it to tune LLaMA 2 13B on the Stanford Alpaca dataset for instruction following and TriviaQA for closed-book question answering. Our results show the superior performance of sub-models in comparison to Standard Fine-Tuning and SFT+ICT (Early-Exit), all achieved with efficient tuning and without additional memory usage during in-

ference.

5.2 Introduction

Large language models are revolutionizing the way we interact with information in today's world [35, 9, 75, 89]. New models are continually emerging, demonstrating their capabilities in understanding and, more importantly, in generating human-like text. Notably, models such as ChatGPT, LLaMA 2 70B [100], and Falcon 180B [2] have had a profound impact on the applicability of large language models (LLMs). However, deploying these expansive language models can become prohibitively expensive.

What distinguishes this new era of ChatGPT-like models is their ability to perform an extraordinarily wide array of tasks in natural language processing (NLP), reasoning, and more, all through behavior cloning [114, 112]. In fact, a single model can leverage the strong contextual learning ability offered by Standard Fine-Tuning to address numerous tasks, spanning from language comprehension to complex reasoning. While this unified usage simplifies the deployment of these models as general assistants, it remains highly inefficient. Enabling dynamic inference, where the computational resources allocated to a given query vary at inference time, can significantly enhance the practicality of employing such models in real-time scenarios. This enables the use of smaller models when the budget is limited or latency is critical. It is important to note that dynamic inference strategies for large models with a substantial number of parameters should not require loading different models during inference.

Previous research has explored methods for training dynamic models capable of adapting to evolving resource constraints [11, 37, 117, 26]. However, existing approaches often rely on complex training procedures or necessitate modifications to the original model architecture. SortedNet [105] introduces a novel approach to training deep neural networks that leverages the inherent modularity of these networks to construct sub-models with varying computational loads. This method sorts sub-models hierarchically based on their computation/accuracy characteristics, facilitating efficient deployment during inference. Furthermore, it employs an efficient updating scheme combining random sub-model sampling with gradient accumulation to minimize the training cost. Consequently, with a single round of training, numerous models can be obtained within a single model.

While the SortedNet approach has primarily been applied to vision and language understanding tasks, given the significant impact of generative language models in today's AI landscape, the efficacy of this method for generative tasks in NLP is of considerable interest. In fact, being able to make a large language model dynamic without the need for Pre-Training and only at the cost of a round of Standard Fine-Tuning can open doors to efficient inference of these models without incurring additional expenses associated with common model compression methods like knowledge distillation and pruning, among others. Moreover, since all the resultant models are components of the original model, the storage requirements and the cost associated with transitioning between different computation demands become minimal. Otherwise, managing multiple models for various scenarios during inference becomes impractical.

In this study, we challenge the conventional approach of relying solely on the last layer's contextual embeddings and use Sorted Fine-Tuning (SoFT) in place of Standard Fine-Tuning to enhance the performance of these models across multiple layers. By doing so, we aim to provide new insights into the efficiency and effectiveness of middle layers in producing high-quality results for specific downstream tasks. Our proposed approach can potentially optimize these sub-models in addition to the main model, ultimately enhancing their overall performance. In this paper, we seek to answer the following questions through systematic evaluation:

i) Do the intermediate layers resulting from Standard Fine-Tuning of a large language model generate accurate and meaningful outputs? ii) Does Standard Fine-Tuning exhibit a sorted behavior, meaning that later layers produce more accurate and meaningful results than earlier layers? If so, to what extent? iii) How can we enhance this sorted behavior with minimal cost?

To answer these questions, we employ LLaMA 2 13B and perform both Standard Fine-Tuning (SFT) and Sorted Fine-Tuning (SoFT) on the Stanford Alpaca [98] and TriviaQA [47] datasets. For Sorted Fine-Tuning, we target 8 sub-models and share the LLM head among them to ensure cost parity. We utilize the PandaLM benchmark [110] to assess the performance of the sub-models on Alpaca dataset. Our findings demonstrate the superior performance of SoFT in comparison to SFT and even to memory-demanding methods like Early Exit [117]. The contributions of this paper can be summarized as follows:

- Extending the SortedNet method for tuning auto-regressive language models for generative tasks by sharing a single LLM head layer among sub-models.
- Generating 8 nested sub-models, ranging from 12 to 40 layers, from LLaMA2 13B by applying Sorted Fine-Tuning on the Stanford Alpaca dataset and TriviaQA benchmarks and at a cost equivalent to Standard Fine-Tuning.
- Evaluating the performance of the sub-models of a LLaMA 2 and demonstrating the effectiveness of SoFT in enhancing the ability of intermediate layers for text generation and question answering through extensive evaluation.

5.3 Related Work

This section briefly introduces the most relevant papers to our work.

Many-in-One Models Deep neural networks (DNNs) are often overparameterized, motivating researchers to explore ways to use the parameters of the models more efficiently. More number of parameters lead to higher costs of deployment for neural networks. Moreover, in practice, these overparametrized DNNs are expected to accommodate customers with varying requirements and computational resources. To address these diverse demands, one can think of training models of different sizes, which can be prohibitively costly (in terms of training and memory), or another alternative is to train many-in-one networks [11]. Many-in-one solutions aim to train a network along with some of its sub-networks simultaneously for specific tasks. For example, we can consider the *Early-Exit* method [117], wherein a prediction head is fine-tuned on top of specific intermediate layers within a network. Another approach is *Layer Drop* [26], which trains a network in any depth by randomly dropping the layers during training. While both Early-Exit and Layer Drop are simple solutions, they are not state-of-the-art in terms of performance. In Early-Exit, we only train the output prediction layer on top of each intermediate layer, and this layer might not have enough capacity to retain a good performance. Layer Drop, conversely, suffers from the abundant number of possible sub-models in training, which makes the training process exhaustive and sub-optimal. Furthermore, this approach requires tuning the extent of dropping layers during training. This additional hyper-parameter, layer drop rate during training determines the best size and setting of the model at the inference time. Deviating from the training drop rate at the inference time can result in a significant drop in performance.

[11] in *Once for All (OFA)* proposed an alternative solution to neural architecture search (NAS). OFA requires training the model and all possible sub-models in an arbitrary progressive way followed by a separate search phase. Dyna-BERT [37] is another work that targets training Dynamic pre-trained many-in-one BERT models in two stages: first, distilling from the main network to the width adaptive networks and then distilling from the width adaptive networks to depth adaptive networks. Both width adaptive and depth adaptive networks have a limited pre-defined set of width and depth for the sub-models. While both OFA and DynaBERT have shown successful results, their solutions are hardly applicable to multi-billion-parameter LLMs because of their complicated multi-stage training process and their search and knowledge distillation requirements. SortedNet [105] is a recent method that forms and trains sub-models of a network in a sorted manner while not requiring any search during training or inference. SortedNet has shown superior performance compared to other previously mentioned methods in terms of simplicity, performance, scalability, and generalization. Considering these benefits, we target deploying the SortedNet training algorithm for developing many-in-one LLMs.

Many-in-One Large Language Models (LLMs) Large language models have recently gained significant attention in the literature [99, 9, 71, 17, 72]. In practice, these LLMs serve users with different tasks, expectations, and computational budget requirements [94]. There are two types of adaptation approaches to make LLMs suitable for customer requirements: first is the so-called parameter efficient tuning (PEFT), and second is model compression. In PEFT, the core backbone model remains the same, and we just update much smaller adapter parameters (e.g. LoRA [42], KRONA [24], Adapter [38, 77], DyLoRA [104], Ladder Side-Tuning [95]) and Compacter [48]. In model compression, the larger model is compressed using any model compression solutions such as knowledge distillation [33, 41, 115], pruning [4], and quantization [78, 19], a good related survey can be found in [122]. Even though PEFT solutions are pretty popular with LLMs, they do not provide dynamic-size LLMs. Model compression solutions can provide models with different sizes, but they need to train each compressed model separately, and they are not many-in-one models.

To the best of our knowledge, this work is the first attempt to obtain many-in-one generative LLMs by applying the Sorted Fine-Tuning to the LLaMA 13B model. Considering the benefits of many-in-one networks and the growing application of LLMs, we are hoping that this will help the community to build more efficient large language models that can be deployed adaptive during inference using methods such as SoFT.

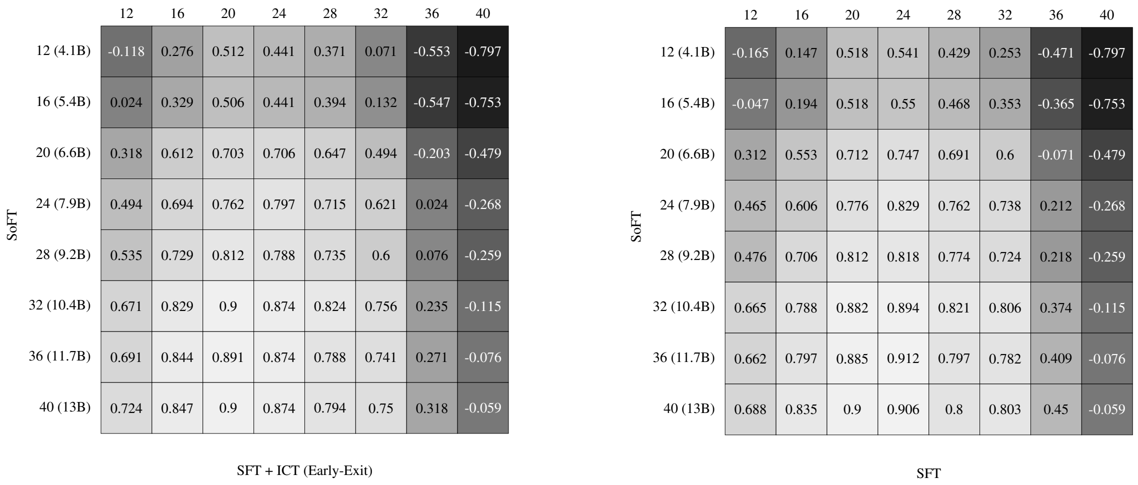
5.4 Methodology

This paper focuses on making generative LLMs many-in-one by unlocking the potential of intermediate layers through the SortedNet approach [105].

Let's consider a language model $f(x; \theta)$ with the parameters θ and the input x. The following is the sorted training procedure:

Forming Sub-Networks First, we need to form the sub-networks of the LLM. For the sake of simplicity and without loss of generality, we focus on the depth-wise sub-networks. Supposed that the sub-network $f_n(x; \theta_n)$ refers to the first n layers of $f(x; \theta)$. In this paper, the language model is considered to be LLaMA2 13B. Since LLaMA2 comprises 40 layers, we define the sub-networks as $n \in \mathbf{B} = \{12, 16, 20, 24, 28, 32, 36, 40\}$.

Calculating the Output of Sub-Networks Each sub-model's output will be predicted using the shared output prediction head from the last layer (original network). Remember that in



	12	16	20	24	28	32	36	40
12 (4.1B)	-0.118	0.276	0.512	0.441	0.371	0.071	-0.553	-0.797
16 (5.4B)	0.024	0.329	0.506	0.441	0.394	0.132	-0.547	-0.753
20 (6.6B)	0.318	0.612	0.703	0.706	0.647	0.494	-0.203	-0.479
24 (7.9B)	0.494	0.694	0.762	0.797	0.715	0.621	0.024	-0.268
28 (9.2B)	0.535	0.729	0.812	0.788	0.735	0.6	0.076	-0.259
32 (10.4B)	0.671	0.829	0.9	0.874	0.824	0.756	0.235	-0.115
36 (11.7B)	0.691	0.844	0.891	0.874	0.788	0.741	0.271	-0.076
40 (13B)	0.724	0.847	0.9	0.874	0.794	0.75	0.318	-0.059

	12	16	20	24	28	32	36	40
12 (4.1B)	-0.165	0.147	0.518	0.541	0.429	0.253	-0.471	-0.797
16 (5.4B)	-0.047	0.194	0.518	0.55	0.468	0.353	-0.365	-0.753
20 (6.6B)	0.312	0.553	0.712	0.747	0.691	0.6	-0.071	-0.479
24 (7.9B)	0.465	0.606	0.776	0.829	0.762	0.738	0.212	-0.268
28 (9.2B)	0.476	0.706	0.812	0.818	0.774	0.724	0.218	-0.259
32 (10.4B)	0.665	0.788	0.882	0.894	0.821	0.806	0.374	-0.115
36 (11.7B)	0.662	0.797	0.885	0.912	0.797	0.782	0.409	-0.076
40 (13B)	0.688	0.835	0.9	0.906	0.8	0.803	0.45	-0.059

Figure 5.1: SoFT vs. SFT + ICT (Early-Exit) (Left) and SoFT vs. SFT (Right). Note that for our SoFT method, the output prediction layer is shared between all sub-models whereas, for Early-Exit, a separate prediction head is learned per sub-model, making inference inefficient. Both SoFT and SFT had equivalent training time (2 Epochs) in this experiment. The number in each cell is calculated by considering wins as the times SoFT sub-models (rows) were preferred, losses as the times SFT sub-models (columns) were preferred and ties when non of them were preferred (Equation 5.2). Algorithm performance is correlated to cell whiteness: white is better, zero is on-par, dark is worse.

the LLaMA model, there is an RMSNorm layer [121] before the output prediction head. This RMSNorm is added before the shared prediction head of every sub-model. This normalization may be an important factor that helps Sorted LLaMA to generalize better for all sub-models.

Objective Function Let $L_n(x;\theta_n)$ be the loss for the nth sub-model for input batch x. To train the network, we define the loss as the summation of the losses of all these sub-models:

$$\mathcal{L} = \frac{\sum_{n \in \mathbf{B}} L_n(x;\theta_n)}{|\mathbf{B}|} \tag{5.1}$$

For the experiments conducted in the paper, $|\mathbf{B}| = 8$. Note that these sub-models have shared parameters through a nested style i.e. $\theta_1 \subset \theta_2 ... \subset \theta_n$.

Training Dataset We utilized the Stanford Alpaca dataset [98], which includes demonstrations of 52K instruction-following examples. We also used TriviaQA open-domain QA benchmark [47] including 110K closed-book question-answer pairs.

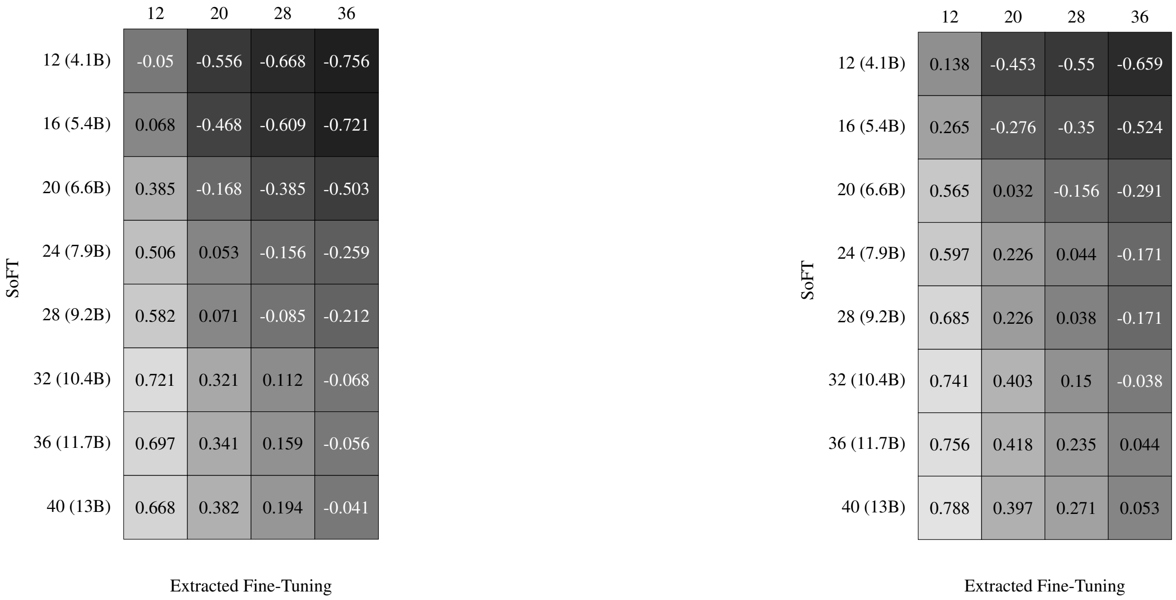


Figure 5.2: SoFT vs. Extracted Fine-Tuning. The left figure shows an equal training time setup (2 epochs), and the figure on the right considers two extra training epochs for SoFT.

Evaluation In this paper, in addition to embedding the last layer, we evaluate the quality of the embeddings of intermediate outputs spanning from block 1 to n. PandaLM benchmark [110] compares the output of different sub-models. PandaLM deploys a large language model (Fine-Tuned LLaMA 7b) to judge the quality of generated text from two sources. PandaLM provides a validation set consisting of 170 instructions[1], to evaluate target models for instruction-following tasks. To ensure that the order of the models' responses does not influence the judgment of the PandaLM evaluator, we reported an average score under both the Model 1 first and the Model 2 first scenarios. The output of the PandaLM evaluation is the number of wins, denoted as W, the number of losses, denoted as L, and the number of ties, denoted as T, in the validation set. The final reported score has been calculated using the following formula:

$$Score = \frac{(W - L)}{T + W + L} \tag{5.2}$$

The final score is a number between -1 and 1, in which 1 represents a strong win rate and -1 means a poor performance of the model.

We used accuracy (exact match) as the evaluation metric for the TriviaQA benchmark.

---

[1] github.com/WeOpenML/PandaLM/blob/main/data/testset-inference-v1.json

Baseline The primary objective of the LLM in this paper is to follow the provided instructions by a query. Therefore, following the setup of Alpaca [98], we fine-tuned LLaMA2 13B on the Stanford Alpaca Dataset with two setups: (1) Regular Standard Fine-Tuning (SFT) as the baseline, focusing only on the training of the last layer of the network as the common practice in the literature; (2) Sorted Fine-Tuning (SoFT), calculating loss for multiple outputs from layer 12 to layer 40 (last layer) with four intervals, and training multiple models simultaneously as explained in the previous section.

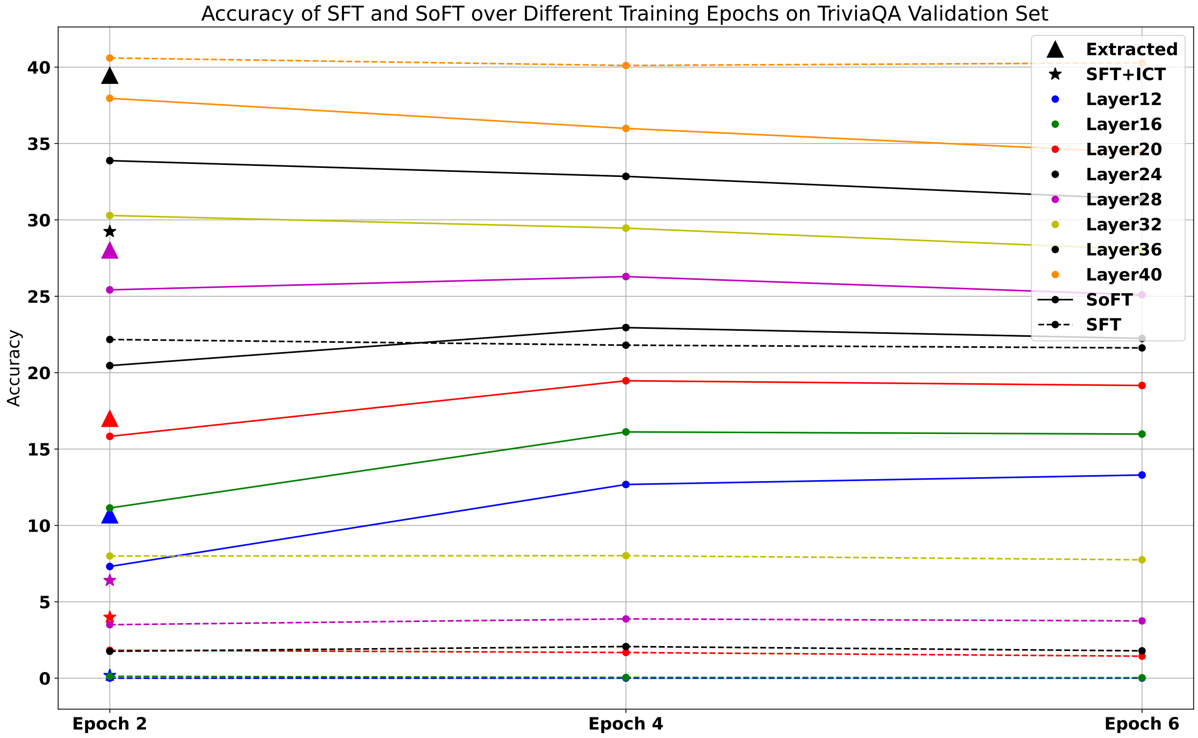
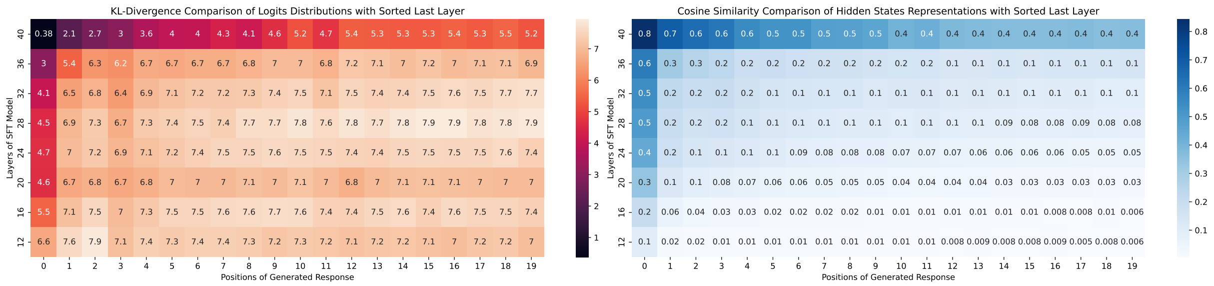


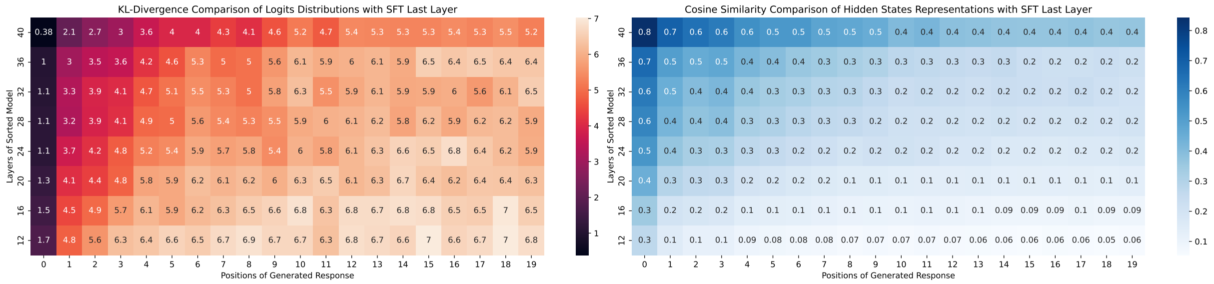
Figure 5.3: The results of TriviaQA. We reported case-sensitive exact match accuracy as the main metric. SFT+ICT and Extracted Fine-Tuned results can be found in Epoch 2, as we found Epoch 2 checkpoint saturated for the original SFT experiment (main LLaMA2 13b model with 40 layers).

5.5 Experiments

This section delves into the experiments' specifics and the analysis provided to understand better the effect of Sorted Fine-Tuning over the performance of a large language model like LLaMA2



(a) SortedLLaMA Last Layer - LLaMA sub-models

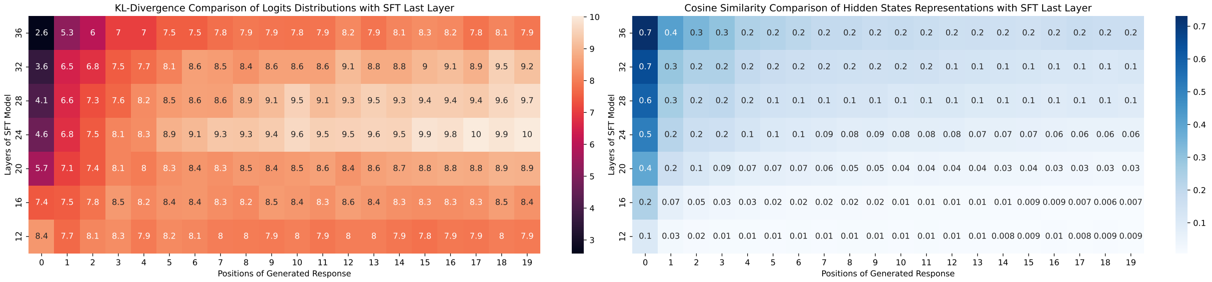


(b) SortedLLaMA sub-models - LLaMA Last Layer

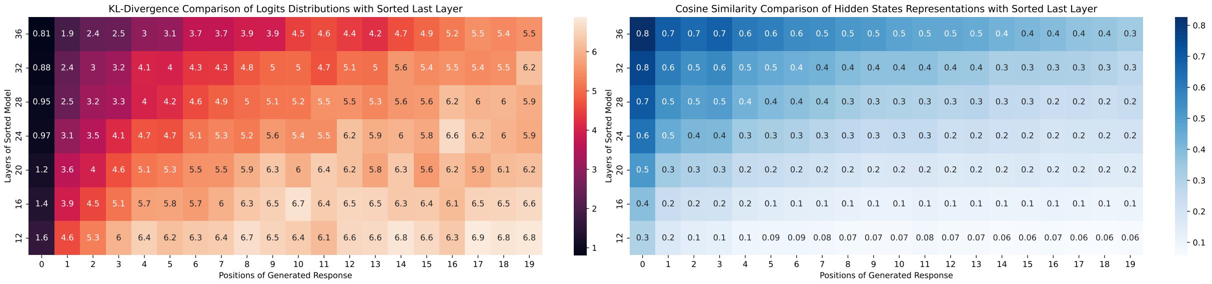
Figure 5.4: An inter-model comparison of sub-models based on output logits and hidden state cosine similarity. The numbers are average of all 170 samples in the PandaLM validation set. The similarity is stronger if the cell is darker.

[100]. The details of the experimental setup used for these experiments are available in the appendix D.0.1. Before diving into results, we are going to define certain notations that we used for different setups in our experiments:

- **SoFT/SFT**: We first train the model with SoFT or SFT paradigms and use the sub-models after training without any further training of the language model head for intermediate layers.
- **SFT+Intermediate Classifier Tuning (ICT)**: We first train the model with SFT paradigm and then further fine-tune the language model head exclusively for each sub-model while keeping their weights frozen. The SFT+ICT is also known as Early-Exit [117] in the literature.
- **Extracted Fine-Tuning**: We extract the sub-models from the learned weights of the pre-trained original model and train each sub-model separately.



(a) LLaMA sub-models vs LLaMA Last Layer



(b) SortedLLaMA sub-models - SortedLLaMA Last Layer

Figure 5.5: An intra-model comparison of sub-models based on output logits and hidden state cosine similarity. The similarity is stronger if the cell is darker.

5.5.1 What is the effect of sorting information across layers of a generative model?

As mentioned before, we generated responses for all the layers $n \in \mathrm{B}$ for both SFT and SoFT-based trained models. Then, we conducted a pair-wise comparison between all the sub-models in the two trained models using the PandaLM evaluator. As the results suggest in Figure 5.1, sorted training significantly unlocks the potential of intermediate layers in generating the desired output. Some generated examples can be found in Table 5.1.

Sorted LLaMA (aka SoFT) is outperforming regular fine-tuning (SFT) in nearly all layer comparisons by a meaningful margin, as shown through automated evaluation in Figure 5.1.

It might be noted that the Layer 12 performance of SFT is slightly better compared to Layer 12 of Sorted LLaMA. We argue this is happening because the outputs of early layers in SFT are mostly gibberish (see Table 5.1 as an example), and the PandaLM evaluator has not been trained on such data. Hence the automatic evaluation results for this layer are not meaningful. To further investigate the reason behind the results for early sub-models, we conducted human evaluation on 6 cells of two tables in Figure 5.1 (Layer 12 of SFT and SFT+ICT vs Layers 12,16, and 20 SoFT) to verify our claim. We observed that SoFT early sub-models could significantly outperform sub-model layer 12 of both SFT and SFT+ICT models, proving the negative impact of gibberish text on PandaLM evaluator performance. As we go to higher layers in SFT, the generated text becomes meaningful, which makes the comparison with the Sorted LLaMA layer counterpart more reasonable.

Moreover, to improve SFT results, inspired by Early-Exit [117], we also tried the scenario in which a separate classifier head is dedicated to all sub-models of SFT. This method has been introduced in the notation section as SFT+ICT. These classification heads have been trained an additional epoch after SFT tuning while keeping the base model frozen. Note that this setting suffers from significant memory overhead during tuning and inference compared to our SoFT method. In fact, the extra number of parameters for SFT+ICT (Early Exit) is $|B| - 1 \times D \times V$, where $|B|$ is the number of sub-models, D is the hidden size of the model, and V is the vocabulary size. For LLaMA 2 13B, this is equivalent to 1B extra parameters.

The results of comparing sorted with the early exit are shown in figure 5.1 (Left). Despite having far more parameters, SFT+ICT (Early-Exit) underperforms our sorted tuning for most sub-models. According to the results, the sub-model in Sorted LLaMA with 36 layers performs almost as well as regular fine-tuning of the full-size model. This showcases the impressive ability of our proposed paradigm to generate powerful, small sub-models that perform similarly to the original model. Another experiment that has been conducted in appendix D.0.3, further investigated the impact of longer training time for SoFT. The results show that our model was still under-trained, and we could observe a significant improvement in Sorted LLaMA performance with longer training time.

Moreover, we compared the performance of Sorted LLaMA sub-models with the actual capacity of these models by fine-tuning the sub-models separately and reporting the results in both equal training time and more training time for SoFT. We extracted 4 sub-models (Layer 12, Layer 20, Layer 28, and Layer 36) and each time fully fine-tuned the extracted sub-model separately for two epochs on the Alpaca dataset. Figure 5.2 and Table D.7 shows the comparison between Extracted Fine-Tuned and SoFT sub-models. The first part in Table D.7 shows the equal training budget setup (2 Epochs) comparison in which SFT demonstrates slightly better performance compared to the similar SoFT sub-models. Further training SoFT will lead to better sorted sub-models in which SoFT outperforms the fully fine-tuned sub-models, proving the positive impact of SoFT on the performance of lower sub-models.

The analysis of the computational overhead of SoFT can be found in D.0.2.

5.5.2 How does SoFT work for other domains?

We further evaluated Sorted LLaMA in a different domain from the instruction following, selecting the TriviaQA [47] benchmark to assess the sub-models performance in open-domain closed-book questions answering.

Figure 5.3 shows the performance of SoFT and three baselines (SFT, Extracted Fine-Tuning and SFT+ICT) in different checkpoints through the training procedure on the TriviaQA benchmark. SoFT sub-models show significant superior performance compared to SFT and SFT+ICT counterparts in all sub-models. Similar to PandaLM, the gap between SoFT and SFT full-model performance is small in TriviaQA, which can underscore the SoFT capability in maintaining full-model performance compared to SFT. We also did Extracted Fine-Tuning on intermediate sub-models for 2 Epochs and results demonstrate close performance of SoFT intermediate layers to Extracted Fine-Tuning counterparts.

5.5.3 How can SoFT accelerate text generation?

Improving Speculative Sampling Speculative Decoding (SD) is a technique introduced by [13] to increase the speed of text decoding in large models. The method utilizes a large target and smaller draft models to generate tokens faster. We can verify the generated tokens by the large model in parallel. We used the same paradigm for Sorted LLaMA as we used earlier sub-models as draft and the full-size model as the target model. As the parameters have been shared between the large and draft models in this setup, we can avoid any extra memory overhead, unlike the standard Speculative Sampling. Table 5.2 reports the inference results of using speculative decoding on Alpaca and TriviaQA benchmarks in SoFT by using three different sub-models as drafts (Layer 12, 16, and 20). As shown combining Speculative decoding and Sorted LLaMA can speed up the token generation up to 1.16× compared to normal auto-regressive decoding in PandaLM with negligible performance drop. Due to the short average length of answers in TriviaQA, speculative decoding does not result in speed up as the draft generation process does not find any opportunity to accelerate inference.

Instance-Aware Dynamic Inference We also dynamically utilize SoFT sub-models to increase the speed of text generation during inference. Based on the confidence of the sub-model's predicted tokens, we decide which sub-model should generate which token. Given each token, the sub-models would process the token in size order (first smallest sub-model 12, then 16, and so on). Wherever in this procedure, the confidence of the predicted token by a sub-model reach a pre-defined threshold, the predicted token would be chosen as the next token and exit the model. We also implemented an adaptive caching mechanism in order to utilize KV caching in this non-trivial scenario where each token can exit from a different layer. Table 5.2 shows that Instance-Aware Dynamic Inference can speed up the normal auto-regressive approach in all benchmarks up to 1.34× in PandaLM and 1.12× in TriviaQA. Furthermore dynamic inference can result in better performance in PandaLM and TriviaQA compared to speculative decoding.

5.5.4 Analysis

A comparison between the learned probability distribution of SoFT versus SFT

Sorted tuning aims to make sub-models performance similar to the full model. To explore the efficacy of the SoFT in closing the gap between sub-models and the full model in instruction following task, we measure the similarity between probability distributions of each token in each sub-model versus the full model using the Kullback–Leibler (KL) divergence. Figure 5.4 (Left) compares the probability distribution of Sorted LLaMA and SFT sub-models at different output positions.

Figure 5.4a (Left) compares different SFT layers and the last Sorted LLaMA layer. The figure shows that only SFT's full-size output distribution is close to the sorted full-size model, while the other layers' distribution diverges faster in the initial steps compared to the SoFT. This is expected as the language model head is unfamiliar with the learned representation of the middle layers in SFT. In the next section, we compared the learned representations of different sub-models to understand SoFT's impact better.

Figure 5.4b (Left) compares the output distribution of all sorted layers to the last SFT layer. Compared to Figure 5.4a (Left), Figure 5.4b (Left) Sorted LLaMA can preserve the output distribution close to the SFT full-size model even in lower layers for initial output tokens.

The comparison between the last layer and the layers 12 to 36 in the SFT model is shown in Figure 5.5a (Left). It is clear from this figure that the output distribution diverges quickly compared to the last layer after generating a few initial tokens, even in higher layers like 36 and 32. It is important to note that this evaluation was generated without adjusting the classifier head.

Finally, Figure 5.5b (Left) demonstrates that in Sorted LLaMA, the likelihood distribution of the produced outcome becomes increasingly more similar to the full-size model as we get closer to the last layer.

A comparison between the learned representation of SoFT versus SFT

During regular fine-tuning, no connection between the language model head and sub-models can intensify the divergence of probability distributions in Figure 5.4 (Left). To overcome this, we conducted another experiment to compare the hidden state representation in the last and middle layers just before passing the hidden states to the language model head. Figure 5.4 (Right) compares the learned hidden state representation of SFT and Sorted LLaMA sub-models at various positions in the output. This will make the analysis independent of the language model head. We used cosine similarity to measure the difference between the two representations. As shown using heatmaps, the cosine similarities are highly correlated to the KL-Divergence comparison explained in the previous section.

Figure 5.4a (Right) compares all SFT sub-models with the Sorted last layer regarding hidden representation similarity. Again, similar to probability distribution analysis, the similarity between the SFT sub-model and Sorted last layer tends to fade immediately after generating the first few tokens, while Figure 5.4b demonstrates the capability of Sorted LLaMA sub-models in preserving the learned representations closely similar to the SFT last layer hidden states.

Figure 5.5a (Right) depicts the heatmap of hidden states cosine similarity among different SFT sub-models compared to the SFT last layer. Similar to its left plot, the similarity quickly diminishes after a few tokens, and this fade is more considerable in earlier layers.

On the other hand, Figure 5.5b (Right) shows that the representations of Sorted sub-models stay similar to the Sorted last layer even after generating multiple initial tokens.

5.6 Conclusion

This work presents sorted LLaMA, a many-in-one language model for dynamic inference obtained using Sorted Fine-Tuning (SoFT) instead of Standard Fine-tuning. Sorted LLaMA unlocks the potential capability of intermediate layers, offering dynamic adaptation without pre-training or additional costs related to model compression. It presents a promising avenue for optimizing generative language models in NLP. Our approach makes the deployment of these models more efficient. As all sub-models remain integral components of the original model, the burden of storage requirements and transition costs between different computational demands is minimized, making the management of multiple models during inference a practical reality.

Our systematic evaluation of instruction following and questions answering benchmarks challenged conventional wisdom by empowering middle layers to produce high-quality results. This, in turn, enables dynamic inference of LLMs with a highly efficient tuning method (SoFT), ultimately optimizing the usage of LLMs. Our encouraging results show the promising capability of SortedNet [105] to train multiple language models with different sizes at once.

5.7 Limitations

Despite showing the effectiveness of the SortedNet approach for large language models, further research is necessary to better understand the scope of its applicability in LLMs. For example, applying this method during pre-training, sorting other model dimensions such as attention heads and hidden dimensions, and investigating the impact of choosing a specific architecture could offer potential avenues for future research. Our study might be slightly biased to automated evaluation, requiring further investigation through human evaluation.

Query: Summarize a meeting from the given list of bullet points. Be sure to convert shorthand into a first-hand account.
Rose: Analyze data and presents findings
John: propose new idea
Jane: appointed to head project
Tom: need more time to fix software bug

Layer	LLaMA Response (SFT+ICT)	Sorted LLaMA Response (SoFT)
12	summatronics21220f0f ...	The meeting was held in order to discuss the project and its plans for the next few months. The presenters were present in the room, and they were able to provide a detailed explanation of the project and its goals. The feedback from the meeting was positive and encouraged the team to continue working towards the goal of delivering the project.
16	The first thing we hear about these characters individually separately apart aparticularly divisidedly mannered manneredness ...	The meeting held at the company's headquarters, where the three named individuals were present. The first person named in the report is Rousness, who has been assigned to task task task. The second person named in the report is John, who has proposed a new idea. The third person named in the report is Jane, who has been assigned to task task task. The third person named in the report is Jane, who has been assigned to task task task. The second person named in the report is Jane, who has been assigned to task task task.
20	meeting held meeting held meeting held participants discussion discussion discussion notable mentionsnone none None notable mentionsnone ...	The meeting held in the room with Rose, John, Jane, and Tom discusses the need for analyzing data and presenting findings , as well as the proposal of new ideas by John. Jane is appointed to head the project, and Tom needs more time to fix the software bug.
24	meeting summary: attended attended attended attended ...	The meeting was attended by Rose, John, Jane, and Tom. Rose was tasked with analyzing data and presenting findings. John proposed a new idea. Jane was appointed to head the project. Tom was granted additional time to fix the software bug.
28	summary: During meeting participants discuss data analysis findings presented by Rose followed proposal new idea by John appoint Jane head project Tom request additional time fix software bug	In the meeting, Rose analyzed data and presented findings, John proposed a new idea, and Jane was appointed to head the project. Tom also stated that he needed more time to fix the software bug.
32	summary: During meeting attended by Rose Analyst who analyzed data presented findings proposed new idea by John appointed Jane head project Tom Software engineer request more time fix bug	In the meeting, Rose analyzed data and presented her findings, while John proposed a new idea. Jane was appointed to head the project, and Tom requested more time to fix a software bug.
36	During the meeting, Rose presented her analysis of data collected from various sources. John proposed a new idea that could potentially improve efficiency and increase productivity. Jane was appointed head of a new project that will be launched soon. Tom requested more time to fix a software bug that has been affecting the company'vity since last month.	In the meeting, Rose analyzed data and presented her findings, while John proposed a new idea. Jane was appointed to head the project, and Tom requested more time to fix the software bug.
40	At the meeting, Rose presented the analysis of the data and shared her findings with the team. John proposed a new idea that he believed could benefit the company. Jane was appointed to head a new project that the company had recently taken on. Tom, however, requested more time to fix a software bug that had been discovered. The team discussed the importance of the bug fix and agreed to give Tom the additional time he needed.	In the meeting, Rose analyzed data and presented her findings. John proposed a new idea, and Jane was appointed to head the project. Tom requested more time to fix the software bug.

Table 5.1: Generated responses in different layers of LLaMA (Early Exit) and Sorted LLaMA for two samples from PandaLM validation set. Black phrases mean the answer is following the instruction and Red ones demonstrate errors and hallucinations.

	PandaLM			TriviaQA		
Auto-regressive Decoding						
Model	**Time per Token (ms)**	**Score**	**Rejection Ratio**	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**
Layer 40 (full)	94.07	-	-	91.27	37.95	-
Speculative Decoding						
Draft Model	**Time per Token (ms)**	**Score**	**Rejection Ratio**	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**
Layer 12	80.86 (1.16×)	-0.144	0.37	110.50 (0.82×)	34.36	0.72
Layer 16	84.10 (1.11×)	-0.211	0.31	118.92 (0.76×)	34.16	0.70
Layer 20	84.50 (1.11×)	-0.144	0.26	139.78 (0.65×)	34.19	0.66
Instance-Aware Dynamic Inference						
Model	**Time per Token (ms)**	**Score**	**Rejection Ratio**	**Time per Token (ms)**	**Accuracy**	**Rejection Ratio**
Layer 12:40	69.91 (1.34×)	-0.050	-	81.01 (1.12×)	36.53	-

Table 5.2: Speed-up in inference time on three PandaLM and TriviaQA benchmarks by utilizing Speculative Decoding and Instance-Aware Dynamic Inference techniques. Score column in PandaLM section means the score of the model versus the Auto-regressive generated results based on Equation 5.2.

Chapter 6

Conclusion

As part of this comprehensive study, we have introduced and explored innovative methodologies that push the boundaries of language models and neural networks, aiming to optimize symbolic regression, dynamic inference, and low-rank adapters. Our novel approach to symbolic regression, SymbolicGPT, integrates language models with symbolic regression and order-invariant representations, offering a scalable and efficient solution that outperforms traditional methods. This is complemented by our dynamic neural network training approach, SortedNet, which capitalizes on the modularity of deep networks, employing a stochastic gradient accumulation method to train multiple sub-networks simultaneously. This not only enhances generalization but also facilitates dynamic inference by allowing samples to exit at intermediate layers, optimizing computational efficiency.

Furthermore, we introduced DyLoRA, a solution that addresses the challenges of rank selection in low-rank adapters, making them dynamic at inference time. This eliminates the need for repetitive re-training and rank optimization, streamlining the process and enhancing efficiency. In parallel, our Sorted LLaMA model, a dynamic language model, utilizes Sorted Fine-Tuning (SoFT) to harness the potential of intermediate layers, allowing for dynamic adaptation without the overhead of pre-training or model compression. This approach optimizes the deployment and management of multiple models during inference, reducing storage and computational demands.

In essence, our work challenges conventional paradigms, demonstrating the potential of leveraging intermediate layers and dynamic methodologies in neural networks and language models. Future directions include exploring the scalability limits of SymbolicGPT, refining the constant optimization step in our algorithm, applying the sorted training to multi-task learning, analyzing the disentanglement ability of sorted mechanism, reducing the performance gap with individual training and further investigating the benefits of sample-efficient inference. Our findings underscore the transformative potential of these methods, paving the way for more efficient and adaptive machine-learning applications.

References

[1] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

[2] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Maitha Alhammadi, Mazzotta Daniele, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of language models: Towards open frontier models. 2023.

[3] Douglas Adriano Augusto and Helio JC Barbosa. Symbolic regression via genetic programming. In *Neural Networks, 2000. Proceedings. Sixth Brazilian Symposium on*, pages 173–178. IEEE, 2000.

[4] Hritik Bansal, Karthik Gopalakrishnan, Saket Dingliwal, Sravan Bodapati, Katrin Kirchhoff, and Dan Roth. Rethinking the role of scale for in-context learning: An interpretability-based case study at 66 billion scale. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11833–11856, Toronto, Canada, July 2023. Association for Computational Linguistics.

[5] Luca Biggio, Tommaso Bendinelli, Aurelien Lucchi, and Giambattista Parascandolo. A seq2seq approach to symbolic regression. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

[6] Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. *arXiv preprint arXiv:2106.06427*, 2021.

[7] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[8] Jure Brence, Ljupčo Todorovski, and Sašo Džeroski. Probabilistic grammars for equation discovery. *Knowledge-Based Systems*, page 107077, 2021.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[11] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[12] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-All: Train One Network and Specialize it for Efficient Deployment, April 2020. arXiv:1908.09791 [cs, stat].

[13] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[14] Gang Chen. Learning symbolic expressions via gumbel-max equation learner network. *arXiv preprint arXiv:2012.06921*, 2020.

[15] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *International conference on machine learning*, pages 1691–1703. PMLR, 2020.

[16] Patrick Chen, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Drone: Data-aware low-rank compression for large nlp models. *Advances in neural information processing systems*, 34:29321–29334, 2021.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari,

Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[18] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[19] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[22] Fnu Devvrit, Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit S Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham M Kakade, Ali Farhadi, et al. Matformer: Nested transformer for elastic inference. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[24] Ali Edalati, Marzieh Tahaei, Ivan Kobyzev, Vahid Partovi Nia, James J Clark, and Mehdi Rezagholizadeh. Krona: Parameter efficient tuning with kronecker adapter. *arXiv preprint arXiv:2212.10650*, 2022.

[25] Utku Evci, Max Vladymyrov, Thomas Unterthiner, Bart van Merriënboer, and Fabian Pedregosa. Gradmax: Growing neural networks using gradient information. *arXiv preprint arXiv:2201.05125*, 2022.

[26] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[27] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*, 2018.

[28] Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, NY, USA, second edition, 1987.

[29] Claire Gardent, Anastasia Shimorina, Shashi Narayan, and Laura Perez-Beltrachini. The webnlg challenge: Generating text from rdf data. In *Proceedings of the 10th International Conference on Natural Language Generation*, pages 124–133, 2017.

[30] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366*, 2021.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[34] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[35] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[36] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *CoRR*, abs/1904.09751, 2019.

[37] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems*, 33:9782–9793, 2020.

[38] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[39] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[40] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.

[41] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

[42] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[43] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[44] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[45] Aref Jafari, Mehdi Rezagholizadeh, Pranav Sharma, and Ali Ghodsi. Annealing knowledge distillation. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 2493–2504, Online, April 2021. Association for Computational Linguistics.

[46] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[47] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

[48] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers. *Advances in Neural Information Processing Systems*, 34:1022–1035, 2021.

[49] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2019.

[50] Samuel Kim, Peter Y Lu, Srijon Mukherjee, Michael Gilbert, Li Jing, Vladimir Čeperić, and Marin Soljačić. Integration of neural network-based symbolic regression in deep learning for scientific discovery. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[51] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[52] Michael Kommenda, Bogdan Burlacu, Gabriel Kronberger, and Michael Affenzeller. Parameter identification for symbolic regression using nonlinear least squares. *Genetic Programming and Evolvable Machines*, pages 1–31, 2019.

[53] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[54] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *International Conference on Learning Representations*, 2020.

[55] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[56] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[57] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[58] Jiwei Li, Will Monroe, and Dan Jurafsky. A simple, fast diverse decoding algorithm for neural generation. *arXiv preprint arXiv:1611.08562*, 2016.

[59] Lei Li, Yankai Lin, Shuhuai Ren, Peng Li, Jie Zhou, and Xu Sun. Dynamic knowledge distillation for pre-trained language models. *arXiv preprint arXiv:2109.11295*, 2021.

[60] Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, and Kai-Wei Chang. Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557*, 2019.

[61] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[62] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *Advances in neural information processing systems*, 32, 2019.

[63] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madian Khabsa. Unipelt: A unified framework for parameter-efficient language model tuning. *arXiv preprint arXiv:2110.07577*, 2021.

[64] Georg S Martius and Christoph Lampert. Extrapolation and learning equations. In *5th International Conference on Learning Representations, ICLR 2017-Workshop Track Proceedings*, 2017.

[65] Ben McKay, Mark J Willis, and Geoffrey W Barton. Using a tree structured genetic algorithm to perform symbolic regression. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 487–492. IET, 1995.

[66] A Murari, E Peluso, M Gelfusa, I Lupelli, M Lungaroni, and P Gaudio. Symbolic regression via genetic programming for data driven derivation of confinement scaling laws without any assumption on their mathematical form. *Plasma Physics and Controlled Fusion*, 57(1):014008, 2014.

[67] Linyong Nan, Dragomir Radev, Rui Zhang, Amrit Rau, Abhinand Sivaprasad, Chiachun Hsieh, Xiangru Tang, Aadit Vyas, Neha Verma, Pranav Krishna, et al. Dart: Open-domain structured data record to text generation. *arXiv preprint arXiv:2007.02871*, 2020.

[68] Matan Ben Noach and Yoav Goldberg. Compressing pre-trained language models by matrix decomposition. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 884–889, 2020.

[69] Jekaterina Novikova, Ondřej Dušek, and Verena Rieser. The e2e dataset: New challenges for end-to-end generation. *arXiv preprint arXiv:1706.09254*, 2017.

[70] Elvis Nunez, Maxwell Horton, Anish Prabhu, Anurag Ranjan, Ali Farhadi, and Mohammad Rastegari. Lcs: Learning compressible subspaces for efficient, adaptive, real-time network compression at inference time. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 3818–3827, January 2023.

[71] OpenAI. Gpt-4 technical report, 2023.

[72] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[73] Peyman Passban, Yimeng Wu, Mehdi Rezagholizadeh, and Qun Liu. ALP-KD: attention-based layer projection for knowledge distillation. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 13657–13665. AAAI Press, 2021.

[74] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[75] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay.

The RefinedWeb dataset for Falcon LLM: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.

[76] Brenden K Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021.

[77] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247*, 2020.

[78] Gabriele Prato, Ella Charlaix, and Mehdi Rezagholizadeh. Fully quantized transformer for machine translation. *arXiv preprint arXiv:1910.10485*, 2019.

[79] Gabriele Prato, Ella Charlaix, and Mehdi Rezagholizadeh. Fully quantized transformer for machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1–14, 2020.

[80] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

[81] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *preprint*, 2018.

[82] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[83] Ahmad Rashid, Vasileios Lioutas, and Mehdi Rezagholizadeh. Mate-kd: Masked adversarial text, a companion to knowledge distillation. *arXiv preprint arXiv:2105.05912*, 2021.

[84] Oren Rippel, Michael Gelbart, and Ryan Adams. Learning ordered representations with nested dropout. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1746–1754, Bejing, China, 22–24 Jun 2014. PMLR.

[85] Lars Ruthotto and Eldad Haber. An introduction to deep generative modeling. *GAMM-Mitteilungen*, page e202100008, 2021.

[86] Subham Sahoo, Christoph Lampert, and Georg Martius. Learning equations for extrapolation and control. In *International Conference on Machine Learning*, pages 4442–4450. PMLR, 2018.

[87] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[88] Iqbal H Sarker. Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2(6):420, 2021.

[89] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[90] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[91] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[92] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[93] Weijie Su, Xizhou Zhu, Yue Cao, Bin Li, Lewei Lu, Furu Wei, and Jifeng Dai. Vl-bert: Pre-training of generic visual-linguistic representations. *arXiv preprint arXiv:1908.08530*, 2019.

[94] Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. Black-box tuning for language-model-as-a-service. *arXiv preprint arXiv:2201.03514*, 2022.

[95] Yi-Lin Sung, Jaemin Cho, and Mohit Bansal. Lst: Ladder side-tuning for parameter and memory efficient transfer learning. *Advances in Neural Information Processing Systems*, 35:12991–13005, 2022.

[96] Marzieh S Tahaei, Ella Charlaix, Vahid Partovi Nia, Ali Ghodsi, and Mehdi Rezagholizadeh. Kroneckerbert: Learning kronecker decomposition for pre-trained language models via knowledge distillation. *arXiv preprint arXiv:2109.06243*, 2021.

[97] Chaofan Tao, Lu Hou, Wei Zhang, Lifeng Shang, Xin Jiang, Qun Liu, Ping Luo, and Ngai Wong. Compression of generative pre-trained language models via quantization. *arXiv preprint arXiv:2203.10705*, 2022.

[98] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

[99] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[100] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[101] Silviu-Marian Udrescu and Max Tegmark. AI feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

[102] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[103] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[104] Mojtaba Valipour, Mehdi Rezagholizadeh, Ivan Kobyzev, and Ali Ghodsi. Dylora: Parameter efficient tuning of pre-trained models using dynamic search-free low-rank adaptation. *arXiv preprint arXiv:2210.07558*, 2022.

[105] Mojtaba Valipour, Mehdi Rezagholizadeh, Hossein Rajabzadeh, Marzieh Tahaei, Boxing Chen, and Ali Ghodsi. Sortednet, a place for every network and every network in its place: Towards a generalized solution for training many-in-one neural networks. *arXiv preprint arXiv:2309.00255*, 2023.

[106] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[107] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[108] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[109] Ruize Wang, Duyu Tang, Nan Duan, Zhongyu Wei, Xuanjing Huang, Guihong Cao, Daxin Jiang, Ming Zhou, et al. K-adapter: Infusing knowledge into pre-trained models with adapters. *arXiv preprint arXiv:2002.01808*, 2020.

[110] Yidong Wang, Zhuohao Yu, Zhengran Zeng, Linyi Yang, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, Xing Xie, et al. Pandalm: An automatic evaluation benchmark for llm instruction tuning optimization. *arXiv preprint arXiv:2306.05087*, 2023.

[111] Yiqun Wang, Nicholas Wagner, and James M Rondinelli. Symbolic regression in materials science. *MRS Communications*, 9(3):793–805, 2019.

[112] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[113] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. *arXiv preprint arXiv:1910.04732*, 2019.

[114] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[115] Minghao Wu, Abdul Waheed, Chiyu Zhang, Muhammad Abdul-Mageed, and Alham Fikri Aji. Lamini-lm: A diverse herd of distilled models from large-scale instructions. *arXiv preprint arXiv:2304.14402*, 2023.

[116] Qiaolin Xia, Haoyang Huang, Nan Duan, Dongdong Zhang, Lei Ji, Zhifang Sui, Edward Cui, Taroon Bharti, and Ming Zhou. Xgpt: Cross-modal generative pre-training for image captioning. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 786–797. Springer, 2021.

[117] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

[118] Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1803–1811, 2019.

[119] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018.

[120] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable Neural Networks, December 2018. arXiv:1812.08928 [cs].

[121] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[122] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*, 2023.

APPENDICES

Appendix A

SymbolicGPT

A.1 Conditional Language Models

Deep Generative Models (DGMs) are multi-layered neural networks that approximate complex probability distributions using a large number of samples. Once trained, these models can be used to estimate the likelihood of any observation [85]. DGMs aim to learn a mapping between a random variable $\mathcal{X}$ defined over $\mathbb{R}^d$ and another random variable $\mathcal{Y}$ defined over $\mathbb{R}^{d'}$. This means for each sample $\mathbf{x} \sim \mathcal{X}$, we assume there is a sample $\mathbf{y} \sim \mathcal{Y}$ such that $\mathbf{y} \approx g(\mathbf{x})$ [85]. To specify we want to obtain the following function:

$$g : \mathbf{x} \rightarrow \mathbf{y} \text{ where } \mathbf{x} \in \mathbb{R}^d, \mathbf{y} \in \mathbb{R}^{d'} \tag{A.1}$$

Pre-trained generators have been proven successful in the literature [81, 82, 10]. Most of these generators, however, are not fully controllable and in order to add any sort of controllability, we need to make these models conditional with respect to a given additional input "**c**". In other words, we need to learn the following function in the most general form:

$$g_c : (\mathbf{x}, \mathbf{c}) \rightarrow \mathbf{y} \text{ where } \mathbf{x} \in \mathbb{R}^d, \mathbf{c} \in \mathbb{R}^c, \mathbf{y} \in \mathbb{R}^{d'} \tag{A.2}$$

As shown in the following equation, the common practice to handle this additional input is to see the new information as an extension of the input. We used a similar approach to propose our SymbolicGPT architecture for the task of symbolic regression. In this equation ';' is simply the concatenation.

$$g_e : [\mathbf{x}; \mathbf{c}] \in \mathbb{R}^{d+c} \rightarrow \mathbf{y} \in \mathbb{R}^{d'} \tag{A.3}$$

Although adding the condition as another input is beneficial, we are missing an opportunity to use an inductive bias to simplify the process of learning. If the condition is passed as another input, the model will need to learn how to distinguish between the input "**x**" and the condition "**c**", and how to change the behavior of the learned function based on the given condition. This might be challenging for the optimizer. There may be several possible values for a condition, each of which may represent a complicated function. We can use the knowledge that we have about this additional input to simplify the process of learning. Therefore, we propose to use a function f that selects the corresponding function g based on the input condition "**c**". In this way, we have:

$$f : \mathbf{c} \rightarrow g_c \ , \ g_c : \mathbf{x} \rightarrow \mathbf{y} \text{ where } \mathbf{x} \in \mathbb{R}^d, \mathbf{c} \in \mathbb{R}^c, \mathbf{y} \in \mathbb{R}^{d'} \tag{A.4}$$

and we just need to train each of these different models g_i accordingly. Based on this intuition, we propose another solution for the task of symbolic regression which we call Conditional SymbolicGPT (C-SymbolicGPT). In the following sections, we will describe our proposed methods in more detail.

A.2 C-SymbolicGPT: Model-agnostic Conditional Generative Models For Symbolic Regression

As mentioned before, we want to train different functions for each value of the condition. In a finite discrete case, this is a reasonable and practical choice. Unfortunately, we do not have access to infinite resources, and therefore in the case of a continuous condition, having many models is practically impossible. Additionally, these models benefits from the knowledge that each separate model learns. Therefore, sharing the parameters of these models helps to generalize better. Sharing all the parameters, however, is a strong assumption. So in order to loosen this assumption, instead of using completely different functions with different parameters we will use the same parameters with different architectures. To determine the architecture of the function "g", we use another learnable function "f". This means instead of:

$$\mathbf{y} = g(\mathbf{x}, \mathbf{c}) \tag{A.5}$$

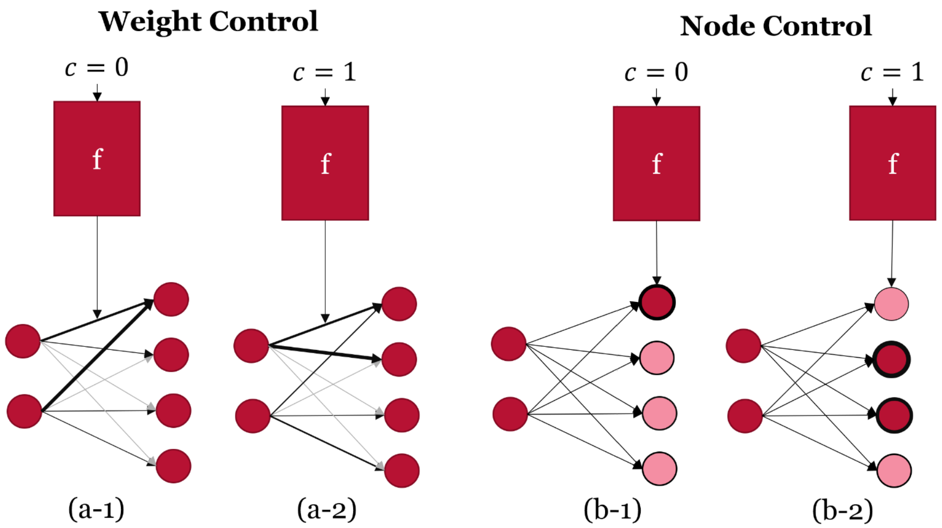


Figure A.1: In this example assume that we have a condition with two possible values (c=0 or c=1). Therefore we are looking to learn two different models for each condition. (a) Figures a-1 and a-2 show the weight control approach for cases where c=0 and c=1. As can be seen, our proposed model tries to come up with different architectures by adjusting the importance of the weights. (b) We call our more practical approach as node control, which only re-weights the importance of the output nodes, not the weights.

According to our proposed method, we have a function “f” that will change the architecture of the generator “g” based on the given condition “**c**”:

$$g_c = f(\mathbf{c}) \tag{A.6}$$

And then we will use the new “g” to generate the corresponding **y** based on the input observation:

$$\mathbf{y} = g_c(\mathbf{x}) \tag{A.7}$$

To further explain the difference between our proposed method and the previously mentioned input-extension method, we demonstrate both methods for the linear case. In the input extension method we have:

$$\mathbf{y} = [\mathbf{x}; \mathbf{c}]W = \sum_{i=1}^{d} \mathbf{x}_i W_i + \sum_{i=d+1}^{d+c} \mathbf{c}_{i-d} W_i \text{ where } W \in R^{(d+c)\times d'} \tag{A.8}$$

As seen, in this input extension method, we need to learn additional set of $W \in R^{c\times d'}$ to learn the contribution of the "$\mathbf{c}_i$" independently of the "$\mathbf{x}$". Instead in our proposed method we have:

$$\mathbf{y} = \sum_{i=1}^{d} \mathbf{x}_i W_i^{adjust} \text{ where } W_i^{adjust} = f(c_i, W_i) \tag{A.9}$$

Our idea is based on the intuition that changing the connections (weights) of the neural network is same as changing the architecture of the model, and therefore come up with new functions. This has been shown in Figure A.1-a, as weight control.

This method, however, might introduce many new parameters. To make it even more efficient, we assume that we just need to change the output of each layer. This means we have the following equation:

$$\mathbf{y} = f(\mathbf{c}, \sum_{i=1}^{d} (\mathbf{x}_i W_i)) \tag{A.10}$$

This can be seen in Figure A.1-b, and we name this method node control. To make this easier in term of implementation we propose to use a novel layer that given "**c**" try to adjust the value of the output of that layer. In other words, we will use "**c**" only to reweights the importance of each neuron. This is different than the previous approaches because "**c**" will not only have effect on the input layer which might get forget later in the next deep layers. We give "**c**" full control over the whole network to appropriately change the behavior of the network based on the condition.

Our proposed method can also be interpreted similar as Dropout [92] with the primary difference that Dropout use a Bernoulli distribution to randomly mask an output. Instead, we are using another network to learn the importance of that output.

A.3 Model-Agnostic Conditional Language Models

Generative language models are complex neural networks that can output distribution over all supported language tokens. As a result, it is possible to sample full sentences from their distributions. Many sampling techniques have been proposed in the literature to produce more human-like sentences. These include nucleus sampling [36], top-k sampling [27], and beam-search [58]. Guiding the output of this process, however, is not an easy task. In this section, a novel method is proposed for controlling a generative model's output using another neural network.

Formally, a left-to-right language model is a generator with the following probability distribution:

$$p(t_1, ..., t_l) = \prod_{i=1}^{l} p(t_i | t_1, ..., t_{i-1}) \tag{A.11}$$

which is only a function of previous tokens $g(e(t_i)|e(t_{0:i-1}))$ where e is the embedding layer that maps each token index to its corresponding vector. We propose adding another term **c** to the conditional distribution and use our proposed method to make the model conditional:

$$p(t_1, ..., t_l) = \prod_{i=0}^{l+1} p(t_i | t_1, ..., t_{i-1}, \mathbf{c}) \tag{A.12}$$

where **c** is the additional condition. We used standard left-to-right language models as the backbone and we added our proposed layers to make the model conditional without changing the original architecture. As shown in the next equation, it is not necessary to change the original model. We just need to add our proposed layers in between.

$$p(t_1, ..., t_l) = \prod_{i=0}^{l+1} p_j(t_i | t_1, ..., t_{i-1}) \text{ where } p_j = f(\mathbf{c}) \tag{A.13}$$

Unlike treating **c** as another token, our proposed method will not get forgotten during training.

A.4 Experiments

A.4.1 Image Generation:

Additionally, we applied our proposed method to improve the generation of images in StyleGAN [49]. As shown in Figure A.2, we successfully shown the effectiveness of our approach. For CIFAR10, we have strong improvements over all the metrics. CIFAR100 also shows some

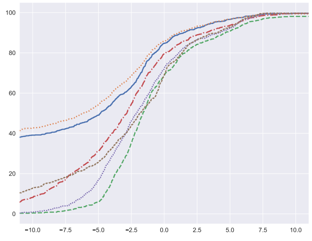
(a) (b)

Figure A.2: Image Generation using a modified StyleGAN using our proposed MCGMs a) Evaluated metrics on CIFAR10 b) Evaluated metrics on CIFAR100

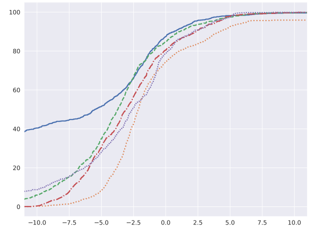
improvement, however, this improvement is not as notable as the CIFAR10 experiment. Due to the larger number of classes, initially, we even expect to have a better improvement in the CIFAR100 experiment. It seems, however, that the greater number of classes with strong overlap may cause some instabilities. This needs further investigation.

A.4.2 Symbolic Regression

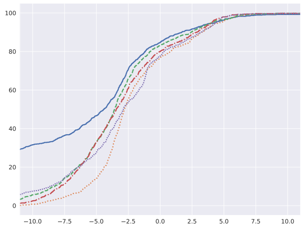
To test our proposed models, we implemented both SymbolicGPT and C-SymbolicGPT. We trained these models in different settings, which we detail below. In all cases, we trained SymbolicGPT (and C-SymbolicGPT) over 20 epochs using a batch size of 128. The embedding size for the T-net vector representation is $e = 512$, and the maximum equation output length was capped at 200 tokens.



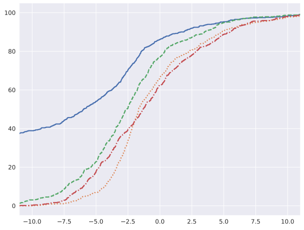
(a) 1 variable



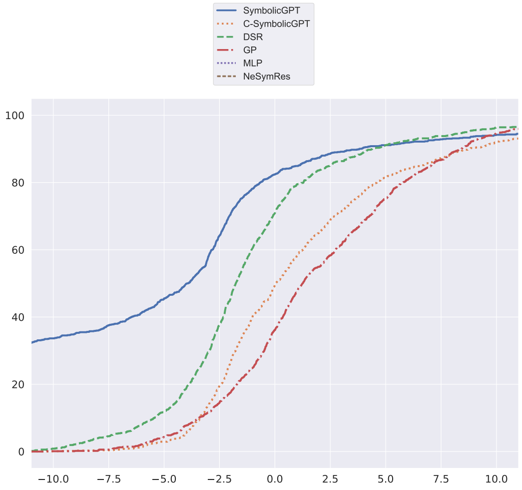
(b) 2 variables



(c) 3 variables



(d) 1 to 9 variables (General)



(e) 1 to 9 variables: This experiment includes all of the benchmark equations in both AI Feynman [101] and Nguyen benchmarks [103]. We used these benchmarks because it is a common practice in the literature.

Figure A.3: Cumulative $\log MSE_N$ over all methods and experiments. Each curve shows the proportion of test cases that attained an error score less than every given threshold. SymbolicGPT finds better fitting equations for more test cases than any other method tested, as well as more highly accurate equations (with $\log MSE_N < -10$).

Appendix B

DyLoRA

B.1 Time Complexity

The training time for DyLoRA is comparable to that of LoRA trained once on a specific rank. Thus, when searching the rank space for LoRA, we need to train it multiple times, whereas our method does not require searching the ranks. Accordingly, DyLoRA's relative time complexity is inversely proportional to the number of possible ranks for which the LoRA model must be searched. In MRPC, DyLoRA (for all the ranks) and LoRA (only on a single rank 8) require a total training time of 408.39 seconds and 399.95 seconds, respectively. Consequently, when we need to train eight LoRA models (Rank=1,2,...,8), it will result in a cost of 399.95*8=3199.6s, compared to the training time of our model, which is only 408.39 seconds. A more efficient implementation of our algorithm may result in a better time complexity.

B.2 Hyperparameters

We did not use any parameter tuning nor MNLI trick (initializing some downstream tasks from MNLI checkpoint instead of pre-trained weights). Therefore, we fine-tuned all the datasets from the original pre-trained weights. We simply followed unified hyper-parameters for all different experiments. Unlike LoRA [42], which reported the median over five random seeds, we reported the mean and standard deviation over five random seeds. See the details in Table B.1.

Model	Parameter	Value
RoBERTa-Base	Optimizer	AdamW
	Warmup Ratio	0.06
	LR Scheduler	Linear
	Batch Size	32
	Epochs	30
	Learning Rate (LR)	4e-4
	Weight Decay	0.1
	LoRA Config	$r_q = r_v = 8$ (unless otherwise mentioned)
	LoRA α	16
	Max Sequence Length	512
	Seeds	10, 42, 4242, 10, 1010
	GPU	Tesla V100-PCIE-32GB
GPT Medium	Optimizer	AdamW
	Adam Beta2	0.999
	Warmup Steps	500
	Clip	0.0
	LR Scheduler	Linear
	Batch Size	8
	Epochs	5
	Learning Rate (LR)	2e-4
	Weight Decay	0.01
	Correct Bias	True
	LoRA Dropout	0.1
	Lable Smooth	0.1
	LoRA Config	$r_q = r_v = 4$
	LoRA α	32
	Seeds	10, 42, 4242
	GPU	Tesla V100-PCIE-32GB

Table B.1: All the hyperparameters that have been used throughout our study.

B.3 GPT Experiments

A summary of the additional experiments that have been conducted to demonstrate the effectiveness of our proposed method for the task of language generation is provided in Table B.2.

Model (Method)	**Trainable Params**	DART		WebNLG	
		BLEU↑	**TER↓**	**BLEU↑**	**TER↓**
Rank=1					
GPT-2 M (LoRA)	0.09M	0.71	0.49	2.80	1.18
GPT-2 M (DyLoRA-Frozen)	0.09M	$44.48_{\pm 0.11}$	$0.49_{\pm 0.00}$	$52.09_{\pm 0.10}$	$0.40_{\pm 0.01}$
GPT-2 M (DyLoRA)	0.09M	$44.77_{\pm 0.17}$	$0.49_{\pm 0.01}$	$53.04_{\pm 0.07}$	$0.40_{\pm 0.00}$
Rank=2					
GPT-2 M (LoRA)	0.19M	15.90	0.48	26.58	0.67
GPT-2 M (DyLoRA-Frozen)	0.19M	$45.04_{\pm 0.14}$	$0.48_{\pm 0.01}$	$52.74_{\pm 0.31}$	$0.40_{\pm 0.01}$
GPT-2 M (DyLoRA)	0.09M	$46.05_{\pm 0.31}$	$0.48_{\pm 0.00}$	$54.32_{\pm 0.09}$	$0.39_{\pm 0.01}$
Rank=3					
GPT-2 M (LoRA)	0.29M	35.84	0.47	43.61	0.47
GPT-2 M (DyLoRA-Frozen)	0.29M	$45.22_{\pm 0.14}$	$0.49_{\pm 0.01}$	$53.03_{\pm 0.55}$	$0.40_{\pm 0.00}$
GPT-2 M (DyLoRA)	0.29M	$46.68_{\pm 0.36}$	$0.48_{\pm 0.01}$	$54.48_{\pm 0.05}$	$0.39_{\pm 0.00}$
Rank=4					
GPT-2 M (LoRA)	0.39M	47.10	0.46	55.57	0.39
GPT-2 M (DyLoRA-Frozen)	0.39M	$45.56_{\pm 0.33}$	$0.48_{\pm 0.00}$	$53.03_{\pm 0.01}$	$0.40_{\pm 0.00}$
GPT-2 M (DyLoRA)	0.39M	$46.56_{\pm 0.42}$	$0.48_{\pm 0.01}$	$54.48_{\pm 0.45}$	$0.39_{\pm 0.00}$
Fine-Tune					
GPT-2 M (FT)*	354M	46.2	0.46		

Table B.2: Rows with * have been reported from the LoRA paper. [42].

Appendix C

SortedNet

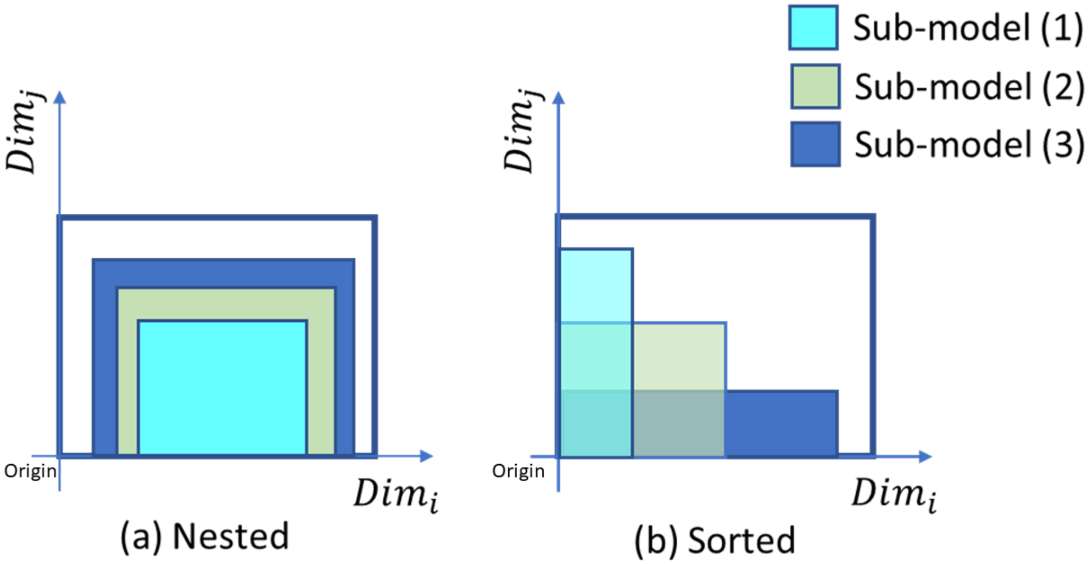
C.1 Sorted vs. Nested Architectures

In this work, we introduce the term of *sorted* architectures to extend and generalize the concept of nested architectures. In contrast to nested models in which smaller sub-models are always totally encapsulated by larger sub-models, our sorted sub-models would be tied to the origin (starting index) of each dimension independently.

C.2 Related Work

Slimmable Networks [120] Slimmable networks is a method for training a single neural network in a way that it can be deployed with adjustable width at the inference time. This solution was proposed particularly for CNN architectures and thus, careful consideration of the batch normalization module for various width sizes is necessary. In this regard, in slimmable networks, switchable batch normalization was used which lead to additional trainable parameters. In contrast to slimmable networks, our SortedNet are architecture agnostic and work in both depth and width dimensions.

Early Exit [117] Early exit refers to a technique which adds a classifier to intermediate layers of an already trained neural network. While the parameters of the main model are frozen, the parameters of the classifiers are updated in a separate fine-tuning process. In this approach, each of the classifiers and their subsequent network can be treated as an independent sub-model. While



(a) Nested

(b) Sorted

Figure C.1: Illustrating the difference between the nested and sorted sub-models. In nested architectures, smaller sub-models are encapsulated by larger sub-models, which is not necessarily the case for what we refer to as sorted models. Moreover, sorted models are tied to the origin (i.e. starting index) of each dimension which might not be the case in nested models.

this solution is relatively straightforward, the performance of the sub-models lags significantly behind that of the main model. Also dedicating a separate classification head to each sub-model can significantly increase the memory demand at inference.

Dayna-BERT [37] Dyna-BERT presents a dynamic compression method for pre-trained BERT models, enabling flexible adjustments in model size, both in depth and width, during inference. While the objective introduced in the DynaBERT paper shares some similarities with our approach, there are several key distinctions. Firstly, in DynaBERT, only a few subsets of the model are functional, whereas our SortedNet does not rely on such assumptions. Secondly, DynaBERT requires an already trained teacher model and utilizes knowledge distillation, whereas our technique operates independently of knowledge distillation. Thirdly, DynaBERT necessitates a search for an optimal sub-model, whereas our solution is inherently *search-free*. Lastly, DynaBERT's applicability is dependent on the architecture, whereas our approach is architecture-agnostic.

Layer-drop [26] Layer-drop is a structured dropout solution at the training time which allows layer pruning at the inference time. Similar to DynaBERT, this solution is applied to pre-trained language models; however, in contrast to DynaBERT, Layer-drop only targets the depth of neural networks and not their width. In Layer-drop, there is no fixed training pattern and any layer can be dropped with a certain probability, which is referred to as drop rate. At the inference time, the number of active layers can be adjusted by the drop-rates that are seen during the training time of that network (i.e. to achieve the best performance on any other drop-rate value, the network needs to be re-trained.). Layer-drop works only in depth while our solution works for both depth and width. Moreover, Layer-Drop requires specific search patterns for dropping layers at the inference time and training time, whereas our solution is search free.

Once-for-All [12] Once-for-all(OFA) targets efficient inference across different devices by first training an OFA network which supports many sub-models with varying latency/accuracy characteristics ; it then searches among the feasible sub-models according to the accuracy and latency requirements of their target device. OFA has a progressive training nature i.e. it goes from the largest model to the smaller sub-models. OFA is different from our solution from the following aspects: first, it needs teacher and knowledge distillation; second, OFA requires a separate Neural Architecture Search (NAS) at the inference time; third, OFA is not architecture agnostic (their solution is for CNN-based neural networks while our SortedNet works for both CNNs and Transformers). Moreover, OFA is different from our solution in terms of the sub-model selection strategy. While our SortedNet selects sub-models in a sorted manner, OFA does not have any particular assumption for sorting sub-models (see Fig. C.2 for more details).

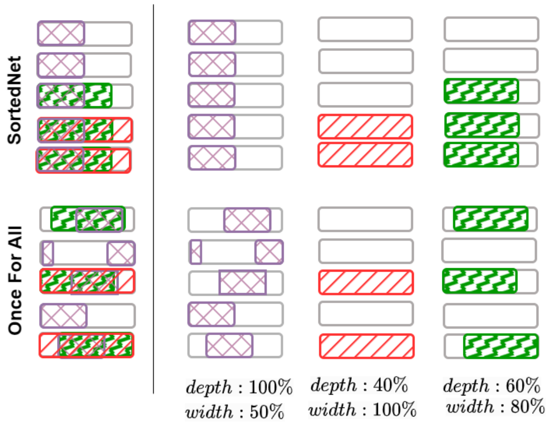


Figure C.2: Comparing SortedNet and Once For All: on a hypothetical 5-layer network, we show how the sub-model selection strategy of SortedNet differs from the Once-for-All [12] approach.

Learning Compressible Subspace [70] Learning Compressible Subspace (LCS) is an adaptive compression technique based on training compressible subspace of neural networks (using a linear convex combination of two sets of weights for the network). While LCS does not require any re-training at the inference time, this solution has some other limitations including: first, it needs double memory at the training time; second, the choices of initial weights and the compression function are unclear and arbitrary (left as a hyper-parameter); third, it is only tried on CNNs; forth, similar to Layer-drop intermediate sub-models are trained randomly which will make the performance of the target model sub-optimal.

C.3 Theoretical Analysis

C.3.1 Parameter Convergence in Identically Trained Sub-networks

Suppose $\hat{f}$ is a sub-network within a larger neural network architecture, and f represents an identical network architecture trained independently. We aim to understand the relationship between the parameters of these two networks, θ for $\hat{f}$ and ϕ for f, as they are trained under identical conditions.

C.3.2 Assumption of Lipschitz Continuity of Gradients

We assume that the gradients of the loss functions for $\hat{f}$ and f, denoted as $\mathcal{L}_{\hat{f}}$ and $\mathcal{L}_f$ respectively, are L-Lipschitz continuous. This implies that:

$$\|\nabla\mathcal{L}_{\hat{f}}(\theta) - \nabla\mathcal{L}_{\hat{f}}(\theta')\| \leq L\|\theta - \theta'\|$$
$$\|\nabla\mathcal{L}_f(\phi) - \nabla\mathcal{L}_f(\phi')\| \leq L\|\phi - \phi'\|$$

for all θ, θ' and ϕ, ϕ' in the parameter space.

C.3.3 Parameter Update Rule

The parameters of the networks are updated via gradient descent as follows:

- For $\hat{f}$:
$$\theta_{t+1} = \theta_t - \eta\nabla\mathcal{L}_{\hat{f}}(\theta_t)$$
- For f:
$$\phi_{t+1} = \phi_t - \eta\nabla\mathcal{L}_f(\phi_t)$$

C.3.4 Derivation of the Bound

We derive a bound on the difference in parameters between $\hat{f}$ and f after each training iteration:

$$\|\theta_{t+1} - \phi_{t+1}\| = \|\theta_t - \phi_t - \eta(\nabla\mathcal{L}_{\hat{f}}(\theta_t) - \nabla\mathcal{L}_f(\phi_t))\|$$

Applying the triangle inequality and the Lipschitz continuity of the gradients, we obtain:

$$\|\theta_{t+1} - \phi_{t+1}\| \leq (1 + \eta L)\|\theta_t - \phi_t\| + \eta C$$

where C is a constant that bounds the difference between the gradients of the loss functions of the networks.

This bound quantifies the evolution of the difference in parameters between $\hat{f}$ and f across training iterations, incorporating the impact of the Lipschitz constant L, the learning rate η, and the constant C that bounds the inherent difference in gradients.

C.3.5 Negligible C under Identical Training Conditions

Given that $\hat{f}$ and f are trained under perfectly identical conditions (same data, initialization, and hyperparameters), the difference in their gradients can be considered negligible, leading us to conclude that C is practically zero. Under this assumption, the bound simplifies significantly:

$$\|\theta_{t+1} - \phi_{t+1}\| \leq (1 + \eta L)\|\theta_t - \phi_t\|$$

This indicates that the difference in the parameters of $\hat{f}$ and f is governed by the Lipschitz constant L and the learning rate η, suggesting that the parameters should remain close throughout the training process, especially when C is negligible.

C.3.6 Performance Bound

We would like to find a performance bound between a trained sub-model (with optimized parameters θ^*) and its corresponding individual model (with optimized parameters ϕ^*). Let's assume that $\phi^* = \theta^* + \Delta\theta$. Then, the performance bound can be calculated as $\Delta f = f(x;\phi^*) - \hat{f}(x;\theta^*)$ in the function value from its optimal value due to a parameter perturbation

Step 1: Second-Order Taylor Expansion Applying the second order taylor expansion to the function $f(x;\phi)$ around ($\phi = \phi^*$), we get:

$$
\begin{aligned}
&f(x;\phi^*) = f(x;\theta^* + \Delta\theta) \approx \\
&f(x;\theta^*) + \nabla_\theta f(x;\theta^*)^T \Delta\theta + \frac{1}{2}\Delta\theta^T H(x;\theta^*)\Delta\theta = \\
&\hat{f}(x;\theta^*) + \nabla_\theta \hat{f}(x;\theta^*)^T \Delta\theta + \frac{1}{2}\Delta\theta^T \hat{H}(x;\theta^*)\Delta\theta.
\end{aligned}
$$

Bear in mind that $f(x;\theta) = \hat{f}(x;\theta)$ and $H(x;\theta^*)$ refers to the Hessian matrix of f at $(\theta = \theta^*)$.

Step 2: Optimum Condition

$$
\nabla_\theta \hat{f}(x;\theta^*) = 0
$$

$$
\Rightarrow \hat{f}(x;\phi^*) \approx \hat{f}(x;\theta^*) + \frac{1}{2}\Delta\theta^T H(x;\theta^*)\Delta\theta
$$

Step 3: Lipschitz Continuity of Gradient

$$
\|\nabla_\theta \hat{f}(x;\theta) - \nabla_\theta \hat{f}(x;\theta')\| \leq L\|\theta - \theta'\|
$$

Step 4: Bounding the Hessian

$$
\|\Delta\theta^T \hat{H}(x;\theta)\Delta\theta\| \leq L\|\Delta\theta\|^2
$$

Step 5: Estimating the Deviation in Function Value

$$
\begin{aligned}
&\Delta f = f(x;\phi^*) - \hat{f}(x;\theta^*) \\
&\Delta f \approx \frac{1}{2}\Delta\theta^T \hat{H}(x;\theta)\Delta\theta \leq \frac{1}{2}L\|\Delta\theta\|^2
\end{aligned}
$$

The deviation Δf in the function value from its optimal value due to a parameter perturbation $\Delta\theta$ is bounded by $\frac{1}{2}L\|\Delta\theta\|^2$ under the assumption of L-Lipschitz continuity of the gradient. This result implies that the function value's deviation grows at most quadratically with the size of the parameter perturbation.

C.4 More Experimental Details

C.4.1 Effect of gradient accumulation on SortedNet performance

It is of interest to explore whether limiting the number of parameter updates is a suitable approach for investigating the influence of gradient accumulation on SortedNet. One possible way to certify this factor is by running SortedNet with different gradient accumulation values while keeping the number of updates fixed. To that end, we consider the same settings as Table 4.5 and repeat the experiment while fixing the maximum number of training epochs. By fixing this value and increasing gradient accumulation values, we implicitly decrease the number of parameter updates. Table C.1 reports the results. Comparing the results of these two tables, it is obvious that the number of updates plays a significant role in the model's performance. For instance, when considering $g_{acc} = 2$, an average performance drop of approximately 2% is observed across all sub-models. This reduction indicates that the underlying model needs more training time for higher values of g_{acc}.

Table C.1: Exploring the impact of limited number of parameters updates on the effect of gradient accumulation in SortedNet-IN. The underlying network and dataset are cPreResNet20 and CIFAR10, respectively.

Grad. Accum.	Num. Updates	Epochs	Accuracy @ Width				Avg.
			100%	75%	50%	25%	
$g_{acc} = 1$	200	200	84.94	84.92	82.54	71.03	80.85
$g_{acc} = 2$	100	200	85.01	85.12	82.24	70.65	80.75
$g_{acc} = 3$	66	200	85.09	85.06	82.64	73.74	81.63
$g_{acc} = 4$	50	200	86.05	86.06	83.66	73.0	82.19

C.4.2 Hyperparameters

This section provides an overview of the hyperparameters and experimental configurations, detailed in Table C.2.

C.4.3 Details of training time comparison

To empirically compare the training time between SortedNet and LCS_p, the elapsed time per epoch for five epochs is recorded independently for each method. We then ignore the first epoch to reduce the impact of first-time loading and initialization. Next, for each method we take the average of the remaining elapsed times. We refer to these averaging times (in seconds) by $\bar{T}_{SortedNet} = 49.7 \pm 2.06$ and $\bar{T}_{LCS\_p} = 292.7 \pm 3.17$ for simplicity. As it is mentioned in Subsection 4.5.6, SortedNet with $g_{acc} = 4$ can be considered as a fair comparison with LCS_p. As a result, each epoch in LCS_p holds four times the significance of SortedNet in terms of the

Table C.2: All the hyperparameters that have been used throughout our study for different experiments. If we didn't mention a parameter specifically, it means we utilized the default value of the HuggingFace Transformers v'4.27.0.dev0'. Otherwise, we highlighted any exception in the main text.

Model	**Parameter**	**Value**
BERT-Base	Optimizer	AdamW
	Warmup Ratio	0.06
	Dropout	0.1
	LR Scheduler	Linear
	Batch Size	32 (RoBertA) / 8 (Bert)
	Epochs	30 (RoBertA) / 3,6 (Bert)
	Learning Rate (LR)	2e-5 (RoBertA / 6e-6 (Bert)
	Weight Decay	0.1
	Max Sequence Length	512
	Seeds	[10, 110, 1010, 42, 4242]
	GPU	Tesla V100-PCIE-32GB
MobileNetV2	Model	"google/mobilenet_v2_1.4_224"
	Optimizer	AdamW
	LR Scheduler	Linear
	Batch Size	128
	Seeds	4242
	Epochs	60 $\times \#$ Models
	GPU	8 × Tesla V100-PCIE-32GB
cPreResNet20	Optimizer	SGD
	Criterion	Cross Entropy
	LR Scheduler	cosine_lr
	Batch Size	128
	Seeds	[40,42,1010,4242]
	Momentum	0.9
	Weight Decay	0.0005
	LR	0.1
	Epochs	[200,400,600,800]
	Gradient Accumulation	[1,2,3,4]

total number of parameter updates. Therefore, we can simply multiply $\bar{T}_{SortedNet}$ by a factor of four to equalize their impacts in term of total number of parameter updates. By doing that, we have $\bar{T}_{SortedNet} = 198.8$, which is almost one-third less than $\bar{T}_{LCS\_p}$.

C.4.4 Can we improve the performance of SortedNet by adjusting the classifier layer?

In Figure 4.2, as mentioned before, we adjusted the performance of the classifiers for the baseline in the experiment but not for the SortedNet. Therefore, as an additional experiment, we wanted to analyze the impact of adjusting the classifier over the performance of our proposed method as well. Same as the previous experiment, we trained the classifier layer for 5 epochs for each sub-model and reported the performance. As shown in Figure C.3, the gain is much higher for very smaller networks than the large ones. The SortedNet shared classifier already doing a good job without additional computation overhead for all sub-models but further adjustments might be beneficial as shown.

C.4.5 Can we extend SortedNet to complex dimensions?

Table C.3: The performance of BERT-base and Bert-large in the GLUE Benchmark over 5 runs for SortedNet (sharing weights across both models), pre-trained berts and different initialization.

Model	Flops	Weights	Acc. MNLI	F1 QQP	Acc. QNLI	Acc. SST-2	Matthews Corr. CoLA	Spearman Corr. STS-B	F1 MRPC	Acc. RTE	avg.
Random Initialized Networks											
$BERT_{BASE}(3\mathcal{L}_B)$	78.96 G	W_B	62.13 ± 0.27	68.74 ± 0.70	61.24 ± 0.45	79.89 ± 0.59	0.00 ± 0.00	12.92 ± 0.63	78.67 ± 0.41	54.51 ± 1.11	52.26
pre-trained Baselines											
$BERT_{BASE}(3\mathcal{L}_B)$*	22.36 G	W_B	84.0	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.5
$BERT_{LARGE}(3\mathcal{L}_L)$*	78.96 G	W_L	86.3	72.1	92.7	94.9	60.5	86.5	89.3	70.1	81.55
Paper Setting											
$BERT_{BASE}(3\mathcal{L}_B)$	22.36 G	W_B	84.22 ± 0.32	87.37 ± 0.08	91.47 ± 0.21	92.61 ± 0.15	54.90 ± 0.79	88.08 ± 0.49	86.91 ± 0.82	62.96 ± 2.36	**81.07**
$BERT_{LARGE}(3\mathcal{L}_L)$	78.96 G	W_L	86.32 ± 0.09	88.36 ± 0.07	92.01 ± 0.29	93.21 ± 0.42	59.39 ± 1.45	88.65 ± 0.33	88.67 ± 0.75	68.23 ± 1.59	83.11
Extracted Networks											
$BERT_{BASE}^{LARGE}(3\mathcal{L}_B)$	22.36 G	W_B	77.43 ± 0.08	84.88 ± 0.15	84.74 ± 0.34	84.98 ± 0.47	12.17 ± 1.62	78.33 ± 4.11	79.44 ± 0.93	55.23 ± 1.08	69.65
Proposed Methods											
Sorted $BERT_{BASE}(\sim 1.5\mathcal{L}_B + 1.5\mathcal{L}_L)$	22.36 G	W_B^L	76.20 ± 0.02	83.58 ± 0.16	83.91 ± 0.18	83.26 ± 0.69	0.08 ± 0.18	70.75 ± 9.25	80.75 ± 1.29	52.85 ± 2.53	66.42
Sorted $BERT_{LARGE}(\sim 1.5\mathcal{L}_B + 1.5\mathcal{L}_L)$	78.96 G	W_L^L	85.93 ± 0.33	87.28 ± 0.14	91.58 v 0.33	93.17 ± 0.26	57.08 ± 1.91	88.18 ± 0.68	87.06 ± 1.02	65.56 ± 1.41	81.98
Sorted $BERT_{BASE}(\sim 3\mathcal{L}_B + 3\mathcal{L}_L)$	22.36 G	W_B^L	77.48	85.16 ± 0.02	84.96 ± 0.23	86.01 ± 0.62	12.58 ± 2.04	79.29 ± 2.80	78.96 ± 0.44	55.81 ± 1.37	70.03
Sorted $BERT_{LARGE}(\sim 3\mathcal{L}_B + 3\mathcal{L}_L)$	78.96 G	W_L^L	86.12	88.26 ± 0.01	92.18 ± 0.28	93.49 ± 0.21	59.84 ± 1.35	88.85 ± 0.44	88.88 ± 1.10	68.45 ± 2.11	**83.26**

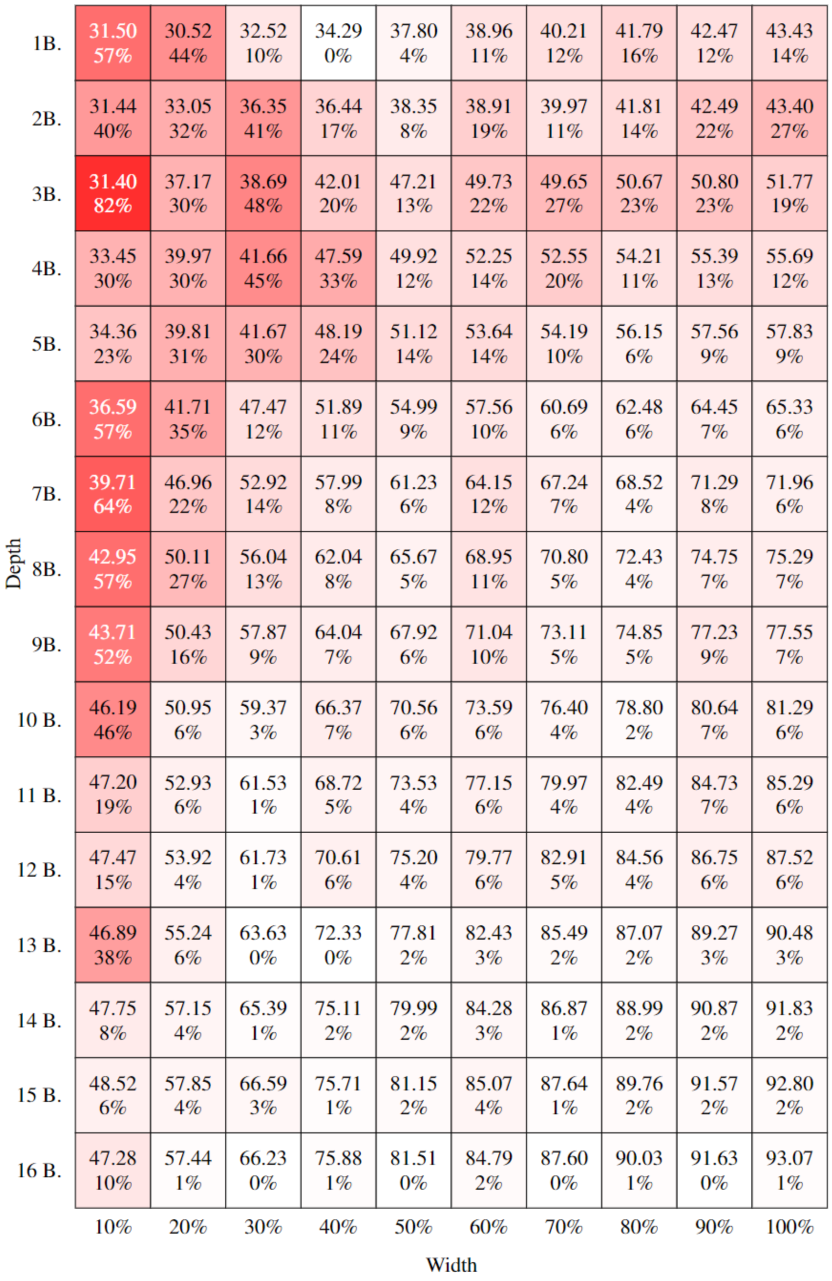
In this section, we are interested to investigate whether SortedNet is applicable to more complex dimensions other than width and depth. For example, can we utilize the SortedNet for sorting the Attention Heads [106]? To achieve this, we conducted an experiment over BERT-large [21] which we tried to sort the information across multiple dimensions at once including, number of layers, hidden dimension, and number of attention heads. In other words, we tried to sort information over Bert-large and Bert-base as Bert-base can be seen as a subset of the

Bert-large and therefore respect the nested property. As reported in Table C.3, in addition to the reported performance of Bert-base and Bert-large according to the original paper [21], we reported the performance of these models in the paper experimental setting. The performance of randomly initialized Bert-base has been reported as well. We also extracted a Bert-base from a Bert-large model, and we reported the performance of such model in the same table. Additionally, we highlighted the number of training updates with respect to each objective function in front of each model. For example, in the last row (Sorted $BERT_{LARGE}$), we approximately trained our Sorted model half of the times ($\sim 3Epochs$) over the objective function of Bert-base ($\mathcal{L}_B$) and the other half of the times over the objective function of Bert-large ($\mathcal{L}_L$) in an iterative random manner as introduced in the section 4.4. The learned Bert-base performance with these methods is still around 10% behind a pre-trained base but we argue that this is the value of pre-training. To investigate the impact, one should apply the SortedNet during pre-training which we will leave for future research. However, the performance of the learned Bert-large is on-par with an individual Bert-large which suggests sharing the weights does not necessarily have a negative impact over learning. It seems, however, the secret sauce to achieve a similar performance is that we should keep the number of updates for each objective the same as the individual training of Bert-large and Bert-base.

C.4.6 What is the impact of Sorting?

In order to better understand the impact of sorting information, we designed an experiment that compare the dependency order of all the neurons in a sorted network. To keep the experiment simple, we designed a one layer neural network with 10 (hidden dimension) $\times$ 2 (input dimension) neurons as the hidden layer and a classifier layer on top of that which map the hidden dimension to predict the probabilities of a 4 class problem. The task is to predict whether a 2d point belong to a specific class on a synthetic generated dataset. We trained both Sorted Network and the ordinary one for 10 epochs and optimize the networks using Adam optimizer [51] with the learning rate of 0.01 and batch size of 16.

As can be seen in Table C.4, the performance of different orders in the original neural network training paradigm can be different and unfortunately there is no specific pattern in it. Therefore, if one search the whole space of different orders (from neuron 1 to neuron n, from neuron n to neuron 1, or even select a subset of neurons by different strategies i.e. for the half size network activate every other neurons like XOXOXOXOXO.) might find better alternatives that work even better than the desirable target order. In this example, the reverse order in average perform better than the target order (86.67% versus 82.22%). However, with the proposed method, we can clearly see that the target order performance consistently is much better than the reverse order (89.25% versus 59.38%). This means, we have been able to enforce the desirable target order as we wanted using our proposed method. For example, neuron 2 is more dependent to neuron 1 in SortedNet in comparison with the ordinary training. In another example, the last 5 neurons are more dependent to the first 5 neurons than other way around. As shown, the performance of the first five neurons is 93.86% while the performance of the last five neurons is only 66.06% in SortedNet. In other words, the gain of adding the last five neurons is quite marginal and most probably prunnable, while the first 5 neurons contains most of the valuable information. It is of interest to further investigate the dependency of neurons to one another and with other metrics which we will leave for future research.



	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
1B.	31.50 57%	30.52 44%	32.52 10%	34.29 0%	37.80 4%	38.96 11%	40.21 12%	41.79 16%	42.47 12%	43.43 14%
2B.	31.44 40%	33.05 32%	36.35 41%	36.44 17%	38.35 8%	38.91 19%	39.97 11%	41.81 14%	42.49 22%	43.40 27%
3B.	31.40 82%	37.17 30%	38.69 48%	42.01 20%	47.21 13%	49.73 22%	49.65 27%	50.67 23%	50.80 23%	51.77 19%
4B.	33.45 30%	39.97 30%	41.66 45%	47.59 33%	49.92 12%	52.25 14%	52.55 20%	54.21 11%	55.39 13%	55.69 12%
5B.	34.36 23%	39.81 31%	41.67 30%	48.19 24%	51.12 14%	53.64 14%	54.19 10%	56.15 6%	57.56 9%	57.83 9%
6B.	36.59 57%	41.71 35%	47.47 12%	51.89 11%	54.99 9%	57.56 10%	60.69 6%	62.48 6%	64.45 7%	65.33 6%
7B.	39.71 64%	46.96 22%	52.92 14%	57.99 8%	61.23 6%	64.15 12%	67.24 7%	68.52 4%	71.29 8%	71.96 6%
8B.	42.95 57%	50.11 27%	56.04 13%	62.04 8%	65.67 5%	68.95 11%	70.80 5%	72.43 4%	74.75 7%	75.29 7%
9B.	43.71 52%	50.43 16%	57.87 9%	64.04 7%	67.92 6%	71.04 10%	73.11 5%	74.85 5%	77.23 9%	77.55 7%
10 B.	46.19 46%	50.95 6%	59.37 3%	66.37 7%	70.56 6%	73.59 6%	76.40 4%	78.80 2%	80.64 7%	81.29 6%
11 B.	47.20 19%	52.93 6%	61.53 1%	68.72 5%	73.53 4%	77.15 6%	79.97 4%	82.49 4%	84.73 7%	85.29 6%
12 B.	47.47 15%	53.92 4%	61.73 1%	70.61 6%	75.20 4%	79.77 6%	82.91 5%	84.56 4%	86.75 6%	87.52 6%
13 B.	46.89 38%	55.24 6%	63.63 0%	72.33 0%	77.81 2%	82.43 3%	85.49 2%	87.07 2%	89.27 3%	90.48 3%
14 B.	47.75 8%	57.15 4%	65.39 1%	75.11 2%	79.99 2%	84.28 3%	86.87 1%	88.99 2%	90.87 2%	91.83 2%
15 B.	48.52 6%	57.85 4%	66.59 3%	75.71 1%	81.15 2%	85.07 4%	87.64 1%	89.76 2%	91.57 2%	92.80 2%
16 B.	47.28 10%	57.44 1%	66.23 0%	75.88 1%	81.51 0%	84.79 2%	87.60 0%	90.03 1%	91.63 0%	93.07 1%

Figure C.3: CIFAR10 Adjusted Classification Accuracy for SortedNet (160 Models) and the baseline. The relative performance gain of each sub-model has been reported at the bottom of each cell with respect to the performance of the same network without adjustment. More white the better.

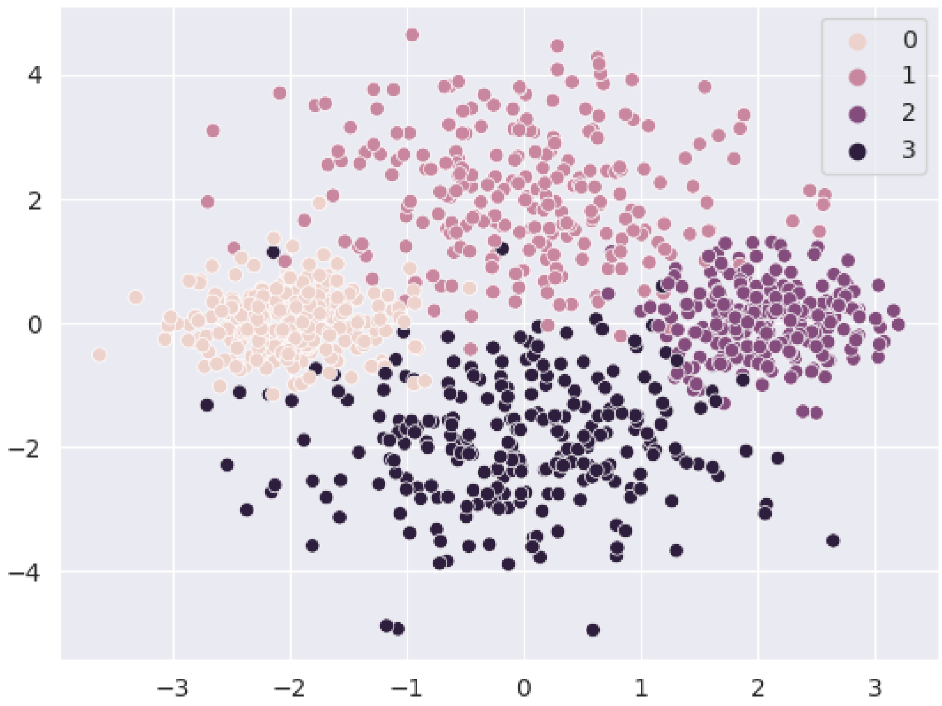


Figure C.4: Synthetically generated dataset with four classes and with the centers of [[-2, 0], [0, 2], [2, 0], [0, -2]] and cluster standard deviation of [0.5, 1, 0.5, 1]. Seed has been fixed to 42, and 1000 samples has been generated.

Table C.4: Order dependency of all neurons in the network using the proposed method (SortedNet) and the ordinary training across 5 random runs. X means we used the neuron as is, and O means we removed the impact of that neuron by making it 0. ↑ higher the better, ↓ the lower the better.

Active Neurons	Network Accuracy	SortedNet Accuracy
Baseline		
XXXXXXXXXX XXXXXXXXXX	94.48 ± 0.13	93.7 ± 0.38
OOOOOOOOOO OOOOOOOOOO	25 ± 0	25 ± 0
Target Order ↑		
XXXXXXXXXO XXXXXXXXXO	93.86 ± 0.31	93.64 ± 0.48
XXXXXXXXOO XXXXXXXXOO	93.7 ± 0.76	93.78 ± 0.37
XXXXXXXOOO XXXXXXXOOO	93.1 ± 0.72	93.62 ± 0.53
XXXXXXOOOO XXXXXXOOOO	92.34 ± 0.99	93.7 ± 0.77
XXXXXOOOOO XXXXXOOOOO	90.38 ± 1.78	93.86 ± 0.65
XXXXOOOOOO XXXXOOOOOO	86.84 ± 8.08	93.66 ± 0.81
XXXOOOOOOO XXXOOOOOOO	80.14 ± 10.78	93.14 ± 1.53
XXOOOOOOOO XXOOOOOOOO	60.96 ± 14.09	85.26 ± 13.77
XOOOOOOOOO XOOOOOOOOO	48.66 ± 1.5	62.58 ± 7.77
avg.	82.22 ± 15.49	**89.25** ± **9.79**
Reverse Order ↓		
OOOOOOOOOX OOOOOOOOOX	51.7 ± 3.95	29.84 ± 10.34
OOOOOOOOXX OOOOOOOOXX	85.52 ± 9.77	35.18 ± 25.32
OOOOOOOXXX OOOOOOOXXX	90.82 ± 1.79	41.52 ± 24.42
OOOOOOXXXX OOOOOOXXXX	86.7 ± 3.79	59.88 ± 13.81
OOOOOXXXXX OOOOOXXXXX	90.92 ± 2.52	66.06 ± 19.19
OOOOXXXXXX OOOOXXXXXX	92.94 ± 0.54	65.52 ± 19.35
OOOXXXXXXX OOOXXXXXXX	93.52 ± 0.26	72.14 ± 16.89
OOXXXXXXXX OOXXXXXXXX	93.84 ± 0.4	82.06 ± 11.24
OXXXXXXXXX OXXXXXXXXX	94.04 ± 0.24	82.2 ± 13.3
avg.	86.67 ± 12.7	**59.38** ± **18.43**

Appendix D

SortedLLama

D.0.1 Experimental Setup

We used the pre-trained LLaMA2 13b weights, publicly available on Hugging Face, as our starting point. For SFT+ICT (Early-Exit) setup, we froze the parameters of the transformer blocks and only further trained the weights of the language model head classifier for one additional epoch. We used a batch size of 32 and gradient accumulation of 8. The learning scheduler was cosine annealing. The learning rate was set to 2e-5 and seed to 42. We trained the models on 8 V100 32GB GPUs. The same GPUs were used during inference time. The training maximum input sequence length was 2024, with a maximum of 50 (TriviaQA) and 256 (PandaLM) generated tokens during inference. Additionally, we used greedy search as the decoding strategy in all of our experiments. We also extended the huggingface assisted decoding code to implement Speculative Decoding and Instance-Aware Adaptive Inference. In Speculative Decoding, we used adaptive K window-size (the same as huggingface) starting with K=4. In Instance-Aware Dynamic Inference, we set the confidence thresholds of intermediate layers as follow: Layer 12 = 0.95, Layer 16 = 0.95, Layer 20 = 0.9, Layer 24 = 0.9, Layer 28 = 0.8, Layer 32 = 0.8 and Layer 36 = 0.7.

D.0.2 Computational Overhead of SoFT

Given the nested pattern of sub-models and the fact that we share the language model head across sub-models, we do not expect to see any computation overhead for SoFT versus SFT. To validate this claim, we compared SoFT and SFT regarding training time and memory usage in our experiment on the Alpaca dataset. Table D.1 shows the result for two main experiments of

Method	Avg Time per Epoch (s)	Avg Memory Usage per Epoch (MB)
SFT	25,765.95	99,168
SoFT	25,269.87 (0.98×)	125,682

Table D.1: Training Time and Memory Usage comparison of SoFT and SFT on Alpaca dataset.

SoFT and SFT. As expected, training with SoFT leads to equal training time compared to SFT. During training, SoFT has about 25% memory overhead in PyTorch compared to SFT, which only provides a single full model at the end.

Case Specific Analysis

Table 5.1 shows a sample of instructions from the PandaLM benchmark and the generated responses by SFT+ICT (Early-Exit) and Sorted LLaMA sub-models. Sorted LLaMA performs better in preserving and transferring the last layer performance to earlier sub-models based on the information made visible by black (related to the query) and red (hallucinations, irrelevant, etc.) colors.

Sorted sub-models generate almost correct answers from the 20 layers sub-model, while the first meaningful result from SFT+ICT sub-models appears in layer 28. Other samples generated by SoFT and Early-Exit can be found in D.0.4.

D.0.3 Additional Experiments

Table D.2 shows the detailed results of the Sorted LLaMA and SFT performance on the PandaLM benchmark in different setup in equal training time (2 Epochs for both SFT and SoFT). As we can see, sorted sub-models outperform their SFT counterparts (and even higher sub-models) , while in SFT+ICT (Early-Exit), as we go higher in sub-models (e.g. layer 36), we can see a noticeable improvement in the performance compared to the SFT. This can demonstrate the importance of tuning the language model classifier in improving text generation capability in the latest layers in the standard fine-tuning format.

Table D.3 shows the SoFT and SFT comparison in a different training time setup in which SoFT has access to doubled training time (4 Epochs). Results show that Sorted LLaMA can outperform standard fine-tuned LLaMA further by continuing the SoFT process. The improvement in Sorted LLaMA sub-models performance can be observed specifically in intermediate layers.

D.0.4 Analysis

Table D.4 and D.5 show some samples generated by sub-models of LLaMA (SFT+ICT) and SoFT on PandaLM evaluation set. In the first query of Table D.4, LLaMA sub-models until layer 36 struggle to generate relevant responses about books in the Crime and Mystery genre. Sorted LLaMA sub-models, however, start to address the related novels from layer 24. The second query in the table is a simpler instruction, which is a multi-label classification problem. Again Sorted LLaMA sub-models start to generate the correct label in much earlier layers (layer 20) compared to the LLaMA sub-models (layer 24). Table D.5 first example shows the performance gap of the LLaMA and Sorted LLaMA intermediate sub-models even in a more severe case. To write a review about a restaurant with certain aspects, LLaMA sub-models before layer 32 hallucinate or generate gibberish, while Sorted LLaMA starts to generate a complete review addressing key points mentioned in the instruction even in the first sub-model (layer 16). In the second example, the same pattern occurs where SoFT sub-models can generate meaningful response starting from layer 16 while LLaMA first reasonable text happens at layer 36.

Table D.6 shows an example of SFT and SoFT performance on TriviaQA benchmark. While LLaMA struggles to generate single answer token even in the sub-models close to the last layer, SoFT could transfer the question answering capability of LLM until sub-layer 20 and still generate the correct final answer.

After all, Sorted LLaMA sub-models demonstrate the ability to generate more comprehensive (Example 1 of Table D.4 and Table D.5) and informative (Table D.4 example 2) answers in earlier layers compared to LLaMA. Based on our observation, LLaMA sub-models mostly tend to generate irrelevant or even gibberish in earlier blocks (layers 12 to 24), while the generated texts by Sorted LLaMA exhibit sufficient learned information to answer the input instruction despite having much fewer parameters.

Sorted LLaMA/LLaMA	12 (4.1B)	16 (5.4B)	20 (6.6B)	24 (7.9B)	28 (9.2B)	32 (10.4B)	36 (11.7B)	40 (13B)
SoFT vs. SFT								
12 (4.1B)	71.0/99.0/0.0	97.5/72.5/0.0	129.0/41.0/0.0	131.0/39.0/0.0	121.5/48.5/0.0	106.5/63.5/0.0	45.0/125.0/0.0	17.0/152.5/0.5
16 (5.4B)	81.0/89.0/0.0	101.5/68.5/0.0	128.5/40.5/1.0	131.5/38.0/0.5	124.0/44.5/1.5	114.0/54.0/2.0	52.0/114.0/4.0	18.0/146.0/6.0
20 (6.6B)	111.5/58.5/0.0	132.0/38.0/0.0	144.5/23.5/2.0	147.5/20.5/2.0	141.5/24.0/4.5	132.5/30.5/7.0	73.5/85.5/11.0	32.5/114.0/23.5
24 (7.9B)	124.5/45.5/0.0	136.5/33.5/0.0	150.0/18.0/2.0	154.5/13.5/2.0	148.0/18.5/3.5	144.5/19.0/6.5	98.0/62.0/10.0	44.5/90.0/35.5
28 (9.2B)	125.5/44.5/0.0	145.0/25.0/0.0	153.0/15.0/2.0	153.5/14.5/2.0	148.0/16.5/5.5	143.5/20.5/6.0	96.5/59.5/14.0	45.0/89.0/36.0
32 (10.4B)	141.5/28.5/0.0	152.0/18.0/0.0	159.0/9.0/2.0	160.0/8.0/2.0	152.0/12.5/5.5	150.5/13.5/6.0	108.5/45.0/16.5	55.5/75.0/39.5
36 (11.7B)	141.0/28.5/0.5	152.5/17.0/0.5	159.0/8.5/2.5	161.5/6.5/2.0	150.0/14.5/5.5	148.5/15.5/6.0	112.0/42.5/15.5	53.0/66.0/51.0
40 (13B)	143.5/26.5/0.0	156.0/14.0/0.0	160.5/7.5/2.0	161.0/7.0/2.0	150.0/14.0/6.0	150.0/13.5/6.5	115.5/39.0/15.5	52.5/62.5/55.0
SoFT vs. SFT+ICT(Early-Exit)								
12 (4.1B)	75.0/95.0/0.0	108.5/61.5/0.0	128.5/41.5/0.0	122.5/47.5/0.0	116.5/53.5/0.0	91.0/79.0/0.0	37.5/131.5/1.0	17.0/152.5/0.5
16 (5.4B)	86.5/82.5/1.0	113.0/57.0/0.0	127.0/41.0/2.0	122.0/47.0/1.0	117.5/50.5/2.0	94.5/72.0/3.5	36.0/129.0/5.0	18.0/146.0/6.0
20 (6.6B)	111.5/57.5/1.0	137.0/33.0/0.0	143.5/24.0/2.5	143.0/23.0/4.0	137.0/27.0/6.0	122.0/38.0/10.0	60.0/94.5/15.5	32.5/114.0/23.5
24 (7.9B)	126.5/42.5/1.0	144.0/26.0/0.0	149.0/19.5/1.5	151.0/15.5/3.5	143.0/21.5/5.5	133.5/28.0/8.5	76.5/72.5/21.0	44.5/90.0/35.5
28 (9.2B)	130.0/39.0/1.0	147.0/23.0/0.0	153.5/15.5/1.0	150.0/16.0/4.0	143.5/18.5/8.0	131.0/29.0/10.0	79.0/66.0/25.0	45.0/89.0/36.0
32 (10.4B)	141.5/27.5/1.0	155.5/14.5/0.0	161.0/8.0/1.0	157.0/8.5/4.5	151.0/11.0/8.0	143.5/15.0/11.5	89.5/49.5/31.0	55.5/75.0/39.5
36 (11.7B)	143.0/25.5/1.5	156.5/13.0/0.5	160.0/8.5/1.5	157.0/8.5/4.5	148.0/14.0/8.0	142.5/16.5/11.0	92.5/46.5/31.0	53.0/66.0/51.0
40 (13B)	146.0/23.0/1.0	157.0/13.0/0.0	160.5/7.5/2.0	157.5/9.0/3.5	149.0/14.0/7.0	143.5/16.0/10.5	97.5/43.5/29.0	52.5/62.5/55.0

Table D.2: Pair-wise comparison for different layers (sub-models) in Standard Fine-Tuning and SoFT at equal training cost (2 Epochs). Each cell consists of three values: Wins, Losses, Ties. Wins demonstrate the number of times that the generated text of the sub-model in row (sorted) is preferred to the sub-model in column (Fine-Tuned) and Losses is the opposite. Numbers are average of two separate experiments with different order of inputs to evaluator in order to neutralize the order bias.

Sorted LLaMA/LLaMA	12 (4.1B)	16 (5.4B)	20 (6.6B)	24 (7.9B)	28 (9.2B)	32 (10.4B)	36 (11.7B)	40 (13B)
SoFT vs. SFT								
12 (4.1B)	88.5/81.5/0.0	108.0/62.0/0.0	134.5/35.5/0.0	135.0/35.0/0.0	129.0/41.0/0.0	120.0/49.0/1.0	57.0/109.5/3.5	23.5/144.0/2.5
16 (5.4B)	106.5/63.0/0.5	120.0/50.0/0.0	140.0/29.0/1.0	144.5/24.5/1.0	142.0/26.5/1.5	136.0/32.0/2.0	70.0/95.0/5.0	34.5/124.5/11.0
20 (6.6B)	127.0/43.0/0.0	138.5/31.5/0.0	151.5/16.5/2.0	152.0/17.0/1.0	143.5/23.5/3.0	144.0/21.5/4.5	94.5/67.5/8.0	47.0/99.5/23.5
24 (7.9B)	138.5/31.5/0.0	149.5/20.5/0.0	159.0/9.0/2.0	158.0/10.5/1.5	151.5/13.5/5.0	149.0/15.5/5.5	107.0/49.5/13.5	53.0/81.0/36.0
28 (9.2B)	137.0/33.0/0.0	149.0/21.0/0.0	158.0/10.0/2.0	159.5/8.5/2.0	150.0/15.0/5.0	149.5/15.0/5.5	107.0/47.5/15.5	50.5/78.0/41.5
32 (10.4B)	146.0/24.0/0.0	157.0/13.0/0.0	163.0/5.0/2.0	163.0/5.0/2.0	154.5/10.5/5.0	151.5/12.5/6.0	117.5/37.5/15.0	63.5/62.0/44.5
36 (11.7B)	149.5/20.5/0.0	160.0/10.0/0.0	164.0/4.0/2.0	162.5/5.5/2.0	157.5/7.5/5.0	154.0/10.0/6.0	119.5/34.5/16.0	62.5/60.0/47.5
40 (13B)	153.5/16.5/0.0	163.0/7.0/0.0	165.5/3.0/1.5	163.5/4.5/2.0	157.0/8.0/5.0	156.0/8.5/5.5	121.0/33.5/15.5	67.5/52.0/50.5
SoFT vs. SFT+ICT(Early-Exit)								
12 (4.1B)	91.5/77.5/1.0	123.5/46.5/0.0	138.5/31.5/0.0	134.0/36.0/0.0	130.5/39.0/0.5	107.5/59.0/3.5	46.0/120.5/3.5	23.5/144.0/2.5
16 (5.4B)	106.5/63.5/0.0	128.5/41.0/0.5	145.0/24.0/1.0	144.5/25.0/0.5	139.0/29.5/1.5	122.5/43.0/4.5	55.5/106.5/8.0	34.5/124.5/11.0
20 (6.6B)	128.0/40.5/1.5	142.0/27.5/0.5	152.5/16.0/1.5	148.0/19.0/3.0	142.0/22.5/5.5	131.0/31.0/8.0	74.0/79.0/17.0	47.0/99.5/23.5
24 (7.9B)	140.5/28.5/1.0	155.0/15.0/0.0	159.0/10.0/1.0	156.0/10.5/3.5	149.5/12.5/8.0	141.5/19.0/9.5	90.0/54.5/25.5	53.0/81.0/36.0
28 (9.2B)	142.0/27.0/1.0	155.5/14.5/0.0	161.5/7.5/1.0	155.5/10.5/4.0	148.5/13.0/8.5	141.0/19.0/10.0	91.5/53.5/25.0	50.5/78.0/41.5
32 (10.4B)	150.0/19.0/1.0	161.0/9.0/0.0	161.5/7.5/1.0	160.0/6.0/4.0	153.0/8.5/8.5	145.0/14.0/11.0	99.0/41.0/30.0	63.5/62.0/44.5
36 (11.7B)	151.0/18.0/1.0	163.5/6.5/0.0	165.0/3.5/1.5	161.0/5.0/4.0	155.0/7.5/7.5	147.5/12.0/10.5	105.5/36.0/28.5	62.5/60.0/47.5
40 (13B)	156.0/13.0/1.0	165.5/4.5/0.0	165.0/3.5/1.5	161.0/5.0/4.0	155.0/7.5/7.5	150.5/10.0/9.5	106.0/35.0/29.0	67.5/52.0/50.5

Table D.3: Pair-wise comparison for different layers (sub-models) in Standard Fine-Tuning and SoFT (4 Epochs) after tuning Language Model classifier for each SFT sub-model.

Query: Choose the best books from the given genre. Crime & Mystery

Layer	**LLaMA Response (SFT + ICT)**	**Sorted LLaMA Response (SoFT)**
12	Therewiguptaranapped byoremandoesleven years ago agomairiensum...	Here are some of the best books from the genre: 1. The Fear of the unknown: The Fist of the Darkness2. The Fist of the Deaths of the Deaths of the Deaths of the Deaths of the Deaths of the Deaths ...
16	Here list list list!!!!!!!!!!!!!!!0̆0e4ndersonally preferences personal ...	Here is a list of top 10 books from the genre Crime & Mystery that you can read for your next reading book list: 1. The Great mystery of the Mystery of the Mystery of the Mystery of the Mystery of the ...
20	Here listicle lists some best books genreed intoxicatedly intoxicatedly ...	Here are some popular and well-known crime and mystery books that you can choose from: 1. The Secret of the Mystery solved by the detective by Jameson Jameson is a classic crime and mystery novel that is widely regarded as one of the best crime and mystery books ever written. 2. The Secret of the Mystery solved by the detective by Jameson Jameson is a classic crime and mystery novel that is widely regarded as one of the best crime and mystery books ever written. 3. The Secret of the Mystery solved ...
24	Here list contains some best books belonging belongs genre crime & mystery genre ordered alphabetically alphabetically ...	Here are some of the best crime and mystery books: 1. The Hatchery by Stephen King - This book is a classic horror/ crime thriller that follows the story of a group of friends who discover a hidden hatchery of horrors. 2. The Last Thistle by Murphy Murphy - This book is a thriller that follows the investigation of a murder in a small town. The story is filled with suspense and twists that will keep you hooked. 3. The Last Thistle by Murphy Murphy - This book is a thriller that follows the investigation of a murder in a small town. The story is filled with suspense and twists that will keep you hooked. 4. The Last Thistle by Murphy ...
28	crime mystery books best crime mystery books crime mystery books best crime mystery ...	Here are some of the best crime and mystery books that you might enjoy: 1. The Great Gatsby by F. Scott Fitzgerald - This classic novel is a classic tale of love, betrayal, and the dark side of the American Dream. 2. The Great Gatsby by F. Scott Fitzgerald - This classic novel is a classic tale of love, betrayal, and the dark side of the American Dream. 3. The Great Gatsby ...
32	Crime mystery genre has many great books to choose from. Here are some recommendations: crime mystery books recommended by readers: 1) crime mystery books recommended by readers: Crime mystery genre has many great books to choose from. Here are some recommendations: crime mystery books ...	Crime & Mystery is a popular genre, and there are many great books that fall into this category. Here are some of the best crime and mystery books that have stood the test of time and are widely acclaimed by readers and critics alike: 1. The Hound of the Baskervilles by Sir Arthur Conan Doyle: This classic detective novel features the iconic detective Sherlock Holmes and his sidekick, Dr. John Watson. The story follows Holmes as he investigates the mysterious death of a wealthy man and the disappearance of his family. 2. The Silent Patient by Sarah C. Dessen: This psychological thriller follows the story of a woman who is institutionized after shooting her husband. The novel explores themes of trauma, mental illness, and the power of love to heal. 3. The Silent majority by Agatha Christie: This classic mystery novel features the beloved detective Hercule Poirot as he investigates the murder of a wealthy man. The novel is full of twists and turns, and the identity of the murderer is not revealed until the end.
36	Crime fiction is a genre that has captivated readers for centuries with intriguvity and suspenseful plots. Here are some of the best crime books that you should definitely read: 1. Sherlock Holmes series by Sir Arthur Conan Doyle: regarded as one of the greatest fictional detectives of all time, Sherlock Holmes is known for his deductive reasoning and ability to solve complex cases. 2. Agatha Christie novels: regarded as the queen of crime fiction, Agatha Christie wrote numerous novels and short stories that are still widely read today. Some of her most famous works include Hercules Poirot series and Miss Marple series. Raymond Chandler novels: regarded as one of the greatest crime writers of all time, Raymond Chandler wrote novels set in Los Angeles featuring detective Philip Marlowe. Some of his most famous works include T̈he Big Sleepänd T̈he Long Goodbye.̈ 4. crime novels by crime novelist Raymond Chandler: regarded as one of the greatest crime writers of all time, Raymond Chandler wrote novels set in Los Angeles featuring detective Philip Marlowe. Some of his most famous	Based on your preference for crime and mystery books, here are some highly recommended titles: 1. To Kill a Mockingbird by Harper Lee: This classic novel explores themes of racism, prejudice, and injustice in the Deep South during the 1930s. 2. The Silent Patient by Alex Michaelides: A psychological thriller about a woman who shoots her husband and then refuses to speak. 3. The Girl with the Dragon Tattoo by Stieg Larsson: This novel kicks off the Millennium Trilogy, following the investigation of a missing journalist and the mysterious disappearance of a young woman. 4. The Da Vinci Code by Dan Brown: This fast-paced thriller follows the trail of a centuries-old conspiracy involving the Catholic Church, the Knights Templar, and the secrets of the Da Vinci Code. 5. The Girl with the Dragon Tattoo by Stieg Larsson: This novel kicks off the Millennium Trilogy, following the investigation of a missing journalist and the mysterious
40	Here are some of the best books in the Crime and Mystery genre: 1. The Hound of the Baskervilles by Sir Arthur Conan Doyle 2. The Maltese Falcon by Dashiell Hammett 3. The Big Sleep by Raymond Chandler 4. The Adventures of Sherlock Holmes by Sir Arthur Conan Doyle 5. The Thin Man by Dashiell Hammett 6. The Murder of Roger Ackroyd by Agatha Christie 7. The Silence of the Lambs by Thomas Harris 8. The Red House Mystery by A.A. Milne 9. The Secret of the Old Clock by Carolyn Keene 10. The Adventures of the Black Stallion by Walter Farley.	Here are some highly recommended crime and mystery books: 1. To Kill a Mockingbird by Harper Lee 2. The Hound of the Baskervilles by Sir Arthur Conan Doyle 3. The Silence of the Lambs by Thomas Harris 4. The Great Gatsby by F. Scott Fitzgerald 5. The Catcher in the Rye by J.D. Salinger 6. The Da Vinci Code by Dan Brown 7. The Girl with the Dragon Tattoo by Stieg Larsson 8. The Adventures of Sherlock Holmes by Sir Arthur Conan Doyle 9. The Secret of the Old Clock by Carolyn Keene 10. The Help by Kathryn Stockett.

Query: Classify the questions in the FAQ into Finding policy details, Change account settings, Filing a claim and viewing status, or Cancelling coverage.
Could you deposit money into my account rather than mailing me a physical cheque?

Layer	**LLaMA Response (SFT + ICT)**	**Sorted LLaMA Response (SoFT)**
12	Yes sir/ sorry sorryonymouslyovoyedance Festivalandeiamot...	The question is not classified as a request or request for information. It is a question that is addressed to the customer's request.
16	belongs belongsfrontieralandersonallyi...	The given question is a question of financial accounting and it belongs to the Cancelling coverage category.
20	belongs category c̆hangeable via email or phone call or online portal or phone contactless payment etceterially...	Change account settings
24	Change account settings	Change account settings
28	This question falls category of Change account settings.	Change account settings
32	Change account settings	Change account settings
36	Change account settings	Change account settings
40	Change account settings	Change account settings

Table D.4: Generated responses in different layers of SFT (Early Exit) and Sorted LLaMA for two samples from PandaLM validation set.

Query: Based on the given keywords, write a review of a restaurant. Name: Pizzeria
good prices, quiet place, tasty, nice staff

Layer	LLaMA Response (SFT + ICT)	Sorted LLaMA Response (SoFT)
12	Hinweis B0̆0e9jar Brothersqueeries...	The restaurant is located in a small town, with an impressive list of options for meals and drinks. The atmosphere is comfortable and comfortable, with a small, large-scale room, with a large enough room for comfortable comfortable rooms. The menu is a great way to choose the best option for meals and drinks. The menu is easy to understand and can be easily used to find the best option for meals and drinks. The menu is also available online and can be easily accessed by anyone who is interested in making a purchase. The menu is also made up of a variety of cuisine, with a variety of cuisine. The menu is also made up of a variety of cuisine...
16	Review: description description Cassandra Smithsoniusque ...	The Pizzeria is a great place to eat and enjoy a relaxing meal. It is a quiet place, so it is perfect for a quiet meal and a relaxing meal. The food is incredibly delicious, and the staff is friendly and friendly. The prices are great, and the atmosphere is comfortable and relaxing. It is a great place to eat and enjoy a relaxing meal.
20	Review: visited recently visited recently visited recently visited recently...	I recently visited Pizzeria, a restaurant that is both affordable and a quiet place to enjoy a meal. The prices are definitely good, and the food is definitely tasteful. The staff is nice and friendly, making the experience even more enjoyable. Overall, I highly recommend Pizzeria for anyone looking for a good meal at a reasonable price.
24	Review: visited recently visited recently visited recently visited recently ...	I recently visited Pizzeria, and I must say, it was a pleasant experience. The prices were reasonable, and the atmosphere was quiet and relaxing. The staff was friendly and helpful, making me feel welcome and comfortable. The pizza was tasty and made me feel satisfied after a long day. Overall, I would recommend Pizzeria to anyone looking for a good pizza, good prices, and a nice atmosphere
28	Revieweddit rating: stars star star star ...	Pizzeria is a hidden gem in the heart of the city. The prices are reasonable and the atmosphere is a refreshing change from the crowded restaurants. The staff is friendly and welcoming, making the experience even more enjoyable. The pizza is tasty and the crust is crisp and crunchy. The sauce is rich and the cheese is melted perfectly. Overall, Pizzeria is a must-visit for anyone looking for a good pizza and a quiet place to enjoy it.
32	Review: visited this place with friends recently after hearing good reviews about their pizza pie. Upon arrival we noticed how quiet it was inside which made it perfect place to relax after hectic weekdays. prices were reasonable considering quality of food served. staff members were friendly and helpful throughout entire duration of visit. overall experience exceed expectations thanks to delicious food served hot freshly baked pizza pie tasteful ingunctions of Italian cu cu cu cu cu cu cu cu cu cu ...	I recently visited Pizzeria and I must say, it was a pleasant experience. The prices were reasonable, and the atmosphere was quiet and serene. The staff was friendly and helpful, making me feel welcome. The pizza was delicious, with a crisp crust and a delicious, creamy sauce. Overall, I would definitely recommend Pizzeria to anyone looking for a good meal in a quiet, relaxing setting.
36	Review of Pizzeria: upon walking into pizzeria, you are greetvity welcomed by nice staff members who are always ready to assist you with your needs. Their prices are reasonable considering the quality of food they offer. Their pizza is mouthwateringly delicious and always freshly made to order. Additionally, pizzeria is a quiet place perfect for dates or family gatherings. Overall, pizzeria is a great place to visit for delicious food and great service. highly recommended!	I recently visited Pizzeria and had a wonderful experience. The prices were reasonable and the atmosphere was quiet and relaxing. The food was delicious and the staff was friendly and attentive. Overall, I would highly recommend Pizzeria to anyone looking for a tasty meal in a nice, quiet setting.
40	Pizzeria is a great place to go for a quiet meal with tasty food and friendly staff. The prices are good, making it a great value for your money. The atmosphere is cozy and the service is prompt. The pizza is cooked to perfection and the toppings are fresh and delicious. Overall, it's a great place to enjoy a meal with friends or family. Highly recommended!	Pizzeria is a great place to enjoy a delicious pizza in a quiet and cozy atmosphere. The prices are reasonable and the staff is friendly and accommodating. The pizzas are cooked to perfection and the toppings are fresh and tasty. The restaurant is perfect for a casual meal with friends or family, and the ambiance is relaxing and inviting. Overall, Pizzeria is a great option for a satisfying pizza experience.

Query: Define what the underlined word means for kids. _keep a promise

Layer	LLaMA Response (SFT + ICT)	Sorted LLaMA Response (SoFT)
12	Thereuh Scientificallyally compromiseable ...	The word machery is a word that is used to describe a person or a person's behavior, or a person's behavior in a particular manner. It means that someone has been able to successfully successfully deal with the situation or situation situationally.
16	means something importantlyrics song songwriter	The word Kick A promise is a common term in the context of a commitment to deliver something or to take action. It means that someone has taken a promise to make a promise to make sure that they are doing something or fulfilling their promise. It is a kind of commitment that is often used in personal and professional settings.
20	When someone promise means promise means promise means promise means promise means ...	Keeping a promise means keeping a promise made to someone else, and not making a promise to someone else or even yourself that you will not do something, but actually doing it. It means being kind, honesty, and kindness towards others. It means being kind to your parents, siblings, friends, and even your peers. It means being kind to yourself too. It means being kind to yourself ...
24	When someone promise promise means promise promise means promise ...	Keep a promise means to make sure that you do what you promised to do. It means that you are reliable and reliable people are people who are trusted and respected. When you promise something to someone, it means that you are committed to it and you will do it.
28	keep promise means promise promise means promise promise means promise promise ...	Keeping a promise means to do what you promised to do, no matter what. It means keeping your word and keeping your word is important. It's important to keep promises because it helps people trust you and trust that you will do what you say you will do.
32	When you promise something means prom promprompr...	Keeping a promise means that you promise something to someone and you do it. It means that you do not break your word and you do what you said you would do. It is important to keep promises because it shows that you are trustworthy and reliable.
36	Keeping a promise means promising something to someone else or yourself, and then doing your best to make sure that promise comes true. It means being honest and keeping your word even when it'vity hard or takes effort. It means being responsible and reliable so people can trust you to do what you say you will do.	Keeping a promise means doing what you said you would do. It's important to follow through on your promises to others and to be reliable.
40	Keep a promise means to do what you said you would do. It means to be honest and to keep your word. When you keep a promise, you show that you are trustworthy and that people can count on you.	Keeping a promise means doing what you said you would do, even if it's hard or takes a long time. It's important to be honest and reliable, and to follow through on your words.

Table D.5: Generated responses in different layers of SFT (Early Exit) and Sorted LLaMA for a sample from PandaLM validation set.

Query: Bandar Seri Begawan International airport is in which country?		
Layer	**LLaMA Response (SFT)**	**Sorted LLaMA Response (SoFT)**
12	iernohotter than than ...	Malta
16	Nederlige territ territorially ...	Burma (Myanmar)
20	Australia AustraliaAustral Australia Australia ...	Brunei
24	Malays Malays Malays Malays ...	Brunei
28	Malays Malays Malays Malays ...	Brunei
32	Brunei	Brunei
36	Brunei	Brunei
40	Brunei	Brunei

Table D.6: Generated responses in different layers of SFT and SoFT for a sample from TriviaQA benchmark.

Sorted/Fully Fine-tuned	12 (4.1B)	20 (6.6B)	28 (9.2B)	36 (11.7B)
2 SFT Epochs/2 SoFT Epochs				
12 (4.1B)	80.0/88.5/1.5	37.5/132.0/0.5	28.0/141.5/0.5	20.0/148.5/1.5
16 (5.4B)	88.5/77.0/4.5	42.0/121.5/6.5	31.5/135.0/3.5	20.0/142.5/7.5
20 (6.6B)	114.0/48.5/7.5	56.0/84.5/29.5	42.5/108.0/19.5	32.0/117.5/20.5
24 (7.9B)	123.0/37.0/10.0	70.5/61.5/38.0	53.5/80.0/36.5	45.5/89.5/35.0
28 (9.2B)	131.0/32.0/7.0	75.0/63.0/32.0	56.0/70.5/43.5	46.5/82.5/41.0
32 (10.4B)	143.5/21.0/5.5	98.0/43.5/28.5	73.0/54.0/43.0	54.0/65.5/50.5
36 (11.7B)	140.5/22.0/7.5	98.5/40.5/31.0	76.0/49.0/45.0	53.0/62.5/54.5
40 (13B)	137.5/24.0/8.5	102.0/37.0/31.0	78.5/45.5/46.0	55.0/62.0/53.0
2 SFT Epochs/4 SoFT Epochs				
12 (4.1B)	94.5/71.0/4.5	44.0/121.0/5.0	37.0/130.5/2.5	26.5/138.5/5.0
16 (5.4B)	105.0/60.0/5.0	55.0/102.0/13.0	51.0/110.5/8.5	34.0/123.0/13.0
20 (6.6B)	129.5/33.5/7.0	73.0/67.5/29.5	58.5/85.0/26.5	47.0/96.5/26.5
24 (7.9B)	132.0/30.5/7.5	89.5/51.0/29.5	70.0/62.5/37.5	51.0/80.0/39.0
28 (9.2B)	140.0/23.5/6.5	89.5/51.0/29.5	66.5/60.0/43.5	48.5/77.5/44.0
32 (10.4B)	144.5/18.5/7.0	103.5/35.0/31.5	77.5/52.0/40.5	55.5/62.0/52.5
36 (11.7B)	146.0/17.5/6.5	105.5/34.5/30.0	84.5/44.5/41.0	60.0/52.5/57.5
40 (13B)	149.0/15.0/6.0	105.0/37.5/27.5	87.5/41.5/41.0	62.5/53.5/54.0

Table D.7: Pair-wise comparison between Extracted fine-tuned and SoFT sub-models.