LeeBERT: Learned Early Exit for BERT with Cross-Level Optimization

Wei Zhu[1,2] *
[1] East China Normal University, Shanghai, China
[2] Pingan Health Tech, Shanghai, China

Abstract

Pre-trained language models like BERT are performant in a wide range of natural language tasks. However, they are resource exhaustive and computationally expensive for industrial scenarios. Thus, early exits are adopted at each layer of BERT to perform adaptive computation by predicting easier samples with the first few layers to speed up the inference. In this work, to improve efficiency without performance drop, we propose a novel training scheme called Learned Early Exiting for BERT (LeeBERT). First, we ask each exit to learn from each other, rather than learning only from the last layer. Second, the weights of different loss terms are learned, thus balancing off different objectives. We formulate the optimization of LeeBERT as a bi-level optimization problem, and we propose a novel cross-level optimization (CLO) algorithm to improve the optimization results. Experiments on the GLUE benchmark show that our proposed methods improve the performance of the state-of-the-art (SOTA) early exiting methods for pre-trained models.

1 Introduction

The last couple of years have witnessed the rise of pre-trained language models (PLMs), such as BERT (Devlin et al., 2018), GPT (Radford et al., 2019), XLNet (Yang et al., 2019), and ALBERT (Lan et al., 2020), etc. By pre-training on the unlabeled corpus and fine-tuning on labeled ones, BERT-like models achieved considerable improvements in many Natural Language Processing (NLP) tasks, such as text classification and natural language inference (NLI), sequence labeling, etc.

However, these PLMs suffer from two problems. The first problem is efficiency. The state-of-the-art (SOTAs) achievements of these models usually rely on very deep model architectures accompanied by high computational demands, impairs their practicalities. Like general search engines or online medical consultation services, industrial settings process generally millions of requests per minute. What makes efficiency more critical is that the traffic of online services varies drastically with time. For example, during the flu season, the search requests of Dingxiangyuan[1] are ten times more than usual. And the number of claims during the holidays is five to ten times more than that of the workdays for online shopping. Many servers need to be deployed to enable BERT in industrial settings, which is unbearable for many companies.

Second, previous literature (Fan et al., 2020; Michel et al., 2019; Zhou et al., 2020) pointed out that large PLMs with dozens of stacked Transformer layers are over-parameterized and could suffer from the "overthinking" problem (Kaya et al., 2019). That is, for many input samples, their shallow representations at a shallow layer are enough to make a correct classification. In contrast, the final layer's representations may be overfitting or distracted by irrelevant features that do not generalize. The overthinking problem leads to not only poor generalization but also wasted computation.

To address these issues, both the industry and academia have devoted themselves to accelerating PLMs at inference time. Standard methods include direct network pruning (Zhu and Gupta, 2018; Xu et al., 2020; Fan et al., 2020; Michel et al., 2019), knowledge distillation (Sun et al., 2019; Sanh et al., 2019; Jiao et al., 2020), weight quantization (Zhang et al., 2020; Bai et al., 2020; Kim et al., 2021) and adaptive inference (Zhou et al., 2020; Xin et al., 2020; Geng et al., 2021; Liu et al., 2020). Among them, adaptive inference has attracted much attention. Given that real-world data is usually com-

*Contact: 52205901018@stu.ecnu.edu.cn.

[1] https://search.dxy.cn/

posed of easy samples and difficult samples, adaptive inference aims to deal with simple examples with only a small part of a PLM, thus speeding up inference time on average. The speed-up ratio can be controlled with certain hyper-parameters to cope with drastic changes in request traffic. What's more, it can address the over-thinking problem and improve the model's generalization ability.

Early exiting is one of the most crucial adaptive inference methods (Bolukbasi et al., 2017). It implements adaptive inference by installing exits, or intermediate prediction layer, at each layer of BERT and exiting "easy" samples at exits of the shallow layers to speed up inference (Figure 1). Strategies for early exiting are designed (Teerapittayanon et al., 2016; Kaya et al., 2019; Xin et al., 2020; Zhou et al., 2020), which decides when to exit given the current obtained predictions (from previous and current layers).

Early exiting architectures' training procedure is essentially a multi-objective problem since each exit is trying to improve its performance. Different objectives from different classifiers may conflict and interfere with one-another (Phuong and Lampert, 2019; Yu et al., 2020). Thus they incorporate distillation loss to improve the training procedure by encouraging early exits to mimic the output distributions of the last exit. The motivation is that the last exit has the maximum network capacity and should be more accurate than the earlier exits. In their work, only the last exit can act as a teacher exit. Besides, the multiple objectives are uniformly weighted.

In this work, we propose a novel training mechanism called Learned Early Exiting for BERT (LeeBERT). Our contributions are three folded. First, instead of learning from the last exit, LeeBERT asks each exit to learn from each other. The motivation is that different layers extract features of varying granularity. Thus they have different perspectives of the sentence. Distilling knowledge from each other improves the expressiveness of lower exits and alleviates the overfittng of the later exits. Second, to achieve the optimal trade-offs between different loss terms, their weights are treated as parameters and are learned along with model parameters. The optimization of the learnable weights and model parameters is formulated as a bi-level optimization problem, optimized with gradient descent. Built upon previous literature (Liu et al., 2019), we propose a novel cross-level optimization (CLO) algorithm to solve the bilevel optimization better.

Extensive experiments are conducted on the GLUE benchmark (Wang et al., 2018), and show that LeeBERT outperforms existing SOTA BERT early exiting methods, sometimes by a large margin. Ablation study shows that: (1) knowledge distillation among all the exits can improve their performances, especially for the shallow ones; (2) our novel CLO algorithm is useful in learning more suitable weights and brings performance gains.

Our contributions are integrated into our LeeBERT framework, which can be summarized as follows:

- We propose a novel training method for early exiting PLMs to ask each exit to learn from each other.
- We propose to find the optimal trade-off of different loss terms by assigning learnable weights.
- We propose a novel cross-level optimization (CLO) algorithm to learn the loss term weights better.

2 Preliminaries

In this section, we introduce the necessary background for BERT early exiting. Throughout this work, we consider the case of multi-class classification with samples $\{(x_n, y_n), x_n \in \mathcal{X}, y_n \in \mathcal{Y}, i = 1, 2, ..., N\}$, e.g., sentences, and the number of classes is K.

2.1 Backbone models

In this work, we adopt BERT and ALBERT as backbone models. BERT is a multi-layer Transformer (Vaswani et al., 2017) network, which is pre-trained in a self-supervised manner on a large corpus. ALBERT is more lightweight than BERT since it shares parameters across different layers, and the embedding matrix is factorized.

2.2 Early exiting architecture

As depicted in Figure 1, early exiting architectures are networks with exits at different transformer layers. With M exits, M classifiers $\mathbf{p}_m : \mathcal{X} \rightarrow \Delta^K$ ($m = 1, 2, ..., M$) are designated at M layers of BERT, each of which maps its input to the probability simplex Δ^K, i.e., the set of probability distributions over the K classes. Previous literature (Phuong and Lampert, 2019; Liu et al., 2020) think

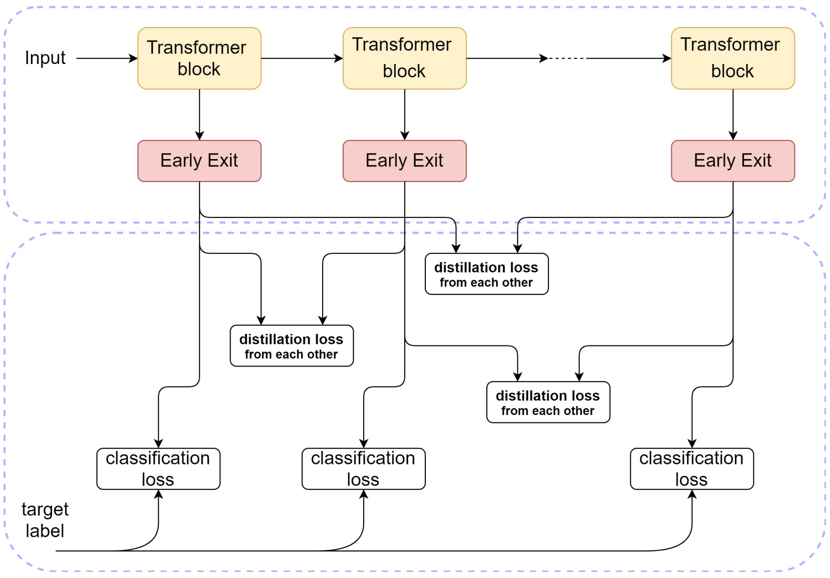


Figure 1: The training procedure of LeeBERT, which differs from the previous literature in two aspects. First, we let exits learn from each other, instead of only asking shallow exits to learn from the deepest exit. Second, the importance of each distillation loss term are retained along with the learning of model parameters.

of $\mathbf{p}_1, ..., \mathbf{p}_M$ as being ordered from least to most expressive. However, in terms of generalization ability, due to the over-thinking problem, later layers may not be superior to shallow layers.

In principle, the classifiers may or may not share weights and computation, but in the most interesting and practically useful case, they share both.

2.3 Early exiting strategies

There are mainly three early exiting strategies for BERT early exiting. BranchyNet (Teerapittayanon et al., 2016), FastBERT (Liu et al., 2020) and DeeBERT (Xin et al., 2020) calculated the entropy of the prediction probability distribution as a proxy for the confidence of exiting classifiers to enable early exiting. Shallow-Deep Nets (Kaya et al., 2019) and RightTool (Schwartz et al., 2020) leveraged the softmax scores of predictions of exiting classifiers, that is, if the score of a particular class is dominant and large enough, the model will exit. Recently, PABEE (Zhou et al., 2020) propose a patience based exiting strategy analogous to early stopping model training, that is, if the exits' predictions remain unchanged for a pre-defined number of times (patience), the model will stop inference and exit. PABEE achieves SOTAs results for BERT early exiting.

In this work, we mainly adopt the PABEE's patience based early exiting strategy. However, in ablation studies, we will show that our LeeBERT framework can improve the inference performance of other exiting strategies.

3 Our LeeBERT framework

In this section, we introduce the proposed LeeBERT framework. First, we present our distillation based loss design, and then we elaborate on how to optimize with learnable weights. Our main contribution is a novel training mechanism for BERT early exiting, which extends Liu et al. (2020) and Phuong and Lampert (2019) via mutual distillation and learned weights.

3.1 Loss objectives

3.1.1 Classification loss

When receiving an input sample (x_n, y_n), each exit will calculate the cross-entropy loss based on its predicted, and all the exits are simultaneously optimized with a summed loss, i.e.,

$$\mathcal{L}_{CE}(x_n, y_n) = \sum_{m=1}^{M} \mathcal{L}_{CE}(\mathbf{p}_m(x_n), y_n). \quad (1)$$

Note that the above objective directly assumes uniform weights for all M loss terms.

3.1.2 Distillation loss

To introduce our contribution, we first remind the reader of the classical distillation framework as introduced in Hinton et al. (2015): assume we want a probabilistic classifier s (student) to learn from another classifier t (teacher). This can be achieved by minimizing the (temperature-scaled) cross-entropy between their prediction distributions,

$$\mathcal{L}_{KD}(t,s) = -\tau^2 \sum_{k=1}^{K} [t^{1/\tau}(x_n)]_k \log[[s^{1/\tau}(x_n)]_k], \tag{2}$$

where $\tau \in \mathbf{R}_+$ is the distillation temperature, and

$$[t^{1/\tau}(x)]_k = \frac{t_k(x)^{1/\tau}}{\sum_{k'=1}^{K} t_{k'}(x)^{1/\tau}}, \tag{3}$$

is the distribution obtained from the distribution $t(x)$ by temperature-scaling, and $[t^{1/\tau}(x)]_k$ is defined analogously.

The temperature parameter allows controlling the softness of the teachers' predictions: the higher the temperature, the more suppressed is the difference between the largest and the smallest value of the probability vector. The temperature scaling allows compensating for the over-confidence of the network's outputs, i.e., they put too much probability mass on the top predicted class and too little on the others. The factor τ^2 in Eq 2 ensures that the temperature scaling does not negatively affect the gradient magnitude.

Returning to the early exiting architecture, we follow the same strategy as classical distillation but use exits of different layers both as students and teachers. For any exit m, let $\mathcal{T}(m) \subset 1,...,M$ (which could be empty) be the set of teacher exits it is meant to learn from. Then we define the overall distillation loss as

$$\mathcal{L}_{KD}(x_n) = \sum_{m=1}^{M} \sum_{t \in \mathcal{T}(m)} \frac{\mathcal{L}_{KD}(\mathbf{p}_t(x_n), \mathbf{p}_m(x_n))}{M * |\mathcal{T}(m)|}. \tag{4}$$

Previous work (Phuong and Lampert, 2019; Liu et al., 2020) considers using only the last exit as as the teacher and all exits learn from it. The usual belief is that deeper exits have more network capacity and more accurate than the early exits. However, the over-thinking phenomenon reveals that later exits may not be superior to earlier ones. The more shallow exit may provide different perspectives in semantic understanding of the input sentences. Thus, to fully learn from available information, later exits can benefit from learning from early exits. With this motivation, we consider two settings:

Learn from Later Exits (LLE). In this setting, early exits learn from all its later exits.

Learn from All Exits (LAE). In this setting, an exit learns from all other exits.

3.2 Weighted loss

Previous work considers uniform weights for the distillation loss terms or classification loss term, which does not effectively take the trade-off among multiple objectives. First, from the perspective of knowledge distillation, intuitively, later exits should place little weights on the very early exits since they have less to offer. And all exits should place higher importance on exits that are performant and not overfitting. Second, different loss objectives are usually competing, which may hurt the final results.

To address these issues, we propose to assign a set of learnable weights to our loss objective, which are updated via gradient descent along with the model parameters. We give weight w_i for each classification loss term and $w_{m,t}$ for the distillation loss term coming from exit m learning from exit t, and the overall loss objective becomes

$$\mathcal{L}(x_n, y_n) = \sum_{m=1}^{M} w_i \mathcal{L}_{CE}(\mathbf{p}_m(x_n), y_n) + \sum_{m=1}^{M} \sum_{t \in \mathcal{T}(m)} w_{m,t} \frac{\mathcal{L}_{KD}(\mathbf{p}_t(x_n), \mathbf{p}_m(x_n))}{M * |\mathcal{T}(m)|}. \tag{5}$$

Note that $\Omega = \{w_i, w_{m,t}\}$ can be understood as a set of learnable training hyper-parameter.

3.3 Optimization of Learned weights

3.3.1 Single vs. Bi-level optimization

Assume we have two datasets D_1 and D_2, which usually are both subsets of the training set D_{tr}. D_1 can be equal to D_2. For a given set of $\Omega = \{w_i, w_{m,t}\}$, the optimal solution $\Theta^*(\Omega)$ of network parameters Θ are derived from D_1, and the optimal Ω^* are determined on D_2. We denote the loss on dataset D as $\mathcal{L}_D(\Theta, \Omega)$, a function of two sets of parameters for convenience. Then the optimization problem becomes

$$\begin{aligned} &min_{\Omega} \mathcal{L}_{D_2}(\Theta^*(\Omega), \Omega), \\ &s.t., \Theta^*(\Omega) = \arg\min_{\Theta} \mathcal{L}_{D_1}(\Theta, \Omega) \end{aligned} \tag{6}$$

Though the above bi-level optimization can accurately describe our problem, it is generally difficult to solve. One heuristic simplification of the above equation is to let $D_1 = D_2 = D_{tr}$, and the optimization problem in Eq 16 reduces to the single-level optimization (SLO),

$$min_{\Theta,\Omega}\mathcal{L}_{D_{tr}}(\Theta,\Omega), \tag{7}$$

which can be solved directly by stochastic gradient descent. This reduced formulation treats the learnable weights Ω just as a part of the model parameters. Despite its efficiency, compared with Θ, the number of parameters in Ω is almost neglectable. Thus optimization will need to fit Θ well for gradient descent, resulting in inadequate solutions of Ω.

The most widely adopted optimization algorithm for Eq 16 is the bi-level optimization (BLO) algorithm Liu et al. (2019), which asks D_1 and D_2 to be a random split of D_{tr}.[2] And the gradient descent is done following:

$$\begin{aligned}\Theta &= \Theta - \lambda_1\nabla_\Theta L_{D_1},\\ \Omega &= \Omega - \lambda_2\nabla_\Omega L_{D_2}.\end{aligned} \tag{8}$$

that is, updating the parameters in an interleaving fashion: one-step gradient descent of Θ on D_1 followed by one step gradient descent of Ω on D_2. Note that $\Theta^*(\omega)$ in Eq 16 is not satisfied in BLO due to first-order approximation, leading gradient updates of ω into wrong directions, collapsing the bi-level optimization.

3.4 Cross-level optimization

We now propose our cross-level optimization algorithm. The gradient descent updating of Θ and Ω follows

$$\begin{aligned}\Theta &= \Theta - \lambda_1\nabla_\Theta L_{D_1},\\ \Omega &= \Omega - \lambda_1\nabla_\Omega L_{D_1} - \lambda_2\nabla_\Omega L_{D_2}.\end{aligned} \tag{9}$$

The above equation is the core of our CLO algorithm, which we will refer to as CLO-v1, which are derived and demonstrated in detail in the Appendix. We can see that our cross-level optimization's core idea is to draw gradient information from both splits of the training set, thus making the updating of Ω more reliable.

Note that updating Ω requires its gradients on both the D_1 set and D_2 set. Thus its computation complexity is higher than the BLO algorithm. We propose a more efficient version of cross-level optimization (CLO-v2), which can also be found in the Appendix. We divide the training procedure into groups, each group containing C steps, Θ is updated solely on the training set for $C-1$ steps, and updated following Eq 9 for the remaining one step. We will call the hyper-parameter C as the cross-level cycle length. CLO-v2 is more efficient than CLO-v1, and our experiments show that CLO-v2 works well and is comparable with CLO-v1.

[2]Note that on each epoch start, the split of D_{tr} can be re-generated.

4 Experiments

4.1 Tasks and Datasets

We evaluate our proposed approach to the classification tasks on GLUE benchmark. We only exclude the STS-B task since it is a regression task, and we exclude the WNLI task following previous work (Devlin et al., 2018; Jiao et al., 2020; Xu et al., 2020).

4.2 Backbone models

Backbone models. All of the experiments are built upon the Google BERT, ALBERT. We ensure fair comparison by setting the hyper-parameters related to the PLM backbones the same with HuggingFace Transformers (Wolf et al., 2020).

4.3 Baseline methods

We compare with the previous BERT early exiting methods and compare other methods that speed up BERT inference.

Directly reducing layers. We experiment with directly utilizing the first 6 and 9 layers of the original (AL)BERT with a single output layer on the top, denoted by (AL)BERT-6L and (AL)BERT-9L, respectively. These two baselines serve as a lower bound for performance metrics since it does not employ any technique.

Static model compression approaches. For model parameter pruning, we include the results of LayerDrop (Fan et al., 2020) and attention head pruning (Michel et al., 2019) on ALBERT. For knowledge distillation, we include DistillBERT (Sanh et al., 2019), BERT-PKD (Sun et al., 2019).[3] For module replacing, we include BERT-of-Theseus (Xu et al., 2020).

Input-adaptive inference. This category includes entropy-based method DeeBERT, score-based method Shallow-deep, and patience-based exiting method PABEE as our baselines. We also

[3]Note that the two methods consider knowledge distillation on the fine-tuning stage, whereas TinyBERT (Jiao et al., 2020) and Turc et al. (2019) investigate knowledge distillation during both the pre-training stage and fine-tuning stage.

Method	#Param	Speed-up	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2
			Dev set						
ALBERT-base	12M	1.00x	57.4	84.6	89.5	89.2	89.6	75.6	91.8
ALBERT-6L	12M	1.96x	51.9	80.2	85.8	84.7	86.8	70.6	88.8
ALBERT-9L	12M	1.30x	53.8	81.2	87.1	86.2	88.3	72.9	90.3
LayerDrop	12M	1.96x	52.2	79.8	85.9	84.5	87.3	71.3	89.7
HeadPrune	12M	1.22x	52.6	80.3	86.2	84.3	88.0	72.1	89.5
DeeBERT	12M	1.88x	53.7	81.7	87.2	86.4	87.4	72.4	89.6
Shallow-Deep	12M	1.95x	54.1	81.5	87.1	86.7	87.8	72.2	89.7
PABEE	12M	1.91x	56.4	83.9	88.7	88.6	88.9	74.4	90.5
FastBERT	12M	1.94x	57.1	84.7	89.1	89.0	89.3	75.6	90.9
FastBERT-CLO-v2	12M	1.95x	57.2	85.0	89.2	89.3	89.5	76.3	91.1
LeeBERT-LLE	12M	1.96x	57.5	85.1	89.5	89.4	89.8	76.7	91.3
LeeBERT-rand	12M	1.95x	57.0	84.8	89.2	89.1	89.2	75.8	91.0
LeeBERT-uniform	12M	1.95x	57.1	84.9	89.1	89.0	89.3	75.9	91.0
LeeBERT-SLO	12M	1.94x	57.2	85.0	89.2	89.3	89.6	76.0	90.9
LeeBERT-BLO	12M	1.93x	57.4	85.1	89.5	89.4	89.8	76.4	91.3
LeeBERT-CLO-v1	12M	1.95x	**57.9**	85.4	**89.9**	**89.7**	**90.3**	**76.9**	91.8
LeeBERT	12M	1.96x	57.8	**85.4**	89.7	89.7	90.2	76.8	**91.8**
			Test set						
ALBERT-base	12M	1.00x	54.1	84.3	87.0	88.3	71.1	73.4	92.8
PABEE	12M	1.89x	53.5	83.6	86.5	88.1	69.8	72.8	92.0
FastBERT	12M	1.95x	54.0	84.4	86.7	88.3	70.5	73.7	92.5
LeeBERT	12M	1.96x	**54.6***	**84.8***	**87.2**	**88.6**	**71.4***	**74.6***	**93.1***

Table 1: Experimental results of models with ALBERT backbone on the development set and GLUE test set. If not specified, LeeBERT and its variants (e.g., LeeBERT-LLE) are optimized using CLO-v2. The mean performance scores of 5 runs are reported. The speed-up ratio is averaged across 7 tasks. Best performances are bolded, "*" indicates the performance gains are statistically significant.

include the results of FastBERT when it adopts the PABEE's exiting strategy.

4.4 Experimental settings

We implement LeeBERT on the base of HuggingFace's Transformers. We conduct our experiments on a single Nvidia V100 16GB GPU.

Training. We add a linear output layer after each intermediate layer of the pre-trained BERT/ALBERT model as the internal classifier. **The hyperparameter tuning is done in a cross-validation fashion on the training set so that the dev set information of GLUE tasks are not revealed.** We perform grid search over batch sizes of 16, 32, 128, and learning rates of {1e-5, 2e-5, 3e-5, 5e-5} for model parameters Θ, and learning rates of {1e-5, 1e-4, 1e-3, 5e-3} for learnable weights Ω. The cross-level cycle length C will be selected from 2, 4, 8. We will adopt the Adam optimizer. At each epoch, the training set is randomly split into D_1 and D_2 with a ratio $5:5$. We apply an early stopping mechanism with patience 5 and evaluate the model on dev set at each epoch end. And we define the dev performance of our early exiting architecture as the average performance of all the exits. We will select the model with the best average performance in cross validation.

We set CLO-v2 as the main optimization algorithm of LeeBERT, and LAE as the main distillation strategy.[4] To demonstrate LeeBERT's ditillation objectives are beneficial, we train LeeBERT with the LLE strategy (LeeBERT-LLE). We also let the loss term weights in FastBERT to be learnable and train with our CLO-v2 algorithm, i.e., FastBERT-CLO-v2.

To compare our LeeBERT's CLO optimization procedure with baselines, we also train LeeBERT with (1) single level algorithm (LeeBERT-SLO); (2) bi-level algorithm (LeeBERT-BLO). To compare CLO-v1 and CLO-v2, we also train the LeeBERT with CLO-v1, i.e., LeeBERT-CLO-v1. Besides, we also include LeeBERT with randomly assigned discrete weights (LeeBERT-rand) and uniform weights (LeeBERT-uniform) as baselines, which will serve to demonstrate that our optimization procedure is beneficial. The discrete weights

[4]Henceforth, unless otherwise specified, our LeeBERT method will be the one with LAE and CLO-v2.

are randomly selected from $\{1, 2, ..., 50\}$, and are normalized so that the loss terms at each exit have weights summed to 1.

Inference. Following prior work, inference with early exiting is on a per-instance basis, i.e., the batch size for inference is set to 1. We believe this setting mimics the common latency-sensitive production scenario when processing individual requests from different users. We report the mean performance over 5 runs with different random seeds. For DeeBERT and Shallow-deep, we set the threshold for entropy or score, such that the speed-up ratio is between 1.80x to 2.1x. For FastBERT and our LeeBERT, we mainly adopt the PABEE's patience based exiting strategy, and we compare the results when the patience is set at 4. How the patience parameter affects the inference efficiency is also investigated for PABEE, FastBERT, and LeeBERT.

4.5 Overall Comparison

Table 1 reports the main results on GLUE with ALBERT as the backbone model. ALBERT is parameter and memory-efficient due to its cross-layer parameter sharing strategy, however, it still has high inference latency. From Table 1 we can see that our approach outperforms all compared methods to improve inference efficiency while maintaining good performances, demonstrating the proposed LeeBERT framework's effectiveness. Note that our system can effectively enhance the original ALBERT and PABEE by a relatively large margin when speeding-up inference by 1.97x. We also conduct experiments on the BERT backbone with the MNLI, MRPC, and SST-2 tasks, which can be found in the Appendix. To give more insight into how early exits perform under different efficiency settings, we illustrate how the patience parameter affect the average number of inference layers (which is directly related to speed-up ratios) (Figure 2), and prediction performances (Figure 3). We also show that one can easily apply our LeeBERT framework to image classification tasks in the Appendix.

4.6 Analysis

We now analyze more deeply the main take-aways from Table 1 and our experiments.

Our LeeBERT can speed up inference. Figure 2 shows that on the MRPC task, with the same patience parameter, LeeBERT usually goes through fewer layers (on average) than PABEE and FastBERT, showing the LeeBERT can improve the efficiency of PLMs' early exiting.

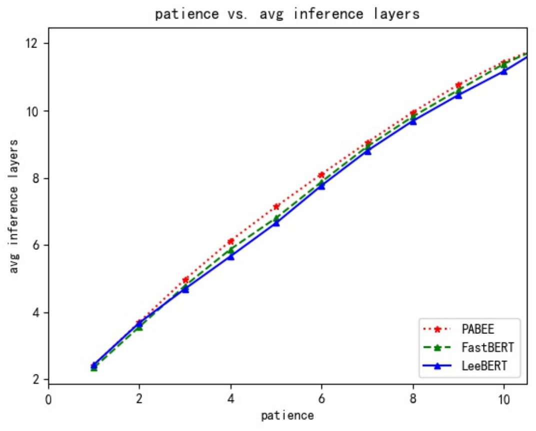
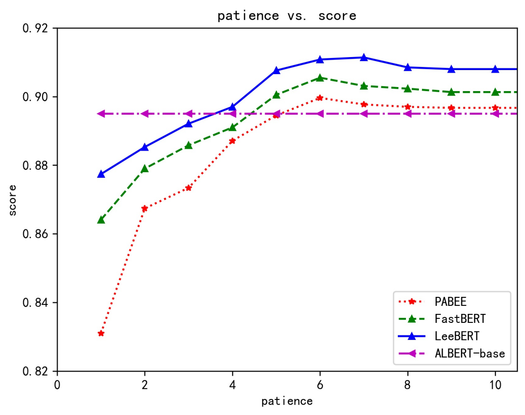


Figure 2: The curve of patience vs. avg inference layers for PABEE, FastBERT and LeeBERT. The task is MRPC.

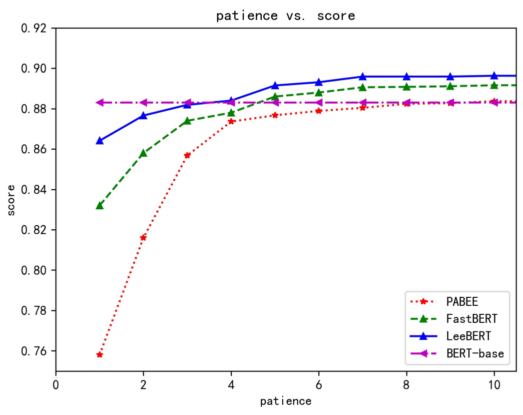
Our knowledge distillation strategies are beneficial. Table 1 reveals that our LAE setting provides the best overall performances on GLUE in terms of distillation strategies. LeeBERT outperforms FastBERT-CLO-v2 on all tasks and exceeds LeeBERT-LLE on 6 of the seven tasks, and the scores on QNLI the results are comparable. This result proves that exits learning from each other are generally beneficial.

Our CLO algorithm brings performance gains. As a sanity check, LeeBERT-rand performs worse than all optimized LeeBERT models. Table 1 also shows that the SLO and BLO algorithms perform worse than our CLO. And we can see that CLO-v1 and CLO-v2 have comparable results. CLO-v1 seems to have slight advantages on tasks with few samples, but the performance gaps seem to be marginal. Since CLO-v2 is more efficient, we will use CLO-v2 as our main optimization algorithm.

The patience-score curves are different for different PLMs. Figures 3(a) and 3(b) show that diffent PLMs have quite different patience-score curves. For ALBERT, early exiting with PABEE's strategy can improve upon the ALBERT-base fine-tuning, and the best performance is obtained with patience 6. With patience 6, the average number of inference layers is 8.11. This phenomenon shows that ALBERT base may suffer from the overthinking problem. With the help of our distillation strategy and CLO optimization, the performance gain is considerable. Note that: (a) Without distilla-



(a) ALBERT backbone



(b) BERT backbone

Figure 3: patience-performance curves for ALBERT and BERT on the MRPC task.

tion, shallow exits' performances are significantly worse, and our distillation can help these exits to improve; (b) with LeeBERT, the performances of the later exits are comparable to the earlier ones, since the over-thinking problem is alleviated by distillation. However, the patience-score curve for BERT is quite monotonic, suggesting that over-thinking problem is less severe. Note that BERT's shallow exits are significantly worse than that of ALBERT, and with LeeBERT, the shallow exits' performances are improved.

Training time costs. Table 2 presents the parameter numbers and time costs of training for LeeBERT compared with the original (AL)BERT, and PABEE, FastBERT. We can see that although exits need extra time for training, early exiting architectures actually can reduce the training time. Intuitively, additional loss objectives can be regarded as additional parameter updating steps for lower layers, thus speeding up the model convergence. LeeBERT-CLO-v1 requires a longer time for training. Notably, our LeeBERT's time costs are comparable with PABEE and FastBERT, even though it has more complicated gradient updating steps.

Working with different exiting strategies. Recall that our results are mainly obtained by adopting the PABEE's patience based exiting strategies. However, our LeeBERT framework is quite off-the-shelf, and can be integrated with many other exiting strategies. Our framework can work under different exiting strategies.[5] When using entropy-based strategy, LeeBERT outperforms DeeBERT by a large margin. When using Shallow-Deep's max probability strategy, LeeBERT outperforms Shallow-Deep on all GLUE tasks.

Method	#Params		Training time	
-	MRPC	SST-2	MRPC	SST-2
w/o early exiting	12M	12M	6.4	113
w PABEE	+18k	+18k	6.2	109
w FastBERT	+18k	+18k	6.0	102
w LeeBERT-CLO-v1	+18k	+18k	13.2	226
w LeeBERT(-CLO-v2)	+18k	+18k	6.5	118

Table 2: Comparison of Parameter numbers and training time costs. The Training time is the time cost (in minutes) until until the best performing checkpoint (on the dev set) with and without early exiting strategies on ALBERT as the backbone model.

5 Conclusion and discussions

In this work, we propose a new framework for improving PLMs' early exiting. Our main contributions lie in two aspects. First, we argue that exits should learn and distill knowledge from each other during training. Second, we propose that early exiting networks' training objectives be weighted differently, where the weights are learnable. The learnable weights are optimized with the cross-level optimization we propose. Experiments on the GLUE benchmark datasets show that our framework can improve PLMs' early exiting performances, especially under high latency requirements. Our framework is easy to implement and can be adapted to various early exiting strategies. We want to explore novel exiting strategies that better guarantee exiting performances in the future.

[5]Due to length limitation, we will leave the detailed results of this ablation study in the Appendix.

References

Haoli Bai, Wei Zhang, L. Hou, L. Shang, Jing Jin, X. Jiang, Qun Liu, Michael R. Lyu, and Irwin King. 2020. Binarybert: Pushing the limit of bert quantization. *ArXiv*, abs/2012.15701.

Tolga Bolukbasi, J. Wang, O. Dekel, and Venkatesh Saligrama. 2017. Adaptive neural networks for efficient inference. In *ICML*.

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

Angela Fan, E. Grave, and Armand Joulin. 2020. Reducing transformer depth on demand with structured dropout. *ArXiv*, abs/1909.11556.

Shijie Geng, Peng Gao, Z. Fu, and Yongfeng Zhang. 2021. Romebert: Robust training of multi-exit bert.

Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Geoffrey E. Hinton, Oriol Vinyals, and J. Dean. 2015. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531.

Xiaoqi Jiao, Y. Yin, L. Shang, Xin Jiang, X. Chen, Linlin Li, F. Wang, and Qun Liu. 2020. Tinybert: Distilling bert for natural language understanding. *ArXiv*, abs/1909.10351.

Y. Kaya, Sanghyun Hong, and T. Dumitras. 2019. Shallow-deep networks: Understanding and mitigating network overthinking. In *ICML*.

Se-Hoon Kim, Amir Gholami, Zhewei Yao, M. W. Mahoney, and K. Keutzer. 2021. I-bert: Integer-only bert quantization. *ArXiv*, abs/2101.01321.

A. Krizhevsky. 2009. Learning multiple layers of features from tiny images.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations. *ArXiv*, abs/1909.11942.

Hanxiao Liu, K. Simonyan, and Yiming Yang. 2019. Darts: Differentiable architecture search. *ArXiv*, abs/1806.09055.

Weijie Liu, P. Zhou, Zhe Zhao, Zhiruo Wang, Haotang Deng, and Q. Ju. 2020. Fastbert: a self-distilling bert with adaptive inference time. *ArXiv*, abs/2004.02178.

Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one? In *NeurIPS*.

Mary Phuong and Christoph H. Lampert. 2019. Distillation-based training for multi-exit architectures. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1355–1364.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108.

Roy Schwartz, Gabi Stanovsky, Swabha Swayamdipta, Jesse Dodge, and N. A. Smith. 2020. The right tool for the job: Matching model and instance complexities. In *ACL*.

S. Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. Patient knowledge distillation for bert model compression. In *EMNLP/IJCNLP*.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-read students learn better: On the importance of pre-training compact models. *arXiv: Computation and Language*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *ArXiv*, abs/1706.03762.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *BlackboxNLP@EMNLP*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

J. Xin, Raphael Tang, J. Lee, Y. Yu, and Jimmy Lin. 2020. Deebert: Dynamic early exiting for accelerating bert inference. *ArXiv*, abs/2004.12993.

Canwen Xu, Wangchunshu Zhou, Tao Ge, Furu Wei, and M. Zhou. 2020. Bert-of-theseus: Compressing bert by progressive module replacing. In *EMNLP*.

Z. Yang, Zihang Dai, Yiming Yang, J. Carbonell, R. Salakhutdinov, and Quoc V. Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*.

Tianhe Yu, Saurabh Kumar, A. Gupta, S. Levine, Karol Hausman, and Chelsea Finn. 2020. Gradient surgery for multi-task learning. *ArXiv*, abs/2001.06782.

W. Zhang, L. Hou, Y. Yin, L. Shang, X. Chen, X. Jiang, and Qun Liu. 2020. Ternarybert: Distillation-aware ultra-low bit bert. *ArXiv*, abs/2009.12812.

Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. 2020. Bert loses patience: Fast and robust inference with early exit. *ArXiv*, abs/2006.04152.

M. Zhu and S. Gupta. 2018. To prune, or not to prune: exploring the efficacy of pruning for model compression. *ArXiv*, abs/1710.01878.

A Derivation of our cross-level optimization algorithm.

We now derive our cross-level optimization (CLO) methods. Our objective is

$$\begin{aligned} &min_{\Omega}\mathcal{L}_{D_2}(\Theta^*(\Omega),\Omega), \\ &s.t.,\Theta^*(\Omega) = \arg\min_{\Theta}\mathcal{L}_{D_1}(\Theta,\Omega) \end{aligned} \tag{15}$$

Assume the optimal solution is Θ^* and Ω^*. The objective in Eq 16 can be viewed as minimizing the gap between $L_{D_1}(\Theta,\Omega)$ and $L_{D_1}(\Theta^*,\Omega^*)$, and minimizing the gap between $L_{D_2}(\Theta^*,\Omega)$ and $L_{D_2}(\Theta^*,\Omega^*)$. Thus, introducing slack variables δ_1 and δ_2, Eq 16 can be reformulated as

$$\begin{aligned} &min_{\Theta,\Omega}\delta_1^2+\delta_2^2, \\ &\quad s.t.,L_{D_1}(\Theta,\Omega) <= L_{D_1}(\Theta^*,\Omega^*)+\delta_1, \\ &\qquad L_{D_2}(\Theta^*,\Omega) <= L_{D_2}(\Theta^*,\Omega^*)+\delta_2, \\ &\qquad \delta_1 >= 0, \delta_2 >= 0. \end{aligned} \tag{16}$$

Using the Lagrangian multiplier method, the Lagrangian function is

$$\begin{aligned} &Lg(\delta_1,\delta_2,\Theta,\Omega,\Lambda) = \delta_1^2+\delta_2^2 \\ &- \lambda_1(L_{D_1}(\Theta,\Omega) - L_{D_1}(\Theta^*,\Omega^*) - \delta_1) \\ &- \lambda_2(L_{D_2}(\Theta^*,\Omega) - L_{D_2}(\Theta^*,\Omega^*) - \delta_2) \\ &- \lambda_3\delta_1 - \lambda_4\delta_2. \end{aligned} \tag{17}$$

To solve this Lagrangian function, the gradient descent updating of Θ and Ω becomes

$$\begin{aligned} &\Theta = \Theta - \lambda_1\nabla_{\Theta}L_{D_1}, \\ &\Omega = \Omega - \lambda_1\nabla_{\Omega}L_{D_1} - \lambda_2\nabla_{\Omega}L_{D_2}. \end{aligned} \tag{18}$$

Now we formally illustrate the CLO-v1 algorithm, which is in Algorithm 1. We also officially give the CLO-v2 algorithm in Algorithm 2.

B Hyper-parameters for each tasks

Table 3 reports the important hyper-parameters of LeeBERT for each task. Note that our hyper-parameter search was done on the training set with cross-validation so that the GLUE benchmarks' dev set information was not revealed during training.

C Results with BERT backbone

We conduct experiments with the BERT backbone on three representative tasks of GLUE, MNLI, MRPC, and SST-2. The results are reported in Table 5. The results show that our LeeBERT framework works well with different types of PLMs.

D Patience-performance curves on sst-2

We also provide the patience-performance curves (Figure 4) on the SST-2 task, with ALBERT and BERT backbones.

E Working with different exiting strategies

Our results are mainly obtained by adopting the PABEE's patience based exiting strategies. Now we demonstrate that LeeBERT can work with other exiting strategies. Table 4 shows that LeeBERT can help improve DeeBERT with its entropy-based exiting method and outperforms Shallow-deep with its max-prediction-based approach.

F LeeBERT are effective for image classification

To demonstrate the effectiveness of LeeBERT on the image classification task, we follow the experimental settings in Shallow-Deep (Kaya et al., 2019). We conduct experiments on two image classification datasets, CIFAR-10 and CIFAR-100 (Krizhevsky, 2009). And ResNet-56 (He et al., 2016) serves as the backbone and we compare LeeBERT with PABEE, DBT from Phuong and Lampert (2019). After every two convolutional layers, an exiting classifier is added. We set the batch size to 128 and use SGD optimizer with learning rate of 0.1. We set the cross level sycle to be 4, and learning rate of the learnable weights Ω are 0.01.

Table 6 reports the results. LeeBERT outperforms the full ResNet-56 on both tasks even when it provides 1.3x speed-up. Besides, it outperforms PABEE and DBT.

Algorithm 1: LeeBERT-CLO-v1

```
Parameters: $\Theta, \Omega$;
Return: the converged early exiting model; **while** *not converge* **do**
    **for** *t=1, ..., T* **do**
        sample batch $B_1$ and $B_2$ from $D_1$ and $D_2$, respectively
        update $\Theta$ with
            $\Theta = \Theta - \lambda_1 \nabla_{\Theta} L_{B_1},$    (10)
        calculate $L_{B_1}$ and $L_{B_2}$ with the updated $\Theta$, and update $\Omega$ with:
            $\Omega = \Omega - \lambda_1 \nabla_{\Omega} L_{B_1} - \lambda_2 \nabla_{\Omega} L_{B_2},$    (11)
    **end**
**end**
```

Algorithm 2: LeeBERT-CLO-v2

```
Parameters: $\Theta, \Omega$, $C$;
Return: the converged early exiting model; **while** *not converge* **do**
    **for** *t=1, ..., T* **do**
        **for** *c = 1, 2, ..., C* **do**
            **if** *c != C* **then**
                sample batch $B_1$ from $D_1$, respectively update $\Theta$ and with
                    $\Theta = \Theta - \lambda_1 \nabla_{\Theta} L_{B_1},$    (12)
                    $\Omega = \Omega - \lambda_1 \nabla_{\Omega} L_{B_1},$
            **end**
            **else**
                sample batch $B_1$ and $B_2$ from $D_1$ and $D_2$, respectively
                update $\Theta$ with
                    $\Theta = \Theta - \lambda_1 \nabla_{\Theta} L_{B_1},$    (13)
                calculate $L_{B_1}$ and $L_{B_2}$ with the updated $\Theta$, and update $\Omega$ with:
                    $\Omega = \Omega - \lambda_1 \nabla_{\Omega} L_{B_1} - \lambda_2 \nabla_{\Omega} L_{B_2},$    (14)
            **end**
        **end**
    **end**
**end**
```

Task	lr of model params	lr of learnable weights	batch size	cross-level steps
CoLA	2e-5	1e-3	16	2
MNLI	1e-5	1e-5	128	4
MRPC	1e-5	1e-4	32	4
QNLI	1e-5	1e-5	128	8
QQP	1e-5	1e-5	128	4
RTE	2e-5	1e-3	16	4
SST-2	2e-5	1e-4	128	4

Table 3: Hyper-parameter settings for each task.

Method	#Param	Speed-up	MNLI	MRPC	SST-2
With DeeBERT's entropy-based exiting strategy					
DeeBERT	12M	1.88x	81.7	87.2	90.6
LeeBERT (ours)	12M	1.92x	83.9	88.6	91.8
With Shallow-Deep's max-prob based exiting strategy					
Shallow-Deep	12M	1.95x	81.5	87.1	90.7
LeeBERT (ours)	108M	2.04x	83.7	88.9	91.7

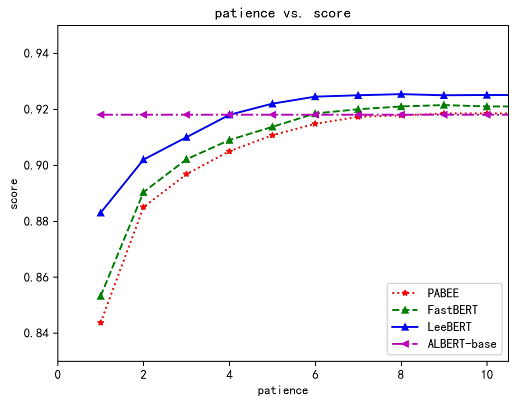
Table 4: Experimental results of LeeBERT when using different early exiting strategies.

Method	#Param	Speed-up	MNLI	MRPC	SST-2
Dev set					
BERT-base	108M	1.00x	83.5	88.3	91.5
BERT-6L	66M	1.96x	79.1	83.9	89.6
BERT-9L	87M	1.30x	80.4	85.8	90.5
DistillBERT	66M	1.96x	79.8	85.3	89.3
BERT-PKD	66M	1.96x	80.6	85.5	89.7
BERT-of-Theseus	66M	1.96x	80.7	85.4	89.6
PABEE	108M	1.86x	81.5	86.2	90.4
FastBERT	108M	1.95x	82.1	86.7	90.8
LeeBERT (ours)	108M	1.97x	83.1	88.5	91.8
Test set					
BERT-base	108M	1.00x	83.3	87.2	92.7
PABEE	108M	1.86x	81.6	85.2	91.3
FastBERT	108M	1.96x	82.0	85.7	91.7
LeeBERT (ours)	108M	1.97x	83.1	87.1	92.6

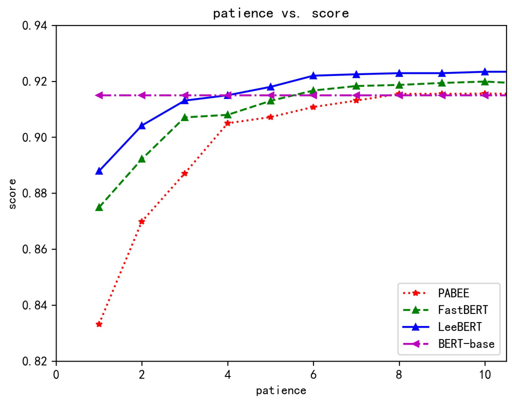
Table 5: Experimental results of models with BERT backbone on the development set and GLUE test set. The mean performance scores of 5 runs are reported. The speed-up ratio is averaged across 3 tasks. Best performances are bolded.

Method	CIFAR-10		CIFAR-100	
-	speed-up	Acc.	speed-up	Acc.
ResNet-56	1.00x	91.8	1.00x	68.6
PABEE	1.26x	91.9	1.22x	69.0
DBT	1.28	92.1	1.25x	69.3
LeeBERT	1.30x	92.5	1.27x	69.6

Table 6: Experimental results of LeeBERT when applied in the image classification tasks.



(a) ALBERT



(b) BERT

Figure 4: patience-performance curves for ALBERT and BERT on the MRPC task.