Weight Squeezing: Reparameterization for Compression and Fast Inference

Anonymous submission

Abstract

In this work, we present a novel approach for simultaneous knowledge transfer and model compression called **Weight Squeezing**. With this method, we perform knowledge transfer from a pre-trained teacher model **by learning the mapping from its weights to smaller student model weights**, without significant loss of model accuracy.

We applied Weight Squeezing combined with Knowledge Distillation to a pre-trained text classification model, and compared it to various knowledge transfer and model compression methods on several downstream text classification tasks. We observed that our approach produces better results than Knowledge Distillation methods without any loss in inference speed. We also compared Weight Squeezing with Low Rank Factorization methods and observed that our method is significantly faster at inference while being competitive in terms of accuracy.

1 Introduction

Today, deep learning has become a key technology in natural language processing (NLP), advancing state-of-the-art results for most NLP tasks. One of the significant achievements in applying deep learning in NLP is transfer learning. These methods include word vectors pre-trained on a large volume of text (Mikolov et al. 2013; Pennington, Socher, and Manning 2014), which are now commonly used to initialize the first layer of neural networks for transfer learning. In recent times, methods such as ULMFiT (Howard and Ruder 2018) have advanced the field of transfer learning in NLP by using language modeling during pre-training. Pre-trained language models now achieve state-of-the-art results on a diverse range of tasks in NLP, including text classification, question answering, natural language inference, coreference resolution, sequence labeling, and more (Qiu et al. 2020).

Another breakthrough in NLP occurred after the introduction of Transformer (Vaswani et al. 2017) neural networks that consist mostly of linear layers that compute the attention between model inputs. Unlike recurrent networks, Transformers have no recurrence over the spatial dimensions, making it possible to significantly increase their size and reach state-of-the-art results for several tasks. (Devlin et al. 2019) introduced BERT (Bidirectional Encoder Representations from Transformers), a language representation model trained to predict masked tokens in texts from unlabeled data. Pre-trained BERT can be fine-tuned to create state-of-the-art models for a wide range of NLP tasks.

While BERT is capable of learning rich representations of text, using it for solving simple downstream tasks could be excessive. This is especially important when running downstream models on edge devices such as mobile phones. A common approach in such cases is model compression.

In this work, we present a novel approach to performing simultaneous transfer learning and model compression that we called **Weight Squeezing** In this method, we propose learning the mapping from the weights of a text classification teacher model based on pre-trained BERT to the weights of a much smaller student model. We compared the proposed method with common approaches for model compression, including variations of Knowledge Distillation and low-rank matrix factorization. We evaluated both Weight Squeezing combined with Knowledge Distillation and training Weight Squeezing on its own. Our experiments show that in most cases, Weight Squeezing achieves better performance than other baseline methods. It also outperforms models trained from scratch.

2 Motivation & Problem Setting

In this section we describe our motivation for building lightweight models.

Firstly, we focus on model size. For instance, a typical mobile app has the size of 100Mb, which significantly restricts the model size that can be used on the device[1]. Smaller models are also better for serving server-side.

Secondly, we keep an eye on model inference time. A model may be efficient in terms of training (i.e., by having fewer parameters), but have a significant computational overhead regardless. Therefore, our focus is on making the student model faster than the teacher one.

Finally, we take into account that access to training resources, such as data and computing power, can be limited. Therefore, by focusing on task-specific compression, we limit

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]We managed to obtain models around 2.1 Mb in size, while the vanilla 6-layer BERT model has a size of around 256 Mb.

the data we use to what we have for specific tasks only (see 3.2 for details).

3 Related Work

3.1 Transfer Learning & Unsupervised Pre-training

Self-supervised training as unsupervised pre-training became one of the key techniques for solving NLP tasks. While the amount of labeled data can be limited, we often have unlabeled data at our disposal. This data can be effectively utilized for pre-training some parts of the models. One way to conduct self-supervised pre-training of NLP models is language modeling. It can be performed using autoregressive models when learning to predict the next words in a text based on previous words.

Transformer and BERT. The Transformer was introduced as a type of neural network architecture for machine translation (Vaswani et al. 2017). While the Transformer decoder can be used to train a language model, the performance of such a model could suffer because of its autoregressive nature. Masked Language Modeling (MLM) with BERT (Devlin et al. 2019) was proposed as an alternative to Language Modeling. This method involves training a model to predict masked words based on unmasked ones. The output of such a model depends on all words in the input text, which makes it capable of learning more complex data patterns.

3.2 Model Compression

The network architecture can be considerably over-parameterized and therefore inefficient in terms of memory and computing resources. Trading accuracy for performance and model size is often reasonable, which brings us to model compression techniques. There are many approaches (Ganesh et al. 2020; Qiu et al. 2020) to compressing BERT, including pruning (Sajjad et al. 2020; Fan, Grave, and Joulin 2019; Guo et al. 2019; Gordon, Duh, and Andrews 2020; Voita et al. 2019; McCarley, Chakravarti, and Sil 2019), quantization (Zafrir et al. 2019; Shen et al. 2019), parameter sharing (Lan et al. 2019), and knowledge distillation. Some of these methods can be combined to achieve better results (Mao et al. 2020).

Low-rank Matrix Factorization. Low-rank matrix factorization approaches focus on reducing the size of model parameters. These approaches include Singular-Value Decomposition (SVD), Tensor Train (TT) Decomposition (Oseledets 2011), and others.

One of the most notable examples of using low-rank matrix factorization methods for NLP is reducing the size of the embedding matrix, which contains a significant part of model parameters. Instead of projecting one-hot vectors directly into the hidden space, (Lan et al. 2019) first project them into a lower-dimensional embedding space, and only then project it into the hidden space. (Khrulkov et al. 2019) parameterize embedding layers based on the TT decomposition. (Shu and Nakayama 2019) learn discrete codes to represent embeddings. (Acharya et al. 2019) also use low-rank matrix factorization on the word embedding layer during training.

These methods can be considered a separate case of the pruning approach where we try to reduce the size of parameters by removing parts that can be considered unnecessary. For SVD, we drop some part of the weights which are related to small singular values, while TT can be seen as a generalization of SVD.

Knowledge Distillation. In the Knowledge Distillation (KD) approach (Ba and Caruana 2014; Hinton et al. 2015; Romero et al. 2014) a smaller *student* model is trained to mimic a *teacher* BERT model. Current KD approaches for BERT can be categorized by what exactly they try to match as follows: distillation on encoder outputs/hidden states: (Zhao et al. 2019; Jiao et al. 2019; Sun et al. 2020, 2019; Sanh et al. 2019), distillation on model output (Zhao et al. 2019; Sun et al. 2019; Sanh et al. 2019; Jiao et al. 2019; Chen et al. 2020), distillation on attention maps (Sun et al. 2020; Jiao et al. 2019).

KD can also be split into two categories: task-agnostic and task-specific. **Task-agnostic KD** involves reducing the size of BERT itself. Most methods fall under this type of compression. The KD methods, such as DistilBERT (Sanh et al. 2019) or MobileBERT (Sun et al. 2020) are trained on the same corpus as the one used when pre-training a BERT model from scratch. A typical scenario when using models such as BERT is to take a model already pre-trained on a very large corpus of data, since training or fine-tuning BERT is computationally expensive and could require a considerable amount of time. In some cases, even storing a large corpus, not to mention using it for training, could be a problem in this task.

Instead of compressing BERT itself, a different approach of **task-specific KD** can be taken by fine-tuning BERT on a downstream task first and then applying compression techniques to train a smaller model. (Tang et al. 2019) use KD to train a single layer BiLSTM student model from BERT. (Mukherjee and Awadallah 2019) show that when given a large amount of unlabeled data, student BiLSTM networks can even match the performance of the teacher. (Turc et al. 2019) pre-train compact student models on unlabeled data and then apply KD.

4 Weight Squeezing

We now introduce a method to perform knowledge transfer and model compression by **learning the mapping between teacher and student weights**.

We start with a pre-trained teacher Transformer model with a large hidden state. In our experiments, we used teacher models obtained by fine-tuning pre-trained BERT models with a classifier on top. It implies that for some linear layer l, we have a weight matrix Θ_l^t with the shape $n \times m$.

We explore a case where the weights of a pre-trained teacher model are too big for running and storing the model on an edge device. For this reason, we may want to train a student model with a smaller number of parameters. Let's say that we want the student model to make the weight matrix Θ_l^s at the same layer l to have the shape equal to $a \times b$, where $a < n$ and $b < m$.

In this work, we propose reparameterizing student weights Θ^s_l as follows:

$$\Theta^s = \mathcal{L}\Theta^t\mathcal{R} \quad (1)$$

where $\mathcal{L}$ and $\mathcal{R}$ are randomly initialized trainable parameters of the mapping (here and below, we omitted the l subscript for simplicity. However, we reparameterize each parameter layer-wise, and each layer has its own mapping). Note that if Θ^s is $a \times b$ matrix and Θ^t is $n \times m$ matrix, then $\mathcal{L}$ and $\mathcal{R}$ are $a \times n$ and $m \times b$ matrices.

In this approach, instead of training student model weights from scratch, we reparameterize them as a trainable linear mapping from teacher model weights. Doing so allows us to transfer knowledge stored in the teacher weights to the student weights.

At the same time, mapping of teacher biases and word embeddings is performed as a single linear mapping as follows:

$$\Theta^s_{single} = \Theta^t\mathcal{R} \quad (2)$$

where biases are matrices of size $1 \times b$ and word embeddings have size $V \times b$, and V is the total number of words in the vocabulary. This reparameterization for word embeddings can be seen as a linear alignment from pre-trained embeddings.

After reparameterization of the student model weights using 1 or 2, we train mapping weights $\mathcal{L}$ and $\mathcal{R}$ using plain negative log-likelihood (Weight Squeezing) or KD loss (Weight Squeezing combined with KD). When the mapping is trained, we compute student weights and use them to make predictions.

4.1 Comparison to Similar Methods

Knowledge Distillation. Weight Squeezing (WS) is orthogonal to the Knowledge Distillation method, and the two methods can be combined to achieve better results. We also observed that, in some cases, the stand-alone WS approach performs better than Knowledge Distillation.

We believe that the reason for that is that the nature of the WS approach allows the model to drop some of the information stored in the teacher parameters if it is excessive, similar to what is done by pruning methods. At the same time, KD methods based on encoder outputs and attention maps (Jiao et al. 2019; Sun et al. 2020, 2019; Sanh et al. 2019) involve training student models to mimic the behavior of the teacher, even if the capacity of the small student model is not enough to store the knowledge of the larger teacher model. To address this issue, we can use additional hyperparameters to evaluate the loss function for the KD method (e.g., extra weights to add balance between plain negative log-likelihood, KL-Divergence between the student and teacher predictions, and other regularization terms). However, it makes the training process more complex and less stable, while Weight Squeezing does not have this problem.

Low-rank Matrix Factorization. Compared to low-rank matrix factorization methods, we do not lose any information stored in the teacher weights when using Weight Squeezing. In essence, even if we map teacher model weights to the weights of a significantly smaller student model, we still have the teacher parameters represented as-is, thus allowing the student model to use all the information stored in them.

In comparison, in low-rank matrix factorization methods, we lose part of the knowledge stored in the teacher model every time we reduce the rank of the factorization. It leads to a trade-off between the number of parameters in the student model and the amount of teacher knowledge that we can transfer.

Consider two teacher models, with $1,000,000$ and $10,000,000$ parameters, respectively. Let's assume that both teachers have the same structure, number of layers and other parameters, with the only difference being the size of their hidden layers. We want to compress both of them into a student model with $100,000$ parameters. The compression of a smaller teacher model could be significantly more accurate than the compression of a bigger one, since the rank of the factorization will be different for these cases. This suggests that factorization methods can be an inappropriate choice for some compression tasks.

Furthermore, in our experiments, we observed that factorization methods could be hard to train due to their large memory footprint. For example, we want to apply factorization using SVD (see 5.2 for details), for which we define the weights of linear layers as follows:

$$\hat{\Theta} = U_{m\times r}\Sigma_{r\times r}V^\top_{r\times n} = \mathcal{U}_{m\times r}V^\top_{r\times n}$$

We now need to evaluate the result of applying this layer to some input $x\hat{\Theta}$ (where x and $\hat{\Theta}$ are $b \times m$ and $m \times n$ matrices, respectively, and b is the batch size of the input).

The first option is to **evaluate a full weight matrix** first and then multiply the resulting matrix by x:

$$x\hat{\Theta} = x(\mathcal{U}V^\top) \quad (3)$$

The second option is to **evaluate** this layer **in a sequential manner**:

$$x\hat{\Theta} = (x\mathcal{U})V^\top \quad (4)$$

However, for Equation 3, we will create a matrix in the computational graph of size $m \times n$ (weight size of the teacher model), which we have to store for backward propagation and for which we have to evaluate gradient. While for Equation 4 we do not have to store a big matrix in the computational graph, we do have to store a $b \times r$ matrix, the size of which, for now, depends on the batch size. Therefore, if the batch size is big enough, method 4 could require even more memory for **training** than in Equation 3.

For TT Decomposition, this leads to a more drastic difference in memory used for model training. Since the evaluation of matrix multiplication for TT cores can be defined sequentially, like for Equation 4 (Novikov et al. 2020), it will create $d - 1$ matrices (where d is the number of cores), the size of which depends on the batch size.

We would also like to note that the factorization rate r cannot be smaller than 1, which is why for every specific model there is a practical limit to the compression that can be performed.

Furthermore, regardless of the factorization rate r, the result after applying the factorized layer will always retain its

original size. Thus, some operations in Transformer are not going to benefit from the factorization of the model weights. For example, if we have a Transformer with the size of the hidden layer equal to 1024, then even with r equal to 1, self-attention will be computed between vectors with shape 1024. Because of this, low rates of r did not lead to a performance boost.

In our experiments, we were able to evaluate layers sequentially for the SVD method when the teacher models were small. However, in some cases, we had to evaluate the full layer weights to train student models. We always had to evaluate the full weights for TT models.

Weight Squeezing is also a method that requires evaluating the full weight matrix first. However, WS does not lead to a large memory footprint since the resulting layer's size is smaller than that of the original teacher network. For inference, we could pre-evaluate student weights once and then use them without any computational overhead. Therefore, in some cases, low-rank methods could be considered an inappropriate choice to train small models for running them in an environment with limited resources. Nevertheless, we still consider these methods as baselines and provide their experimental results for reference.

5 Experiments

5.1 Training Details

We trained all models on **AG's News**, **DBPedia**, and **Yelp Reviews** datasets. The datasets are described in section A.1 of the technical appendix.

We used manual tuning to select the model hyperparameters, with up to 20 hyperparameter search trials. Hyperparameter search ranges are shown in Table 9. The hyperparameters used in the best-performing models are shown in 5.2 and in the technical appendix A.3.

We selected hyperparameter configurations with the best performance, measured by model accuracy on the validation set during the training phase. We then evaluated the models on the test set and showed the results. We repeated each experiment 5 times and showed the mean and standard deviation values of the measurements.

Since we focused on making models smaller in terms of the overall number of parameters, we trained classifiers with 6 Transformer layers, 2 self-attention heads, and hidden size selected from the range $[16, 32, 64, 128]$ (full list of the number of the model parameters can be found in Table 1).

For knowledge transfer, we use teachers fine-tuned to downstream tasks. The teacher models were initialized from BERT models pre-trained on the Masked Language Modelling task. For each task, we experimented with two teacher models with 6 layers, 768 and 128 hidden sizes, and 12 and 2 self-attention heads, respectively (we will refer to these sizes as 768H12A and 128H2A later).

5.2 Baselines

We experimented with various approaches to applying Weight Squeezing. These approaches include:

1. Standalone Weight Squeezing (WS)
2. Weight Squeezing paired with Knowledge Distillation (WS + KD)

We compared WS with the following methods:

1. Training a student model of an appropriate size from scratch (Scratch)
2. Knowledge Distillation (KD)
3. Knowledge Distillation on Encoder Output (KD-EO)
4. Low-rank Matrix Factorization combined with Knowledge Distillation (SVD + KD, TT + KD)

Training from Scratch. For the first experiment, we trained models of appropriate hidden sizes from scratch without utilizing any pre-training.

Teacher Models. We took BERT models pre-trained on the MLM task with configurations 768H12A and 128H12A and fine-tuned them on downstream classification tasks. We used these pre-trained teacher models as the source for model compression techniques in our experiments.

We also fine-tuned a **DistilBERT** model[2]. Note that the number of parameters in the DistilBERT model is approximately equal to the number of parameters in the **768H12A** teacher model used in our experiments, which makes it more difficult to use for edge devices compared to models trained with WS and KD. Therefore, its results are provided only for reference.

Knowledge Distillation. We applied the Knowledge Distillation approach to teacher models for training the student model with a smaller hidden size to **match the teacher's predictions** (KD in Tables 3, 4, 5, 6). The resulting loss function is defined as:

$$-\alpha \log(p_c^s) - \beta \sum_i p_i^t \log(p_i^s) \quad (5)$$

where p^t and p^s are the output probabilities of student and teacher models, p_i^t, and p_i^s are the i-th components of teacher and student predictions, respectively, and c is the index of ground truth label. Thus, the first term is a negative log-likelihood of the student predictions, while the second stands for part of KL-Divergence between teacher and student predictions, which we could optimize with respect to student parameters. In all experiments, α and β values were equal to 0.2 and 1.0, respectively.

We also experimented with Knowledge Distillation **on encoder outputs** (KD-EO in Tables 3, 4, 5, 6), where we trained the linear mapping of the student's hidden states to those of the teacher after every Transformer layer. Training this mapping can be considered a generalization of matching values of student and teacher hidden states when they have different sizes and cannot be compared directly.

The resulting loss function is defined as:

$$-\alpha \log p_c^s - \beta \sum_i p_i^t \log p_i^s + \gamma \sum_j L2(h_j^t, f_j(h_j^s))$$

[2]The WS method can be used to compress BERT itself, and the compressed model then fine-tuned on a downstream task. However, applying WS to compress BERT is not the topic of this work.

Figure 1: Plots showing the accuracy of trained models. For all models, except the one trained from scratch, we took the maximum value between the two results shown for models trained with 768H12A and 128H2A teachers. Note that the **SVD and TT** methods perform **slower** during inference on CPU and GPU (see 5.4). **T** stands for the reference performance of the teacher model.

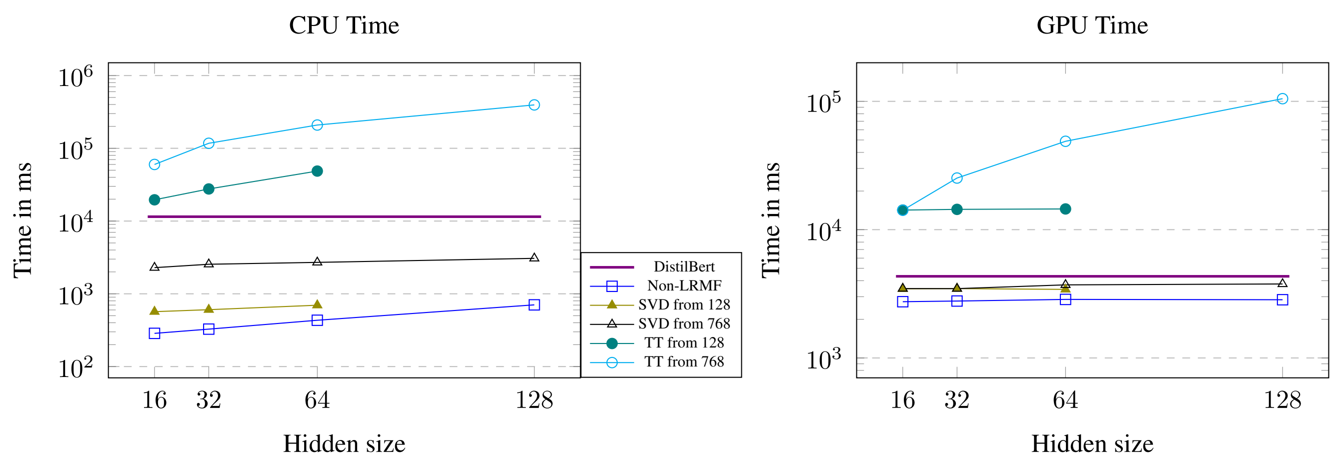

Figure 2: Plots showing the performance time of models, compressed by Non-LRMF methods or SVD/TT. Time performance showed in fastest setting: 128 sequence length, batch size 1. Teacher model times are approximately equal to DistilBERT.

where the first two terms in the equation above correspond to terms in Equation 5, while the last term is the mean squared error between teacher hidden state h_j^t from layer j and mapped student hidden state $f_j(h_j^s)$ from the same layer. We optimized this loss with respect to student parameters and parameters of each mapping f_j. We used γ value equal to 1000 in our experiments, while α and β values differed from the plain KD setting and were equal to 1 and 10, respectively. Note that there are several hidden states in each layer corresponding to different words in the input sequence. For this reason, we used the first hidden state of this sequence to evaluate the loss function.

Weight Squeezing. For **plain Weight Squeezing** (WS in Tables 3, 4, 5, 6) setting, we used fine-tuned teacher models as the source of mapping for weights reparameterization. This way, we reparameterized the parameters of all linear layers as in Equation 1 and embedding vectors as in Equation 2. Weights of the mapping for linear layers were initialized as Xavier Normal, while the mapping for the embedding matrix was initialized as Xavier Uniform. We trained the resulting model using plain negative log-likelihood. We optimized the likelihood with respect to the mapping parameters used to reparameterize the student model parameters. We also optimized the rest of the parameters of the student model which were not reparameterized (e.g, the weights of the layer normalization).

We also trained a model using **Weight Squeezing with Knowledge Distillation** (WS + KD in Tables 3, 4, 5, 6). Instead of using negative log-likelihood as the loss function, we used the KD loss function 5. In our experiments, we used the same α and β, equal to 0.2 and 1.0, respectively.

Low-rank Matrix Factorization. We also experimented with the matrix factorization approach (LRMF) and applied **Singular Value Decomposition** (SVD + KD in the results tables) to its weights.

$$\Theta = U_{m\times m}\Sigma_{m\times n}V_{n\times n}^\top$$

where Σ is a diagonal matrix of the singular values, and U and V are the left and right singular vectors of the weight. Note that if Θ is $m \times n$ matrix, then U is $m \times m$, Σ is $m \times n$ and $V^\top$ is $n \times n$ matrices.

However, we could obtain the reduced form of this weight by keeping only r largest singular values as follows:

$$\hat{\Theta} = U_{m\times r}\Sigma_{r\times r}V_{r\times n}^\top = \mathcal{U}_{m\times r}V_{r\times n}^\top$$

If we were to perform this operation on the weights of the teacher model, then instead of storing nm parameters for each weight, we would have to keep only $r(n+m)$ parameters. If r value is small enough, then the total number of parameters will be reduced compared to the original model, without introducing computation overhead.

In our experiments, we applied SVD to the parameters of the teacher model, obtained its reduced form, and then fine-tuned it using the loss function from the Equation 5 with α and β, equal to 0.2 and 1.0, respectively.

We also applied **TT Decomposition** (TT + KD in the results tables) to compress teacher models. We decomposed teacher weights using TT with 4 cores. For training, we also used the loss function from the Equation 5 with the same α and β, equal to 0.2 and 1.0, respectively.

Note that in these approaches, we do not directly train the student model with the specified hidden state size as in all the methods above. Instead, we injected a bottleneck in the middle of each layer, which allowed us to reduce the total number of parameters in the model. For this reason, we compared Weight Squeezing trained from the **128H2A** teacher BERT model with hidden states with sizes in range **[16, 32, 64]** with the SVD model with r equal to **[10, 22, 50]** and with maximum r equal to **[25, 50, 128]** for the TT approach to reach a similar number of parameters in both of these approaches. For the **768H12A** teacher model with hidden states in the range **[16, 32, 64, 128]**, we used max r equal to **[5, 16, 28, 49]** for TT. For SVD trained from the **768H12A** teacher model, we compared **[16, 32, 64, 128]** models with r equal to **[1, 8, 16, 40]**.

5.3 Expected Validation Performance

Using the model evaluation method proposed in (Dodge et al. 2019), we calculated the expected validation performance given a number of hyperparameter trials for all methods on the AgNews dataset to further explore the results' significance.

The results are shown in Figure 3. We observed that WS achieved better accuracy with fewer hyperparameter trials than any other method.

For the experiments, we compressed 128H2A teacher model into a student classifier with a hidden state size of 16. For LRMF methods, we used the appropriate factorization ranks to make SVD and TT approaches have approximately the same number of parameters as a plain classifier. Hyperparameter search ranges are shown in Table 9 in the technical appendix.

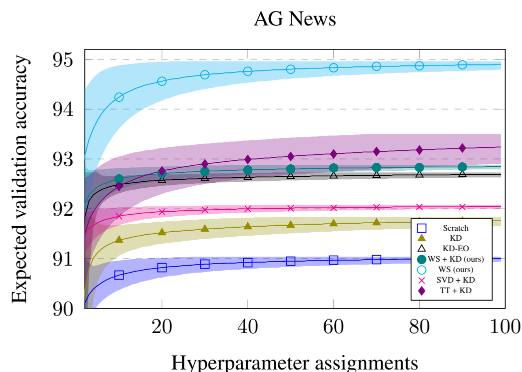

Figure 3: Expected validation accuracy for WS and baselines.

5.4 Inference Speed Measurements

For inference speed measurement, we compared Non-Low Rank Matrix Factorization models (including KD, KD-EO, WS and models trained from scratch) with models factorized using the TT and SVD methods.

In our experiments, we used a classifier with the hidden layer size equal to 16. For the TT method, we performed decomposition with 4 cores and a maximum decomposition rank equal to 5 and 25 for 768 and 128 original model hidden layer size, respectively. For the SVD method, we decomposed matrices with rank equal to 10. All models have a comparable number of parameters (see Table 1).

We evaluated models on sequences with input lengths of 128 and 512. For GPU measurements, we ran the models on 1000 samples, and on 100 samples for CPU. In all experiments, we tested the speed with batch sizes equal to 1 and 16. We repeated the measurements 5 times and reported the mean and standard deviation values of the total computation time of the models. For CPU measurements, we used 1.8 GHz Intel DualCore i5, while for GPU measurements, we used NVIDIA Tesla T4.

All models were prepared with PyTorch JIT compilation, since we found that it slightly increases the speed of all models in this experiment.

6 Results

6.1 Accuracy

Plots showing the accuracy of trained models can be found in Figures 1 and 1. See Tables 3, 4, 5, 6 in the technical appendix for the full list of the results.

While using WS combined with KD increases the performance of the model, plain WS outperformed plain KD method in every experiment. We showed that plain WS method reached results comparable with KD-EO method, achieving better results in some experiments.

We observed that SVD and TT methods were competitive to WS, outperforming most baselines on the Yelp-f and the Yelp-p datasets. However, WS outperformed both SVD and TT on the AgNews and the DBPedia datasets in most of the experiments.

Our findings show that LRMF methods suffer from low factorization rates r. For example, r equal to 50 for SVD trained from 128H2A teacher leads to significantly better results than r equal to 16 for SVD trained from 768H12A teacher. Both of these models have an equal number of parameters.

All models outperformed the model trained from scratch, except for LRMF methods with small (8 and 1 for SVD, and 16 and 5 for TT) factorization ranks trained from 768H12A teacher.

We conducted a study on the expected validation performance on the AgNews for all methods and provided the results in 5.3. We observed that WS achieved better accuracy with a lower number of hyperparameter trials than any other method.

6.2 Inference Speed

We observed that non-LRMF methods, including WS, outperformed SVD and TT methods in every experiment (see Figure 2). On CPU, SVD is **1.54-20.7** times slower than non-LRMF methods, and TT is **50.9-560.0** times slower. On GPU, SVD is **1.2-8.49** times slower, and TT is **5.07-417.0** times slower than non-LRMF methods in the experiments conducted.

The detailed results are reported in Tables 7 (CPU) and 8 (GPU) in the technical appendix.

6.3 Model Size

We compared the number of model parameters for the models used to the number of parameters for proposed method (See Table 1). The Plain BERT row stands for the number of parameters in the ordinary classifier, which was built on top of the BERT model with a specific number of layers and self-attention heads. The number of teacher model parameters is shown in bold.

The DistilBERT model has an approximately equal number of parameters to the teacher model. This makes it more difficult to use for inference on edge devices when model size matters. The KD approach has the same amount of parameters as the models in the Plain BERT row.

For the TT and SVD methods, we show the number of parameters for the specific factorization rates used to make these approaches have a number of parameters approximately equal to the plain classifier (see 5.2 for more details).

Note that **after the WS model is trained, we no longer have to evaluate the mapping result**, and the number of parameters for the inference setting is equal to the Plain BERT row for the appropriate model parameters.

SA Heads:	2				12
Hidden size:	16	32	64	128	768
Plain BERT	0.5M	1.0M	2.3M	**5.2M**	**67.0M**
WS (from 128)	0.8M	1.6M	3.2M	6.3M	-
WS (from 768)	4.7M	9.5M	19M	39M	-
SVD (from 128)	0.5M	1.1M	2.3M	-	-
SVD (from 768)	0.6M	1.1M	2.3M	5.1M	-
TT (from 128)	0.5M	1.1M	2.2M	-	-
TT (from 768)	0.5M	1.1M	2.2M	5.2M	-

Table 1: Number of parameters for each model.

7 Conclusion & Future Work

We introduced Weight Squeezing, a novel approach to knowledge transfer and model compression. We showed that it can be used to compress pre-trained text classification models and create compelling lightweight and fast models. We showed this approach could be a competitive alternative to Knowledge Distillation methods.

While the current work was focused on transferring knowledge to task-specific models, we would like to apply Weight Squeezing for task-agnostic compression for creating more applicable BERT models trained for tasks such as Masked Language Modelling.

We are currently experimenting with the initialization of mappings and plan to continue this research.

We are also interested in applying this method in domains beyond NLP to compress other types of layers (convolutional, etc.).

References

Acharya, A.; Goel, R.; Metallinou, A.; and Dhillon, I. 2019. Online embedding compression for text classification using low rank matrix factorization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 6196–6203.

Ba, J.; and Caruana, R. 2014. Do deep nets really need to be deep? In *Advances in neural information processing systems*, 2654–2662.

Chen, D.; Li, Y.; Qiu, M.; Wang, Z.; Bofang Li, B. D.; Deng, H.; Huang, J.; Lin, W.; and Zhou, J. 2020. AdaBERT: Task-Adaptive BERT Compression with Differentiable Neural Architecture Search. In *arXiv preprint arXiv:2001.04246*.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–4186.

Dodge, J.; Gururangan, S.; Card, D.; Schwartz, R.; and Smith, N. A. 2019. Show Your Work: Improved Reporting of Experimental Results. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2185–2194.

Fan, A.; Grave, E.; and Joulin, A. 2019. Reducing Transformer Depth on Demand with Structured Dropout. In *ICLR*.

Ganesh, P.; Chen, Y.; Lou, X.; Khan, M. A.; Yang, Y.; Chen, D.; Winslett, M.; Sajjad, H.; and Nakov, P. 2020. Compressing Large-Scale Transformer-Based Models: A Case Study on BERT. In *arXiv preprint arXiv:2002.11985*.

Gordon, M. A.; Duh, K.; and Andrews, N. 2020. Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning. In *arXiv preprint arXiv:2002.08307*.

Guo, F.-M.; Liu, S.; Mungall, F. S.; Lin, X.; and Wang, Y. 2019. Reweighted Proximal Pruning for Large-Scale Language Representation. In *arXiv preprint arXiv:1909.12486*.

Hinton, G.; Vinyals, O.; ; and Dean, J. 2015. Distilling the knowledge in a neural network. In *arXiv preprint arXiv:1503.02531*.

Hinton, G. E.; Srivastava, N.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. R. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* .

Howard, J.; and Ruder, S. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 328–339.

Jiao, X.; Yin, Y.; Shang, L.; Jiang, X.; Chen, X.; Li, L.; Wang, F.; and Liu, Q. 2019. TinyBERT: Distilling BERT for Natural Language Understanding. In *arXiv preprint arXiv:1909.10351*.

Khrulkov, V.; Hrinchuk, O.; Mirvakhabova, L.; and Oseledets, I. 2019. Tensorized embedding layers for efficient model compression. In *arXiv preprint arXiv:1901.10787*.

Lan, Z.; Chen, M.; Goodman, S.; Gimpel, K.; Sharma, P.; and Soricut, R. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *arXiv preprint arXiv:1909.11942*.

Lehmann, J.; Isele, R.; Jakob, M.; Jentzsch, A.; Kontokostas, D.; Mendes, P. N.; Hellmann, S.; Morsey, M.; Van Kleef, P.; Auer, S.; et al. 2015. DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6(2): 167–195.

Lin, Z.; Feng, M.; dos Santos, C. N.; Yu, M.; Xiang, B.; Zhou, B.; and Bengio, Y. 2017. A structured self-attentive sentence embedding. In *arXiv preprint arXiv:1703.03130*.

Liu, L.; Jiang, H.; He, P.; Chen, W.; Liu, X.; Gao, J.; and Han, J. 2020. On the Variance of the Adaptive Learning Rate and Beyond. In *ICLR*.

Mao, Y.; Wang, Y.; Wu, C.; Zhang, C.; Wang, Y.; Yang, Y.; Zhang, Q.; Tong, Y.; and Bai, J. 2020. LadaBERT: Lightweight Adaptation of BERT through Hybrid Model Compression. In *arXiv preprint arXiv:2004.04124*.

McCarley, J.; Chakravarti, R.; and Sil, A. 2019. Structured Pruning of a BERT-based Question Answering Model. In *arXiv preprint arXiv:1910.06360*.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119.

Mukherjee, S.; and Awadallah, A. H. 2019. Distilling Transformers into Simple Neural Networks with Unlabeled Transfer Data. In *arXiv preprint arXiv:1910.01769*.

Novikov, A.; Izmailov, P.; Khrulkov, V.; Figurnov, M.; and Oseledets, I. 2020. Tensor train decomposition on tensorflow (t3f). *Journal of Machine Learning Research* 21(30): 1–7.

Oseledets, I. V. 2011. Tensor-train decomposition. *SIAM Journal on Scientific Computing* 33(5): 2295–2317.

Pennington, J.; Socher, R.; and Manning, C. D. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.

Qiu, X.; Sun, T.; Xu, Y.; Shao, Y.; Dai, N.; and Huang, X. 2020. Pre-trained Models for Natural Language Processing: A Survey. In *arXiv preprint arXiv:2003.08271*.

Romero, A.; Ballas, N.; Kahou, S. E.; Chassang, A.; Gatta, C.; and Bengio, Y. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550* .

Sajjad, H.; Dalvi, F.; Durrani, N.; and Nakov, P. 2020. Poor Man's BERT: Smaller and Faster Transformer Models. In *arXiv preprint arXiv:2004.03844*.

Sanh, V.; Debut, L.; Chaumond, J.; and Wolf, T. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *NeurIPS EMC2 Workshop*.

Shen, S.; Dong, Z.; Ye, J.; Ma, L.; Yao, Z.; Gholami, A.; Mahoney, M. W.; and Keutzer, K. 2019. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. In *arXiv preprint arXiv:1909.05840*.

Shu, R.; and Nakayama, H. 2019. Compressing Word Embeddings via Deep Compositional Code Learning. In *arXiv preprint arXiv:1711.01068*.

Sun, S.; Cheng, Y.; Gan, Z.; and Liu, J. 2019. Patient Knowledge Distillation for BERT Model Compression. In *ACL*.

Sun, Z.; Yu, H.; Song, X.; Liu, R.; Yang, Y.; and Zhou, D. 2020. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. In *arXiv preprint arXiv:2004.02984*.

Tang, R.; Lu, Y.; Liu, L.; Mou, L.; Vechtomova, O.; and Lin, J. 2019. Distilling Task-Specific Knowledge from BERT into Simple Neural Networks. In *arXiv preprint arXiv:1903.12136*.

Turc, I.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. In *arXiv preprint arXiv:1908.08962*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Voita, E.; Talbot, D.; Moiseev, F.; Sennrich, R.; and Titov, I. 2019. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 5797–5808.

Zafrir, O.; Boudoukh, G.; Izsak, P.; and Wasserblat, M. 2019. Q8BERT: Quantized 8Bit BERT. In *arXiv preprint arXiv:1910.06188*.

Zhang, M.; Lucas, J.; Ba, J.; and Hinton, G. E. 2019. Lookahead Optimizer: k steps forward, 1 step back. In *Advances in Neural Information Processing Systems*, 9593–9604.

Zhao, S.; Gupta, R.; Song, Y.; and Zhou, D. 2019. Extreme Language Model Compression with Optimal Subwords and Shared Projections. In *arXiv preprint arXiv:1909.11687*.

A Appendices

A.1 Datasets

AG news dataset. The AG news corpus consists of news articles from AG's corpus of news articles on the web[3] pertaining to the 4 largest classes.

DBPedia ontology dataset. DBpedia is a crowd-sourced community effort to extract structured information from Wikipedia (Lehmann et al. 2015). The DBpedia ontology dataset was constructed by picking 14 nonoverlapping classes from DBpedia 2014.

Yelp reviews. The Yelp reviews dataset was obtained from the Yelp Dataset Challenge. This dataset contains review texts with two classification tasks – Yelp-f predicting a rating the user has given, and Yelp-p predicting a polarity label.

Splitting & Preprocessing. We randomly split the data into training and validation sets. Input sequences were lowercased and encoded using pre-trained BPE with 30522 tokens. We also truncated input sequences to maximum lengths.

The sizes of these splits and the maximum input length for every dataset are shown in Table 1.

Dataset	Train	Valid	Test	Length
AG News	114k	6k	7.6k	128
DBPedia	500k	60k	70k	128
Yelp-f	600k	50k	50k	512
Yelp-p	550k	10k	38k	512

Table 2: Sizes of the train, validation, and test sets of the training data. Input sequences were truncated to maximum length

A.2 Transformer Architecture

The key part of the Transformer layer is the Self-Attention mechanism (Lin et al. 2017). Below, we provide a description of this mechanism and how a Transformer layer is constructed.

Firstly, to compute the attention score between key and query vectors, the layer uses the dot product operation. After applying the softmax function to map the result to the $[0, 1]$ interval, it is multiplied by a value vector:

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

where d_k is the dimension of keys and acts as a scaling factor, and Q, K, V are the query, key, and value vectors.

The above is how we have defined Single Head attention. To get Multi-Head Attention, the Transformer layer splits the vector spaces into equal parts. The results of every attention head are concatenated into one vector:

$$\text{MHA} = \text{Concat}\left(\text{Attention}_1, \ldots, \text{Attention}_n\right),$$

where MHA stands for Multi-Head Attention.

The reason why the mechanism is called Self-Attention is that Transformer Encoders, such as BERT, take sequence tokens as the query, key, and value vectors. The vectors computed after the Self-Attention mechanism pass through a residual connection. The result goes to the Feed-Forward Network (FFN), which is defined as follows:

$$\text{FFN}\left(X\right) = \sigma\left(XW_1 + b_1\right) W_2 + b_2,$$

where W_i, b_i are trainable parameters of FFN, X is a n-dimension vector, and σ is a non-linear function.

The output of the Transformer Layer is computed by residual connection with FFN.

A.3 Optimization Details

For all models, we used the same optimization strategy using Rectified Adam (Liu et al. 2020) and Lookahead (Zhang et al. 2019). For Lookahead, we used default α equal to 0.5 and k equal to 6. For all experiments, we selected the best performing learning rate from the range $[1\text{e}{-5}, 5\text{e}{-5}, 1\text{e}{-4}, 5\text{e}{-4}, 1\text{e}{-3}, 5\text{e}{-3}, 1\text{e}{-2}]$.

We used a linear learning rate warm-up for the first 4000 optimization steps and linearly decayed its value to 0 for the rest of the training. Betas were equal to $(0.9, 0.999)$. We also used dropout (Hinton et al. 2012) with a rate equal to 0.1. Gradients with norms larger than 1 were clipped.

We trained the teacher model with the batch size equal to 32, while the other models were trained with the batch size equal to 128.

For the **AG News** dataset, we trained the model for 50 epochs, while teacher models were trained for 5 epochs.

For **DBPedia**, we trained models for 20 epochs. The teacher model with configuration 768H12A was trained for 2 epochs, while the 128H2A teacher was trained for 10 epochs.

Yelp-f models were trained for 16 epochs. The 768H12A teacher model was trained for 3 epochs and the 128H2A teacher was trained for 7 epochs. **Yelp-p** models were trained for 15 epochs with teachers trained for 3 and 5 epochs for 768H12A and 128H2A configurations, respectively.

Models with inputs restricted to 128 tokens were trained for approximately 2-4 hours, depending on the hidden size, while training models on inputs with lengths equal to 512 took around 10-16 hours on a single NVIDIA Tesla T4 GPU. The exception is TT models, for which we used 8 NVIDIA Tesla T4 GPUs for training, with input lengths equal to 512.

[3] http://groups.di.unipi.it/ gulli/AG_corpus_of_news_articles.html

AG News				
Model ↓ Size →	16	32	64	128
Scratch	90.00 ± 0.23 (90.02)	89.94 ± 0.30 (90.04)	90.21 ± 0.16 (90.60)	90.60 ± 0.15 (91.05)
DistilBert, accuracy: 93.46 ± 0.16				
BERT 768H12A, teacher accuracy: 93.46 ± 0.11				
KD	90.41 ± 0.14 (90.47)	90.20 ± 0.39 (90.55)	90.14 ± 0.28 (90.47)	90.32 ± 0.21 (90.87)
KD Encoder Outputs	91.13 ± 0.18 (91.34)	91.43 ± 0.11 (91.70)	91.43 ± 0.39 (91.55)	91.84 ± 0.18 (92.08)
WS + KD (ours)	91.49 ± 0.23 (92.21)	92.11 ± 0.16 (92.55)	91.87 ± 0.26 (92.46)	92.20 ± 0.21 (92.40)
WS (ours)	91.54 ± 0.22 (92.20)	92.01 ± 0.11 (92.57)	91.76 ± 0.15 (92.45)	92.03 ± 0.16 (92.40)
SVD + KD	68.38 ± 0.75 (68.22)	88.88 ± 0.05 (89.43)	90.82 ± 0.22 (90.95)	91.05 ± 0.06 (91.60)
TT + KD	86.68 ± 0.28 (87.06)	89.17 ± 0.16 (89.91)	89.92 ± 0.26 (90.48)	90.63 ± 0.22 (91.19)
BERT 128H2A, teacher accuracy: 92.80 ± 0.17				
KD	91.67 ± 0.12 (91.54)	92.16 ± 0.12 (91.92)	91.82 ± 0.09 (91.88)	92.08 ± 0.13 (92.25)
KD Encoder Outputs	92.36 ± 0.10 (92.54)	92.63 ± 0.06 (92.76)	92.78 ± 0.15 (92.90)	92.86 ± 0.08 (93.14)
WS + KD (ours)	**92.54 ± 0.12 (92.77)**	**92.82 ± 0.10 (93.05)**	**92.95 ± 0.07 (93.04)**	**93.02 ± 0.06 (93.13)**
WS (ours)	92.03 ± 0.27 (92.51)	92.18 ± 0.17 (92.63)	91.83 ± 0.17 (92.48)	91.86 ± 0.23 (92.46)
SVD + KD	91.79 ± 0.01 (91.86)	92.04 ± 0.06 (92.07)	92.53 ± 0.12 (92.97)	–
TT + KD	91.76 ± 0.18 (91.93)	92.08 ± 0.24 (92.49)	92.42 ± 0.32 (92.56)	–

Table 3: Accuracy on the AG News dataset. Every model was trained five times. Mean and standard deviation values of test accuracy are shown. Mean value of validation accuracy is shown in the brackets. '-' means no experiments were conducted.

DBPedia				
Model ↓ Size →	16	32	64	128
Scratch	98.08 ± 0.20 (98.07)	98.24 ± 0.04 (98.21)	98.12 ± 0.05 (98.07)	98.48 ± 0.02 (98.45)
DistilBert, accuracy: 99.28 ± 0.01				
BERT 768H12A, teacher accuracy: 99.29 ± 0.01				
KD	98.60 ± 0.04 (98.62)	98.69 ± 0.02 (98.70)	98.67 ± 0.03 (98.65)	98.78 ± 0.01 (98.77)
KD Encoder Outputs	98.69 ± 0.03 (98.69)	98.82 ± 0.02 (98.79)	98.87 ± 0.04 (98.86)	98.99 ± 0.02 (98.97)
WS + KD (ours)	**98.85 ± 0.03 (98.83)**	**98.96 ± 0.03 (98.91)**	**99.01 ± 0.02 (98.95)**	98.99 ± 0.02 (98.96)
WS (ours)	**98.85 ± 0.01 (98.82)**	**98.95 ± 0.01 (98.91)**	98.93 ± 0.03 (98.91)	98.90 ± 0.03 (98.89)
SVD + KD	32.88 ± 0.74 (32.93)	97.49 ± 0.01 (97.46)	98.86 ± 0.02 (98.62)	**99.02 ± 0.01 (98.97)**
TT + KD	97.61 ± 0.06 (97.57)	98.57 ± 0.02 (98.59)	98.70 ± 0.01 (98.77)	98.76 ± 0.01 (98.77)
BERT 128H2A, teacher accuracy: 99.08 ± 0.01				
KD	98.58 ± 0.03 (98.57)	98.63 ± 0.03 (98.60)	98.60 ± 0.04 (98.56)	96.61 ± 1.04 (96.64)
KD Encoder Outputs	98.53 ± 0.03 (98.56)	98.67 ± 0.01 (98.69)	98.77 ± 0.03 (98.78)	98.84 ± 0.01 (98.87)
WS + KD (ours)	98.81 ± 0.02 (98.78)	98.88 ± 0.02 (98.85)	98.89 ± 0.03 (98.86)	98.89 ± 0.02 (98.86)
WS (ours)	98.79 ± 0.03 (98.76)	98.87 ± 0.02 (98.83)	98.87 ± 0.02 (98.82)	98.84 ± 0.01 (98.80)
SVD + KD	98.78 ± 0.02 (98.74)	98.92 ± 0.01 (98.86)	98.95 ± 0.02 (98.93)	–
TT + KD	98.64 ± 0.02 (98.63)	98.65 ± 0.04 (98.64)	98.52 ± 0.06 (98.49)	–

Table 4: Accuracy on the DBPedia dataset. Mean and standard deviation values of test accuracy are shown. Mean value of validation accuracy is shown in the brackets. '-' means no experiments were conducted.

Yelp Full				
Model ↓ Size →	16	32	64	128
Scratch	59.94 ± 0.14 (59.83)	60.03 ± 0.25 (59.83)	60.46 ± 0.09 (60.42)	60.51 ± 0.20 (60.44)
DistilBert, accuracy: 68.66 ± 0.04				
BERT 768H12A, teacher accuracy: 69.11 ± 0.06				
KD	61.48 ± 0.06 (62.14)	61.57 ± 0.15 (62.13)	61.43 ± 0.05 (62.02)	61.67 ± 0.13 (62.17)
KD Encoder Outputs	61.70 ± 0.05 (62.39)	62.43 ± 0.27 (63.15)	62.26 ± 0.16 (62.99)	63.25 ± 0.23 (64.29)
WS + KD (ours)	61.00 ± 0.13 (61.93)	62.69 ± 0.10 (63.66)	64.21 ± 0.20 (64.99)	67.27 ± 0.11 (68.46)
WS (ours)	60.54 ± 0.25 (61.39)	62.12 ± 0.10 (63.15)	63.09 ± 0.08 (64.01)	65.14 ± 0.61 (66.06)
SVD + KD	53.11 ± 0.14 (53.22)	62.34 ± 0.09 (61.99)	66.94 ± 0.07 (67.23)	**67.89 ± 0.18 (67.83)**
TT + KD	–	–	60.77 ± 0.15 (61.19)	60.98 ± 0.04 (61.18)
BERT 128H2A, teacher accuracy: 67.11 ± 0.07				
KD	61.82 ± 0.15 (62.05)	62.20 ± 0.16 (62.21)	62.17 ± 0.23 (62.29)	62.67 ± 0.18 (62.80)
KD Encoder Outputs	61.97 ± 0.14 (62.16)	63.62 ± 0.19 (63.74)	64.43 ± 0.60 (64.55)	66.99 ± 0.12 (67.11)
WS + KD (ours)	62.15 ± 0.19 (62.43)	65.34 ± 0.29 (65.45)	66.79 ± 0.12 (66.88)	67.32 ± 0.07 (67.34)
WS (ours)	61.28 ± 0.09 (61.55)	62.91 ± 0.18 (63.12)	65.28 ± 0.41 (65.38)	66.02 ± 0.34 (66.10)
SVD + KD	**65.98 ± 0.05 (66.03)**	**67.11 ± 0.06 (67.04)**	**67.41 ± 0.08 (67.41)**	–
TT + KD	65.40 ± 0.07 (65.42)	66.76 ± 0.11 (66.92)	66.83 ± 0.05 (66.97)	–

Table 5: Accuracy on the Yelp Full dataset. Mean and standard deviation values of test accuracy are shown. Mean value of validation accuracy is shown in the brackets. '-' means no experiments were conducted.

Yelp Polarity				
Model ↓ Size →	16	32	64	128
Scratch	93.66 ± 0.06 (93.32)	93.64 ± 0.03 (93.26)	93.66 ± 0.13 (93.27)	93.73 ± 0.02 (93.46)
DistilBert, accuracy: 93.69 ± 0.05				
BERT 768H12A, teacher accuracy: 97.44 ± 0.01				
KD	93.62 ± 0.04 (93.46)	93.64 ± 0.05 (93.32)	93.65 ± 0.08 (93.33)	93.53 ± 0.11 (93.25)
KD Encoder Outputs	94.07 ± 0.09 (94.08)	94.62 ± 0.17 (94.64)	94.51 ± 0.21 (94.55)	95.71 ± 0.21 (96.01)
WS + KD (ours)	93.55 ± 0.05 (93.24)	94.13 ± 0.13 (93.83)	94.84 ± 0.11 (94.95)	94.80 ± 0.13 (94.89)
WS (ours)	93.41 ± 0.12 (93.65)	94.25 ± 0.16 (93.96)	94.48 ± 0.08 (94.78)	94.75 ± 0.11 (94.86)
SVD + KD	92.25 ± 0.03 (92.19)	93.71 ± 0.04 (93.67)	95.59 ± 0.03 (95.67)	**96.18 ± 0.03 (96.68)**
TT + KD	–	–	93.29 ± 0.01 (93.38)	94.04 ± 0.27 (93.97)
BERT 128H2A, teacher accuracy: 96.60 ± 0.07				
KD	93.73 ± 0.07 (93.43)	93.72 ± 0.08 (93.45)	93.76 ± 0.17 (93.46)	93.85 ± 0.05 (93.53)
KD Encoder Outputs	94.07 ± 0.06 (93.99)	94.66 ± 0.13 (94.54)	94.62 ± 0.13 (94.50)	96.21 ± 0.05 (96.27)
WS + KD (ours)	94.12 ± 0.08 (94.21)	94.26 ± 0.85 (94.35)	95.91 ± 0.08 (95.96)	95.24 ± 0.48 (95.33)
WS (ours)	93.29 ± 0.07 (93.36)	93.70 ± 0.06 (93.87)	93.98 ± 0.10 (94.13)	94.48 ± 0.04 (94.63)
SVD + KD	**95.95 ± 0.03 (96.00)**	**96.24 ± 0.08 (96.25)**	**96.49 ± 0.06 (97.18)**	–
TT + KD	93.54 ± 0.03 (93.46)	93.55 ± 0.03 (93.42)	93.53 ± 0.06 (93.46)	–

Table 6: Accuracy on the Yelp Polarity dataset. Mean and standard deviation values of test accuracy are shown. Mean value of validation accuracy is shown in the brackets. '-' means no experiments were conducted.

CPU					
Method $\rightarrow$ Size $\downarrow$	Non-LRMF	SVD from 128	SVD from 768	TT from 128	TT from 768
Sequence length 128, batch size 1					
768	11255 $\pm$ 91 ms	-	-	-	-
128	**705 $\pm$ 11 ms** (**$\times$1**)	-	3072 $\pm$ 41 ms ($\times$4.36)	-	395088 $\pm$ 1338 ms ($\times$560)
64	**433 $\pm$ 5 ms** (**$\times$1**)	697 $\pm$ 20 ms ($\times$1.61)	2704 $\pm$ 60 ms ($\times$6.24)	48640 $\pm$ 4958 ms ($\times$112)	208891 $\pm$ 566 ms ($\times$482)
32	**327 $\pm$ 3 ms** (**$\times$1**)	605 $\pm$ 15 ms ($\times$1.85)	2548 $\pm$ 32 ms ($\times$7.79)	27625 $\pm$ 48 ms ($\times$84.5)	117246 $\pm$ 1066 ms ($\times$359)
16	**286 $\pm$ 6 ms** (**$\times$1**)	568 $\pm$ 6 ms ($\times$1.99)	2288 $\pm$ 23 ms ($\times$8)	19592 $\pm$ 209 ms ($\times$68.5)	59972 $\pm$ 783 ms ($\times$210)
Sequence length 128, batch size 16					
768	11283 $\pm$ 228 ms	-	-	-	-
128	**706 $\pm$ 4 ms** (**$\times$1**)	-	3886 $\pm$ 28 ms ($\times$5.5)	-	
64	**288 $\pm$ 9 ms** (**$\times$1**)	624 $\pm$ 16 ms ($\times$2.17)	3438 $\pm$ 21 ms ($\times$11.9)		N/A
32	**182 $\pm$ 1 ms** (**$\times$1**)	537 $\pm$ 19 ms ($\times$2.95)	3408 $\pm$ 21 ms ($\times$18.7)	N/A	
16	**153 $\pm$ 3 ms** (**$\times$1**)	489 $\pm$ 8 ms ($\times$3.2)	3173 $\pm$ 12 ms ($\times$20.7)		
Sequence length 512, batch size 1					
768	51306 $\pm$ 44 ms	-	-	-	-
128	**3467 $\pm$ 33 ms** (**$\times$1**)	-	24245 $\pm$ 284 ms ($\times$7)	-	1541404 $\pm$ 813 ms ($\times$445)
64	**2205 $\pm$ 20) ms** (**$\times$1**)	3427 $\pm$ 122 ms ($\times$1.55)	22777 $\pm$ 734 ms ($\times$10.3)	182859 $\pm$ 1594 ms ($\times$82.9)	838924 $\pm$ 2986 ms ($\times$380)
32	**1722 $\pm$ 7 ms** (**$\times$1**)	3042 $\pm$ 49 ms ($\times$1.77)	22549 $\pm$ 337 ms ($\times$13.1)	115513 $\pm$ 1420 ms ($\times$67.1)	472463 $\pm$ 3860 ms ($\times$274)
16	**1563 $\pm$ 10 ms** (**$\times$1**)	2889 $\pm$ 38 ms ($\times$1.85)	21353 $\pm$ 214 ms ($\times$13.7)	79516 $\pm$ 480 ms ($\times$50.9)	251326 $\pm$ 2159 ms ($\times$161)
Sequence length 512, batch size 16					
768	66557 $\pm$ 87 ms	-	-	-	-
128	**4796 $\pm$ 123 ms** (**$\times$1**)	-	38788 $\pm$ 705 ms ($\times$8.09)	-	
64	**2995 $\pm$ 26 ms** (**$\times$1**)	4604 $\pm$ 20 ms ($\times$1.54)	36671 $\pm$ 193 ms ($\times$12.2)		N/A
32	**2354 $\pm$ 30 ms** (**$\times$1**)	4219 $\pm$ 52 ms ($\times$1.79)	36795 $\pm$ 083 ms ($\times$15.6)	N/A	
16	**2175 $\pm$ 29 ms** (**$\times$1**)	4091 $\pm$ 87 ms ($\times$1.88)	35909 $\pm$ 129 ms ($\times$16.5)		

Table 7: CPU inference speed results for Non-Low Rank Matrix Factorization and (SVD, TT)-decomposed models. **Non-LRMF methods include KD, KD-EO, models trained from scratch and WS**. We compare models of comparable size. Each cell shows time in milliseconds needed for running the model on 100 samples. '-' means no experiments were conducted, while N/A means that the running times were too high to list here.

GPU					
Method → Size ↓	Non-LRMF	SVD from 128	SVD from 768	TT from 128	TT from 768
Sequence length 128, batch size 1					
768	4334 ± 28 ms	-	-	-	-
128	**2843 ± 21 ms ($\times$1)**	-	3773 ± 15 ms ($\times$1.33)	-	104939 ± 119 ms ($\times$36.9)
64	**2859 ± 17 ms ($\times$1)**	3422 ± 26 ms ($\times$1.2)	3713 ± 33 ms ($\times$1.3)	14502 ± 220 ms ($\times$5.07)	48838 ± 62 ms ($\times$17.1)
32	**2777 ± 46 ms ($\times$1)**	3473 ± 50 ms ($\times$1.25)	3740 ± 36 ms ($\times$1.35)	14392 ± 48 ms ($\times$5.18)	25251 ± 19 ms ($\times$9.09)
16	**2744 ± 21 ms ($\times$1)**	3459 ± 46 ms ($\times$1.26)	3474 ± 34 ms ($\times$1.27)	14209 ± 153 ms ($\times$5.17)	14166 ± 155 ms ($\times$5.16)
Sequence length 128, batch size 16					
768	3779 ± 45 ms	-	-	-	-
128	**259 ± 2 ms ($\times$1)**	-	1201 ± 1.1 ms ($\times$4.6)	-	107927 ± 101 ms ($\times$417)
64	**192 ± 2 ms ($\times$1)**	266 ± 3 ms ($\times$1.39)	1071 ± 1.4 ms ($\times$5.58)	12343 ± 87 ms ($\times$64.3)	50795 ± 73 ms ($\times$265)
32	**186 ± 4 ms ($\times$1)**	236 ± 2 ms ($\times$1.29)	1060 ± 0.6 ms ($\times$5.7)	7001 ± 7 ms ($\times$37.6)	24216 ± 22 ms ($\times$130)
16	**181 ± 1 ms ($\times$1)**	237 ± 2 ms ($\times$1.31)	984 ± 0.4 ms ($\times$5.44)	4381 ± 12 ms ($\times$24.2)	7853 ± 6 ms ($\times$43.4)
Sequence length 512, batch size 1					
768	17841 ± 146 ms	-	-	-	-
128	**2888 ± 18 ms ($\times$1)**	-	7949 ± 25 ms ($\times$2.75)	-	426795 ± 1066 ms ($\times$148)
64	**2879 ± 20 ms ($\times$1)**	3497 ± 50 ms ($\times$1.21)	7235 ± 7 ms ($\times$2.51)	45911 ± 101 ms ($\times$15.9)	202186 ± 180 ms ($\times$70.2)
32	**2744 ± 11 ms ($\times$1)**	3495 ± 28 ms ($\times$1.27)	7108 ± 5 ms ($\times$2.59)	28287 ± 44 ms ($\times$10.3)	100660 ± 34 ms ($\times$36.7)
16	**2756 ± 16 ms ($\times$1)**	3495 ± 28 ms ($\times$1.29)	6381 ± 8 ms ($\times$2.32)	18856 ± 37 ms ($\times$6.84)	35950 ± 11 ms ($\times$13)
Sequence length 512, batch size 16					
768	16611 ± 128 ms	-	-	-	-
128	**1415 ± 6 ms ($\times$1)**	-	7151 ± 16 ms ($\times$5.06)	-	N/A
64	**961 ± 1 ms ($\times$1)**	1365 ± 3ms ($\times$1.42)	6575 ± 0.8 ms ($\times$6.84)	49882 ± 62 ms ($\times$51.9)	204314 ± 957 ms ($\times$213)
32	**797 ± 1 ms ($\times$1)**	1264 ± 0.8 ms ($\times$1.59)	6664 ± 0.3 ms ($\times$8.36)	29513 ± 38 ms ($\times$37)	106420 ± 353 ms ($\times$134)
16	**736 ± 1 ms ($\times$1)**	1241 ± 1 ms ($\times$1.69)	6248 ± 1.1 ms ($\times$8.49)	18304 ± 18 ms ($\times$24.9)	39843 ± 54 ms ($\times$54.1)

Table 8: GPU inference speed results for Non-Low Rank Matrix Factorization and (SVD, TT)-decomposed models. **Non-LRMF methods include KD, KD-EO, models trained from scratch and WS**. We compare models of comparable size. Each cell shows time in milliseconds needed for running the model on 1000 samples. '-' means no experiments were conducted, while N/A means that the running times were too high to list here.

Parameter Name	Range
General	
learning rate	$[1e{-5}, 5e{-5}, 1e{-4}, 5e{-4}, 1e{-3}, 5e{-3}, 1e{-2}]$
epochs	$[20, 30, 50]$
warmup steps	$[0, 1000, 2000, 4000, 5000]$
Knowledge Distillation	
α	$[0.2, 0.5, 0.7, 1.0, 1.5, 2, 4, 6, 8, 10]$
β	$[0.2, 0.5, 0.7, 1.0, 1.5, 2, 4, 6, 8, 10]$
Knowledge Distillation Encoder Outputs	
α	$[0.2, 0.5, 0.7, 1.0, 1.5, 2, 4, 6, 8, 10]$
β	$[0.2, 0.5, 0.7, 1.0, 1.5, 2, 4, 6, 8, 10]$
γ	$[10, 100, 1000, 2000, 3000]$
Tensor-Train	
(tensor cores, rank)	$[(3, 21), (4, 25), (5, 27)]$

Table 9: Hyperparameter search ranges for methods used in experiments for evaluating Expected Validation Performance and the experiments from section 5. The General section lists ranges of hyperparameters used in every method. For the Knowledge Distillation methods, we provide ranges of α and β parameters used for all experiments with KD loss 5 (this includes all methods except for training a model from scratch, plain WS and KD-EO). In the TT method, we tuned the number of decomposition cores. The maximum rank of factorization was selected to make this model have a number of parameters approximately equal to that of a plain classifier with hidden layer size equal to 16 (see 5.2 for more details).