Fredrik Ekholm

Knowledge Distillation for Transformers

Computer Science Tripos – Part II

Trinity College

October 6, 2023

Declaration

I, Fredrik Ekholm of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed Fredrik Ekholm

Date October 6, 2023

Proforma

Candidate Number:	**2386G**
College:	**Trinity College**
Project Title:	**Knowledge Distillation for Transformers**
Examination:	**Computer Science Tripos – Part II, 2022**
Word Count:	**11,700 words**[1]
Line Count:	**2,882**
Project Originator:	Aaron Zhao
Supervisor:	Aaron Zhao, Robert Mullins

Original Aims of the Project

Knowledge Distillation (KD) is a technique for increasing the performance of small neural networks, by using larger pre-trained networks to guide the training. This project aimed to implement and evaluate this technique on transformer models trained on language tasks. To this end, it intended to build a framework for data processing and training, in which it is easy to set up and run knowledge distillation experiments. The proposed extensions aimed at also implementing model quantisation, comparing different KD techniques, and running more extensive evalution.

Work Completed

The project was a success. It fulfilled all success criteria, and showed that by using knowledge distillation, an improvement in accuracy can be achieved compared to a baseline model trained without KD. Further improvements were made using quantisation, the size of the base model is reduced by a factor of 15 with a less than 1% drop in accuracy on average.

To facilitate these experiments, I have implemented a pipeline for KD experiments, which improves on the existing implementations in terms of felxibility, scalability,reproducability, and testing.

Special Difficulties

None.

[1]This word count was computed by `texcount`

Contents

1 Introduction

1.1 Motivation

Neural networks have recently found many applications in different products, but their increasing size is a barrier to deploying them on resource-constrained hardware [4, 38]. The ability to reduce the size of a model without sacrificing performance would open up new use cases of highly capable AI systems. The transformer model [40] is the dominating neural network architecture for language processing and generation tasks, but state-of-the-art models take up 100s GB of space to run [5, 50, 45].

I am investigating *Knowledge Distillation* (KD) as a technique to reduce the size of transformer language models. KD involves first training a larger model to achieve good performance on a task and then guiding the training of a smaller student model be extracting knowledge from the teacher model. In particular, we feed samples from a training dataset to both the teacher and the student. The predicted class probabilities, and potentially intermediate states, are compared to compute a loss. The student learns to mimic the behaviour or the teacher by being minimising this loss.

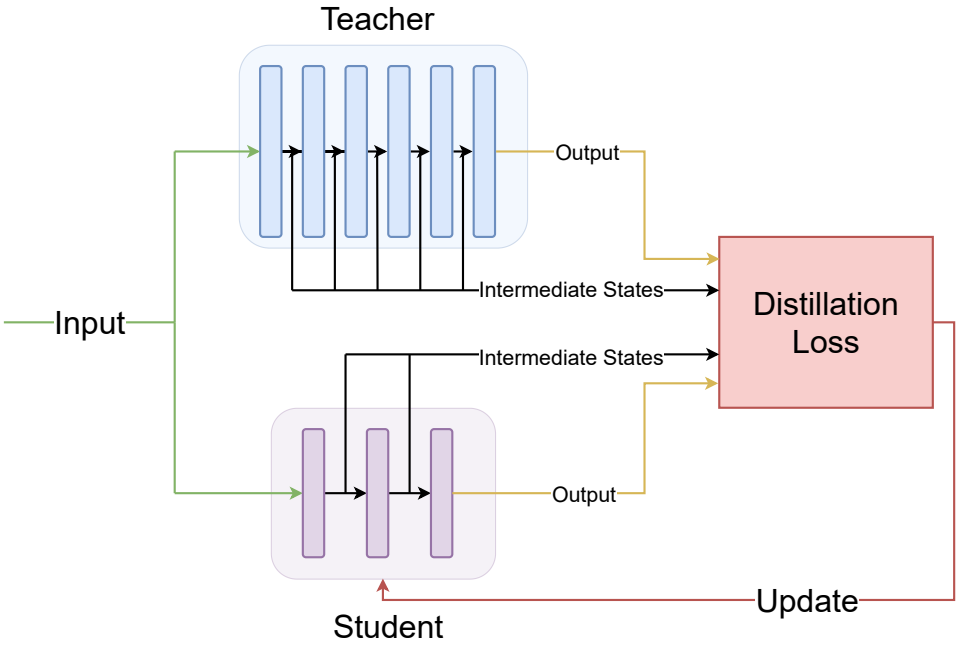


Figure 1.1: Overview of the Knowledge Distillation training procedure.

I motivate this approach by explaining that 1. We want models to be smaller, yet still maintain their performance; 2. Bigger models typically perform better; 3. Bigger models can be compressed to reduce their size; and 4. Intermediate states from the teacher model can provide useful supervision signals.

We want smaller models State-of-the-art models such as GPT3 [5] now need as much as 800GB of memory to run. Up until very recently, to run a state-of-the-art language model one would need a dedicated cluster of high-end GPUs. If we could reduce the size of the models, it would not only be much cheaper to run but it could also enable individuals to run them on personal computers and mobile phones. Furthermore, lower computational cost means lower carbon footprint. Recently, it is estimated that the inference cost of ChatGPT lead to 7 to 15 metric tons of CO_2 being released per day [30][1].

Smaller models, like the ones this project works with (10s to 100s of MBs), also benenfits from size reduction with reduced latency and computational cost (affecting e.g. battery consumption). It can also enable higher performing models tobe run them IoT and edge devices, with uses such as home assistants and wearable technology and further applications within industry, medicine, and science.

Big models are easier to train A common pattern in machine learning is that whenever you can find a large enough dataset, training a model with more parameters will give you higher performance. [35, 8]. For language based tasks, you can often pre-train your model on a large unlabelled text corpus. These are easy to get hold of, for example by aggregating public domain books or scraping the internet [12, 52]. The pre-training procedure produces a model with general langauge understanding, which can then cheaply be fine-tuned on a downstream task such as sentence sentiment classification or question answering. Even with a constant number of training iterations, the downstream performance increases with size of model [10, 34, 35]. This is not only due to the larger models' ability to model more complex functions, but also because of their better convergence [2, 24], and improved ability to generalise [6].

Big models can be compressed Large transformer language models are typically heavily over-paremeterised. [20, 24]. There is redundancy in the model weights, and multiple parts of the model being used for the same function. This redundancy helps the larger models reach higher performance when training, but means that a smaller model could represent the same function. Therefore, large models can effectively be compressed, and compressing a large model instead of training a small model can lead to better performance [24]. Even though KD trains the student model from scratch it can be viewed as a model compression algorithm, and we can thus expect a distilled model to outperform a model of the same size trained from scratch.

Intermediate states are meaningful In the case of transformers for language modelling, intermediate representations have been shown to contain significant linguistic information. [1] This indicates that intermediate representations of a higher performing larger model could be a good supervision signal to train a smaller network.

[1] equivalent to 400-800 British households

1.2 Related Work

Many different model compression techniques have been proposed to reduce the computational cost of inference. Other than knowledge distillation, commonly used ones include

- **Weight Pruning:** By removing individual weights or nodes that have low contribution to the model's performance, a smaller network is achieved. Techniques for identifying parameters to prune include simple ones like Magnitude Based Pruning [37], and more complex iterative methods like PLATON [49].
- **Quantisation:** By using a low-precision representation of model weights than the typical 32-bit floats, fewer bits are needed to store all weights we reduce memory costs. Techniques for this include fixed-point quantisation, MSFloat [36] and ternarisation [22]. The loss of precision typically causes a degredation of performance, some of which can be alleviated using Quantisation Aware Training [42].

Knowledge Distillation of transformers has been studied extensively before [27]. In this project, I'm reproducing the results from TinyBERT [18] and TernaryBERT [51].

TinyBERT introduces a scheme for comparing intermediate states between the student and teacher models for transformer knowledge distillation. Previously, KD for transformers was done using only the predicted class probablities. TernaryBERT combines the TinyBERT knowledge distillation with quantisation aware training to train a ternarised version of BERT.

Later works have improved on the KD technique. One design dimension of KD when number of layers in teacher and student differ is what layer in the teacher model to compare each student layer against. While TinyBERT employs a fixed mapping, ALP-KD [31] introduces an attention based technique for this, where the mapping is adaptive and depends on the similary of different intermediate states for each training sample.

1.3 Contributions

This dissertation provides the following contributions:

- It provides a flexible framework for running KD experiments with (potentially quantised) transformers, improving on existing implementations in terms of flexibility, scalability, reproducability and improved testing. Throughout Chapter 3 I compare against the open sourced implementation from the TinyBERT paper [18], explaining how mine differs.
- A 3x reduction of model size using standard KD, and a 14x reduction if quantised KD is used. This verifies the implementation from the TinyBERT [18] and TernaryBERT [51] papers.
- A novel experiment using a learnable linear layer mapping. My results highlight a fundamental problem with increasing expressivity of the KD loss that has received little discussion from literature, where adding more learnable parameters to the KD loss is commonplace [18**?** , 31].

2 Preparation

2.1 Background

Before 2017, RNNs [15, 7] held state-of-the-art results in language modelling and sequence to sequence tasks such as translation [3, 39]. However, RNNs suffer from limited parallelism within their computation, as they need to process all tokens sequentially. With the motivation of increasing parallelism, Vaswani et al. [40] introduced the Transformer, a sequence to sequence architecture that utilises *self-attention* to process the entire sequence at once. The self attention mechanism predicts the relevance between every pair of tokens in the input, and then sends information between tokens, weighted by the relevance (see Section 2.2). Their transformer model achieved state-of-the-art results on machine translation and English constituency parsing.

The improved parallelism compared to RNNs makes it is more feasible to pre-train transformers on a large unlabelled text corpus, giving the model a general language understanding. The pre-trained model can then be fine-tuned for text classification on a task specific dataset. Approaches following this pattern, such as GPT [34], BERT [10], XLNet [47] and RoBERTa [26], has achieved state-of-the-art results on NLP sentence classification benchmarks.

I implement Knowledge Distillation using a BERT model as teacher, and closely follow the BERT training procedure in my Knowledge Distillation approach. Therefore, I will introduce the architecture and training process of BERT in detail.

2.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a class of transformer architectures with a pre-training procedure.

The Transformer Architecture

The architecture follows closely to the original Transformers paper [40], but using only the encoder half as the model will not used for sequence to sequence tasks.

The input to the model is a sequence of tokens, encoded as integers. These are first passed to an *embedding* layer, a learnable mapping from token to real vector of length d, giving a sequence of vectors H, the `hidden state`. This sequence of vectors, one vector per input token, is then transformed by a series of l transformer layers. Each transformer layer consist of two sublayers: Multi-Head Attention (MHA) and a position-wise feed-forward network (FFN).

MHA is used to send information between positions, and aggregates the information into a new sequence of hidden states. The feed forward network can then perform arbitrary computation using that aggregated information.

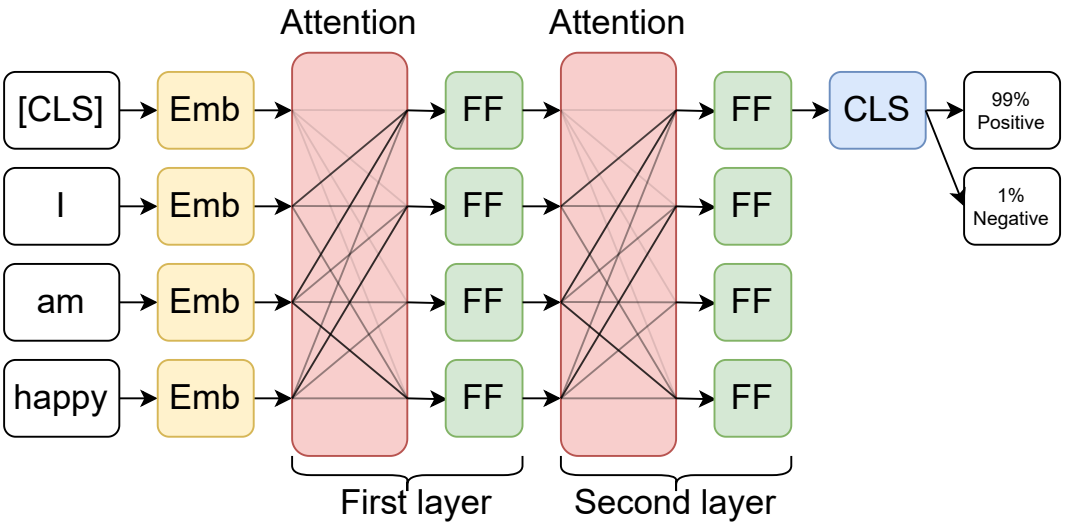


Figure 2.1: Simplified diagram of a transformer, with the embedding layer in yellow, Multi-Head Attention layers in red, Feed-Forward layers in green and the classification layer in blue.

Multi-Head Attention The attention function takes three sequences of vectors as input: Q (Query), K (Key) and V (Value). The query and key vectors are all pairwise compared, to computing an *attention score* between every pair of positions. This represents how important the information currently stored in position i is for position j. The value vector represent the information we want to send between positions. For every position, we sum up all V value vectors, weighted by the attention score, to produce the output.

The pairwise comparisons are done using dot products, and thus the matrix with attention scores can be calculated as

$$A = \frac{QK^T}{\sqrt{d_k}}$$

where d_k is the size of the key vectors and acts as a constant scaling factor. To ensure the attention scores sum to 1 for every position, we pass the attention scores through a `softmax` function. The weighted sum of value vectors is expressible as a matrix multiplication:

$$\texttt{Attention}(Q, K, V) = \texttt{softmax}(A)V$$

To increase expressivity of the attention layer, we run a attention functions in parallel. Each of these is referred to as a *head*, giving us Multi-Head Attention. For head i, vectors Q_i, K_i and V_i are computed for an input vector H using learnable linear transformations W_i^Q,W_i^K,W_i^V:

$$Q_i = W_i^Q H$$

$$K_i = W_i^K H$$

$$V_i = W_i^V H$$

Attention is applied to these matrices, and the results from all heads are concatenated together, and transformed by a learnable linear transformation W^O.

$$h_i = \texttt{Attention}(Q_i, K_i, V_i)$$

$$\texttt{MHA}(H) = \texttt{Concat}(h_1, ...h_a)W^O$$

Position-Wise Feed-Forward Network After every MHA layer, a Feed-Forward Network transforms each vector in the sequence independently by applying two linear layers with an activation function in-between. As Feed Forward Networks are universal approximators [16], we can regard this as performing an arbitrary computation using the aggregated information from the MHA layer.

2.2.1 WordPiece Tokenisation

As the input to the model is a sequence of integers, we first have to *tokenise* the input sentence(s). BERT uses a *WordPiece* tokeniser. The WordPiece tokeniser has a vocabulary consisting of 30000 common root words and word suffixes. Short words will be tokenised to a single token, while longer words might need to be tokenised to a root-token and a series of suffix-tokens.

2.2.2 Input/Output Format

The input to BERT is prepended by a `[CLS]` token, and the output at this position is used for classification (see Figure 2.1). When a sentence pair is used as input, a `[SEP]` token is used to seperate them.

2.2.3 Language Model Pre-training

As previously mentioned, it is often beneficial to first train a model to gain a general language understanding by using a large unlabelled text dataset. This way, much more training data can be produced, and a larger model can be used without overfitting, increasing performance.

When pre-training on unlabelled data, a training objective has to be defined. This should be some task for which training samples can be constructed just from text, and where good performance requires a general language understanding. OpenAI's GPT [34] used a next-token-prediction task, but Devlin et al. [10] argues that such a left-to-right architecture is limiting. Thus, for BERT they propose the pre-training objective *Masked Language Modelling*, where missing words in the middle of sentences have to be predicted. To learn to compare sentences for downstream sentence pair tasks, BERT also introduces a *Next Sentence Classification* task. In the BERT paper, pre-training is run for 40 epochs on a 25GB of data, taking 4 days to complete.

Masked Language Modelling (MLM) The main pre-training objective is prediction of masked out tokens, where the idea is that to learn to fill in missing words the model will need to develop a general language understanding. Specifically, we add a point-wise FFN layer to BERT that takes a vector final hidden states as input and outputs a probability distribution over tokens. To generate a training sample we first choose a *sentence* (any consecutive sequence of tokens) from the text corpus. 15% of the token position are then randomly chosen for prediction. As the `[MASK]` token does not appear during finetuning on a task-specific dataset, we do not mask all the positions chosen for prediction. Instead, to generate the input sequence 80% of the chosen positions are replaced with the token `[MASK]`, 10% are replaced with a random token and 10% are left as they are. The model is then trained with the original sentence as target, using CrossEntropy as loss function.

Next Sentence Classification To aid the model in learning relationships between sentences, we also train the model on predicting whether a given pair of sentences come one after another in the corpus. During training data generation we will use the a sentence sampled at random 50% of the time and the immediately following sentence 50% of the time. A classification layer is added, which takes the final hidden state in the first position as input, and returns a binary probability distribution on whether the next sentence is the immediately following one. The model training loss is the sum of the MLM loss and CrossEntropy loss of the next sentence prediction.

2.2.4 Task-specific Fine-tuning

When we have a pre-trained BERT model, it can be effectively and cheaply fine-tuned on a small task-specific dataset. We add a single linear prediction layer mapping the final hidden state at the `[CLS]` token to the number of prediction classes, and train for a few epochs using cross entropy loss.

2.3 Knowledge Distillation

The idea of Knowledge Distillation [14] is to use a larger teacher model to efficiently extract knowledge from the training data, and then use the knowledge to guide the teaching of a smaller student model. The general setup is the following (Figure 1.1): First, we train a large model to perform well on some classification task. We then feed samples from some *transfer* dataset to both the teacher and the student. The predicted class probabilities and intermediate states, are compared to compute a loss, capturing the difference between teacher and student. Gradient Descent is used to minimise this loss by updating the student, teaching it to mimic the teacher's behaviour .

This has a couple benefits, compared to training the student from scratch

- **Higher information density**: each training target is now a high dimensional vector instead of a single label, meaning more information is transferred to the student in each iteration.

- **Learning useful representations faster**: without KD, the model has to infer what makes useful intermediate representations only from the labels of the training samples. For example, it might be useful to identify the subject of the sentence for a sentence classification model. This could already encoded in the teacher's intermediate states, and thus will be quickly learnt by the student.

- **Data augmentation**: As training labels are not used for training, we can augment training samples without worrying about the appropriate label changing. For text classification tasks, this is a big benefit, and discussed in more detail in Section 2.3.3

However, the technique comes with some limitations. A major one is that the student's performance is limited by the teacher's performance, which needs to be trained without KD. We also note that the useful intermediate states in the teacher might not be the same ones as for a smaller model, and when their architecture differ it can be difficult to map the states to each other in a meaningful way.

I follow the KD-method presented in the TinyBERT paper [18]. To introduce a mathematical framework for KD, we first define f^T and f^S to be some *behaviour* functions of the teacher and student model respectively. These are functions from model input to some informative intermediate representation, and can for example be the activations of a specific layer. KD is formally modelled as minimising the following objective function

$$\mathcal{L}_{KD} = \sum_{x \epsilon \mathcal{X}} L(f^S(x), f^T(x))$$

where $L(\times)$ is a loss function evaluating the difference in behaviour function output of the teacher and student model, and $\mathcal{X}$ is the training set. During the training process, the parameters of the teacher model are kept frozen.

2.3.1 Transformer Distillation

To set up KD for transformers, suitable behaviour functions and loss functions need to be chosen. We will assume that the teacher and student both follow the BERT architecture, and will therefore use the attention matrices and the hidden states, and the model predictions as behaviour functions.

Attention based distillation We define a loss between the student's and the teacher's (pre-softmax) attention matrices. For a given pair of student and teacher layer, let A_i^S be the student's attention matrix for head i, and let A_i^T be the teacher's attention matrix. Attention based distillation loss is defined as:

$$L_{attn}(A^S, A^T) = \frac{1}{h}\sum^{h} \texttt{MSE}(A_i^S, A_i^T)$$

where `MSE` referes to the Mean Squared Error.

Hidden States based Distillation As length of hidden vectors might differ between student (d^S) and teacher (d^T), comparing them isn't as straightforward. To remidy this, we learn a linear mapping W_h from size d^S to size d^T. This mapping is learnt together with the student during KD training, and can either be the same for all transformer layers, or we can learn an individual mapping for each transformer layer of the student.

$$L_{hidn}(H^S, H^T) = \texttt{MSE}(H^S W_h, H^T)$$

Embedding layer distillation The output of the embedding layer can be seen as the initial hidden state, and is therefore already taken into account in the Hidden States based Distillation

Prediction layer distillation We add a loss term comparing the predicted logits of teacher and student. Specifically, we use soft cross entropy with the teacher logits (z^T) as target and the student logits (z^S) as prediction

$$L_{pred} = SoftCrossEntropy(z^T, z^S)$$

We use `Transformer Layer Distillation` to refer to the combination of Attention based distillation, Hidden States based Distillation, and Embedding layer distillation.

Layer Mapping

As the number of layers l_S in the student might be smaller than the number of layers l_T in the teacher, we need to decide what teacher model layer to compare each student layer against. The TinyBERT paper compares mapping the student's layers to a prefix of the teacher layers, to a suffix, or to uniformly spread them out. To be general, we define the loss for an arbitrary layer map g.

Index the layers of a model from 0 to l, where 0 refers to the embedding layer and l refers to the last transformer layer. Each layer of the student will be compared against a single layer in the teacher. Let $g(i)$ be a function mapping from student layer index to teacher layer index. We can then write the overall transformer KD loss as:

$$L_{model} = \sum_{x \in X} \sum_{i=0}^{l_S} L_{layer}(f_i^S(x), f_{g(i)}^T(x))$$

2.3.2 General and Task-specific Distillation

As described in Section 2.2.3, BERT is first pre-trained on a large text corpus and then fine-tuned on a smaller task-specific dataset. In this project, I will follow the TinyBERT training

method [18], which mirrors the standard BERT training procedure when performing knowledge distillation.

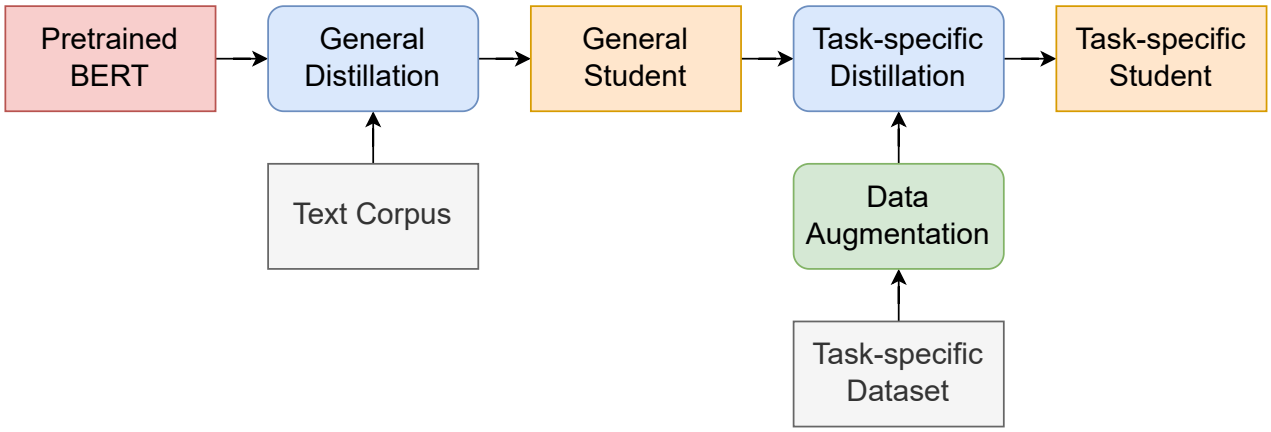


Figure 2.2: Overview of the General and Task-specific distillation

General Distillation In order to extract knowledge from a pre-trained BERT teacher model, we perform KD with the same setup as BERT pre-training. Specifically, we generate training data samples as described in Section 2.2.3. Then, the student model is trained with this data as input, using Transformer Layer Distillation. After General Distillation, we say that we have a `general student model`.

Task-specific Distillation To train the student on a specific prediction task, a BERT model that has been fine-tuned on the desired task is used as a teacher. The student is initialised using the weights from the general student model, but we add an additional layer for prediction. To train the prediction layer, we utilise prediction layer KD, but we also keep the Transformer layer KD, as the intermediate representations of the fine-tuned teacher model on this task might differ slightly from the general intermediate representations learnt in General Distillation.

2.3.3 Data Augmentation

Data augmentation is a popular technique for reducing overfitting [11, 33]. The idea is to increase the size of a training set by making random alterations of the original samples, exposing the model to more variety and making it generalise better. A challenge for data augmentation in language understanding tasks is that a small change such as swapping a single word can completely alter the meaning of the sentence, and render the label incorrect.

In KD, labels are not needed to train the student model and thus the procedure is especially suitable for data augmentation. In fact, we could arbitrarily alter the input and we would still be training the student model to imitate the teacher. However, we are primarily interested in the student's performance on samples similar to those in the training set, and thus there is a trade-off between how general the samples we train on are, and how targeted the training is to the goal.

Many different approaches for NLP data augmentation have been proposed, including word replacements/insertions/deletions/swaps [43], back translation [46], and spelling error injection. I will implement the augmentation procedure from the TinyBERT paper [18] which is based on doing random word replacements. A pre-trained MLM BERT model is used to predict suitable replacement words for every word consisting of a single token. For multi-token words, GloVe [32] embeddings are used to find suitable replacements. The algorithm is specified in Algorithm 1.

Algorithm 1: DataAugmentation

Input: $\mathbf{x}$ is a sequence of words, each word a sequence of WordPiece tokens

Parameters:

N: number of augmented samples to produce

K: number of replacement candidates per word

p_r: probability of replacing a word

Output: D', a set of augmented sentences

$D' \leftarrow []$

$C \leftarrow []$ //candidate sets

for i *in* $1 \dots |x|$ **do**

  if $x[i]$ *is single token word* **then**

    $\mathbf{x}' \leftarrow \mathbf{x}$, $\mathbf{x}'[i] \leftarrow$ [MASK]

    $C[i] \leftarrow K$ most probable words of BERT($\mathbf{x}'$)[i]

  else

    $C[i] \leftarrow K$ words with most similar GloVe embedding to $\mathbf{x}[i]$

while $|D'| < N$ **do**

  $\mathbf{x}' \leftarrow \mathbf{x}$ **for** i *in* $1 \dots |x|$ **do**

    $p \leftarrow$ random value uniformly drawn from $[0, 1]$

    if $p \leq p_r$ **then**

      $\mathbf{x}'[i] \leftarrow$ random word from $C[i]$

  $D' \leftarrow \mathbf{x}'$

return D'

2.4 Quantisation

Quantisation is a model compression technique that aims to reduce computation and memory requirements of a model by representing parameters and activations using fewer bits than the standard 32-bit floating points. As discussed in Section 1.1, large models will typically be over-parameterised, and thus heavy quantisation can often be applied with only minimal loss in accuracy. *Post-training quantisation* refers to the simpler approach of applying the quantisation just once, after training a model. 16 bit quantisation is often effective this way, but for lower precisions *Quantisation-aware training* (QAT) might be necessary to not lose significant performance. In QAT, quantisation is is simulated during training, such that the

performance of the quantised model is directly optimised. Details of how this is done are in Section 2.4.2.

Using QAT, 8-bit quantisation of BERT with little loss of performance can be achieved [48]. For even lower precision, such as 4-bit quantisation or ternarisation, where each parameter only takes on 3 different values, QAT is not enough. I will implement the procedure from the TernaryBERT paper [51], where a ternarised version of BERT is trained using knowledge distillation with the full precision model as teacher. Here I introduce the quantisation methods that will use in the implementation, and then I introduce quantisation aware training.

2.4.1 Quantisation Methods

When quantising a model, the quantisation might depend on some parameters (e.g. the unit value for a fixed point quantisation). These parameters will typically be shared among all weights in a single layer or shared among all weights in one row of the weight matrix. As there is an order of magnitude fewer number of these parameters than there are model weights, we can afford to store them in high precision with minimal overhead.

Many quantisation method requires special hardware or low level CUDA implementations to take advantage off. Thus I will be simply simulating the effect of quantisation but while storing values in float32 representation. Each quantisation method is defined by a *quantisation function*, that takes in a full precision value, and returns what the represented value would be after quantisation, as a float32.

8-bit Integer Quantisation Representing real values as 8-bit integers is a common quantisation technique. We describe symmetric and min-max quantisation, two different methods for integer quantisation. The quantised values of symmetric quantisation distribute symmetrically around 0 and need a single parameter - the maximum possible absolute value, $\max(|x|)$.

$$s = \frac{2x_{max}}{255}$$
$$Q_{sym}(x) = round(x/s) \times s$$

Min-max quantisation instead distributes quantised values evenly between $\min(x)$ and $\max(x)$, increasing accuracy when the values are not centred around zero. Two parameters, x_{min} and x_{max} need to be stored.

$$s = \frac{x_{max} - x_{min}}{255}$$
$$Q_{minmax}(x) = round((x - x_{min})/s) \times s + x_{min}$$

Ternarisation Weight ternarisation is a technique where weights can take only three values: (-1,0,1), represented by 2 bits. This lets us replace many of the expensive float point multiplications with simpler addition operation during inference.

Ternarisation depends on a single parameter α, the scaling factor. The effective quantised weights will be $\hat{\mathbf{w}} = \alpha\mathbf{b}$, where $\mathbf{b} \in \{-1, 0, 1\}^n$ with n being the number of parameters in w. We aim to find the value of α and quantised weights b_i that minimises the distances between ternarised and full precision weights $\mathbf{w}$. Thus we need to solve the following optimisation problem:

$$\begin{aligned} \min_{\alpha,\mathbf{b}} \quad & \|\mathbf{w} - \alpha\mathbf{b}\|_2^2 \\ \text{s.t.} \quad & \alpha > 0, \mathbf{b} \in \{-1, 0, 1\}^n \end{aligned} \tag{2.1}$$

Let $\mathbf{I}_\Delta(\mathbf{w})$ be a thresholding function defined by

$$\mathbf{I}_\Delta(\mathbf{w})_i = \begin{cases} 1 & \text{if } \mathbf{w}_i > \Delta \\ 0 & \text{if } \Delta > \mathbf{w}_i > -\Delta \\ -1 & \text{if } -\Delta > \mathbf{w}_i \end{cases} \tag{2.2}$$

It can be shown [17]that there is a Δ such that optimal solution to Equation (2.1) is:

$$\begin{aligned} \mathbf{b} &= \mathbf{I}_\Delta(\mathbf{w}) \\ \alpha &= \frac{\|\mathbf{b} \odot \mathbf{w}\|_1}{\|\mathbf{b}\|_1} \end{aligned} \tag{2.3}$$

Finding the optimal value of Δ requires an expensive sorting operation. However, (Li et al., 2016) [22] approximates the solution by

$$\Delta = \frac{0.7\|\mathbf{w}\|_1}{n} \tag{2.4}$$

Clipping For all quantisation methods, full-precision values might be clipped before quantisation to prevent a few large values to harm the accuracy.

2.4.2 Quantisation-aware Training

If weights would be stored only in their quantised form, we would run into significant problems during training [9, 17]. For example, if training with a low learning rate, the difference between a model before and after a gradient step might be so small that the two models are identical with quantisation. That would mean the training does not progress.

To alleviate this problem we keep full-precision weights of the model during training. Let w be the full-precision weights. In every training iteration, we calculate quantised weights $\hat{w} = Q_w(w)$ where Q_w is the combined quantisation function for all weights (which might apply different functions on different parts of the weights). The quantised weights are used to run a forward pass of the model, and calculate a loss L. A gradient of the quantised weights w.r.t loss $\frac{\partial L}{\partial \hat{w}}$ is calculated. This gradient is used to apply an update step to the full-precision weights.

Dataset	|Train|	Task	Metric	Domain
CoLA	8.5k	acceptability	Matthews corr.	misc.
SST-2	67k	sentiment	acc.	movie reviews.
MRPC	3.7k	paraphrase	F1	news.
QQP	364k	paraphrase	F1	social QA questions.
MNLI	393k	NLI	acc.	misc.
QNLI	105k	NLI	acc.	Wikipedia.
RTE	2.5k	NLI	acc.	news, Wikipedia.

Table 2.1: The seven datasets I use for evaluation

2.5 GLUE Dataset

The GLUE dataset [41] is a popular natural language understanding benchmark, consisting of nine tasks. Each task has a train split for training, a development split for local evaluation and a test split whose labels are hidden. To evaluate a model on the test split, it has to be uploaded to the GLUE submission server. For my evaluation, I will exclude the datasets STS-B, as it is a regression task, and WNLI as it has a problematic train-dev split [1]. The other datasets can be seen in Table 2.1.

The datasets span four different tasks:

- **Acceptability**: Given a single sentence, determine if it is linguistically acceptable
- **Sentiment**: Given a single sentence, determine if it's sentiment is positive or negative.
- **paraphrase**: Given a pair of sentences, determine if one is a paraphrase of the other
- **NLI (Natural Language Inference)**: Given a pair of sentences, determine if the second sentence can be inferred from the first sentence.

2.6 Requirements Analysis

The project aims to achieve the following:

- An implementation of an end-to-end framework for training models with Knowledge Distillation and running reproducible experiments.
- An implementation of the KD losses introduced in Section 2.3.1.
- Fine-tuning $\text{BERT}_{\text{BASE}}$ on SST-2, to use as a teacher model.
- Experiments running the distillation procedure on SST-2 and evaluating the performance of the distilled models.

[1] https://gluebenchmark.com/faq

Upon meeting these criteria, I work on the following extensions:

- Evaluate the approach on more datasets in GLUE.
- Implement and evaluate knowledge distillation with quantisation.
- Experimenting with different approaches for layer mapping.

Component	Priority	Difficulty	Risk	Reason
Training procedures	High	Medium	Medium	Many moving parts, requirement for multi-GPU support
BERT pre-training data generation	Medium	High	High	Tricky data manipulation and high data volume, needs to run fast
KD-losses	High	Medium	Low	Tricky tensor manipulations
Data augmentation	High	Medium	Low	Tricky tensor manipulations
GLUE loading	Low	Low	Low	
Quantisation	Low	Medium	Medium	Modifying existing model and custom gradient calculations

Table 2.2: Risk assessment of the required components

Components In order to meet the success criteria and extensions, the following components need to be implemented:

- A parallel **pre-training data processing pipeline** for preparing the raw text corpus to be used for the BERT Pre-training procedure. This includes tokenisation the text, sampling pairs of sentences for next sentence prediction task and random masking.
- The **KD-losses**.
- Functions to **load and prepare the GLUE datasets** for training (tokenisation, inserting control tokens), supporting both single sentence classification and sentence pair classification.
- **Dataset augmentation** procedure for the task specific distillation, with support for parallelism.
- **BERT Pre-training and Fine-tuning procedures**, both with support for checkpointing, utilising multiple GPUs, continuously logging loss and accuracy.
- Implementation of **quantisation**, including implementation of PyTorch Embedding and Linear layers with support for ternary quantisation and fixed point quantisation, and an implementation of self attention with support for quantisation.

In Table 2.2 I perform a risk assessment of these components.

2.7 Software Engineering Methodology

As the requirements from Section 2.6 are clear from the beginning of the project, I approach the core of the project with a waterfall model:

$$\textit{Requirements analysis} \longrightarrow \textit{Design} \longrightarrow \textit{Implementation} \longrightarrow \textit{Testing} \longrightarrow \textit{Evaluation}$$

In the design phase, I flesh out the requirements with a detailed description of what the main functions and classes I need to implement are, producing a structure similar to Figure 3.1. The implementation and testing steps are interwoven when possible, as I write unit tests before implementing a module where applicable to pursue test-driven development, and where applicable I manually test a module directly after implementing it. Due to the nature of the project, the last step is evaluation instead of the typical deployment and maintenance.

When the core is finished, I repeat this model for the quantisation extension.

2.8 Testing

Testing challenges It is in general difficult to verify the correctness of a machine learning project. In contrast to classical software engineering, we do not have any clear criteria for correctness and it is difficult to predict beforehand what the performance of a correctly implement system should be. Furthermore, a poor performance is not necessarily caused by a buggy implementation, but could be caused by poorly chosen architecture or hyperparameters. On the other hand, a well performing model doesn't guarantee that the implementation is correct.

In light of this, I use several testing approaches to ensure that my ML pipeline, models and algorithms are correctly implemented:

Unit testing I verify the correctness of some critical components using unit testing. This includes the implementation of quantisation, KD-losses, data augmentation and the BERT pre-training dataset generation.

Comparison against baselines I fine-tune the same models on the same dataset, without KD, and show that the introduction of the KD technique significantly improves accuracy.

Reproduction of prior results A large portion of my project is the implementation of two papers, TinyBERT and TernaryBERT. My results mirror those given from running their open source implementations, indicating correctness of my implementation.

2.9 Starting Point

I use a pre-trained BERT base, and the tokeniser it comes with. These are both implemented in the HuggingFace Python library [44]. Other than this, everything is implemented from scratch.

Dependency	License
NumPy Pandas PyTorch	3-clause BSD
HuggingFace Transformers HuggingFace Datasets BERT Pre-trained models	Apache 2.0
Python matplotlib	PSFL
GLUE Dataset	CC By 3.0
Wikipedia Dataset	CC By 4.0

Table 2.3: Licences of dependencies

2.10 Licensing

All software dependencies of my project use permissive licences and allow me to use their code without restriction. The pre-trained models from the BERT paper [10] are released under the same license as their open source code, Apache 2.0, and the datasets I'm using are licensed with CC By 3.0 and 4.0 respectively. As I am not redistributing any of these, my project can make use of them freely.

To enable other developers and researchers to use my work, I license it under the MIT licence, which complies with the licenses of my dependencies.

3 Implementation

3.1 Overview

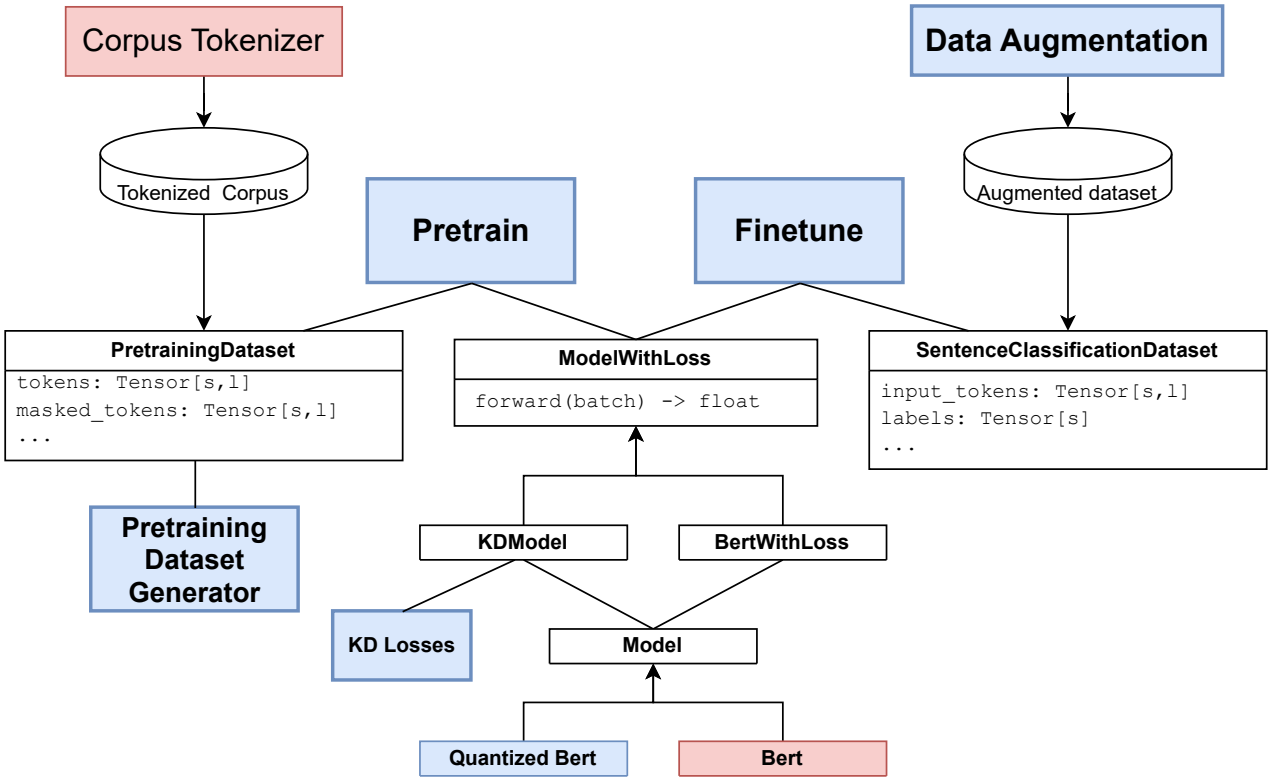


Figure 3.1: Overview of my implementation. Blue cells are main components, requiring 200+ lines of code. Red components are the only parts not implemented by me (I use the HuggingFace's [44] implementations). Upwards arrows indicate inheritance. Lines indicate the higher component using the lower one. Cylinders are file structures stored on disk.

To run experiments testing my KD implementation, a pipeline for training and evaluating language models is needed. Figure 3.1 show an overview of the important components.

Desired characteristics In addition to common desired characteristics such as readability and extensionability, three important characteristics the pipeline should have are:

- *Scalability:* The datasets I work with are on the order of 20GB for pre-training and 1GB for the largest finetuning tasks. Thus, care need to be taking during data preparation and training to effectively utilise the available compute and memory resources. During a single pre-training run, the model will see tens of billions of tokens, taking hundreds of GPU hours. To minimise this time, the pipeline should scale to utilise all the GPUs available in a compute node effectively. I also want to be able to resume a training run if it for some reason terminates early.

Num. GPUs	1	3	3
Approach		`DataParallel`	`torch.distributed`
Training time (s)	**1726** ± 21	**1241** ± 7	**637** ± 6
Speedup factor	1×	1.39 ×	2.71×

Table 3.1: Time taken to run $\frac{1}{64}$-th epoch of pre-training. We use `torch.distributed` and get a speedup factor close to the theoretical maximum of 3x, but a bit is lost in communication overhead

- *Reusability:* The BERT pre-training procedure is similar to the general distillation procedure, with the difference that in the later case two models are run and a distillation loss is calculated instead of a label loss. Similarly, the BERT finetuning procedure and the task-specific distillation share most functionality requirements. Therefore, I want to implement the training procedures so that they can be used for training both with or without KD.
- *Reproducibility:* In order for others to verify my results, it is important that it is easy to re-run all experiments presented in Chapter 4 and get the same result.

In the following sections, I describe how these characteristics were achieved.

3.1.1 Scalability

Dataset pre-processing The dataset pre-processing is more optimized than previous implementations and supports parallellisation, and described in detail in Section 3.2.

Dataset loading For pre-training and general distillation, I save the dataset in smaller *dataset-chunks*. This enables us to load only a single part to memory at time, reducing overhead and enabling us to assign less RAM to the compute node I run the training job on.

Multi-GPU training PyTorch provides two different ways of achieving multi-GPU data parallelism. The `DataParallel` module is easier to use, but only supports a single process and has additional overhead in sending weights and gradients back an forth between CPU and GPU.

I instead use `torch.distributed` framework citepytorchDistributed, which provides multi-processed multi-GPU support. Each GPU has its own process, and we use available commands such as `torch.distributed.all_reduce` to synchronise gradient, loss values and prediction across processes. One process is designated for logging and checkpointing. Measurements showing the speedup can be found in Table 3.1.

Checkpointing Some training jobs might run for multiple days, meaning high risk for early termination due to e.g. HPC time limits or out of memory problems. We therefore regularly save checkpoints, containing model weights and state of optimiser and learning rate scheduler. When starting a training job, there is an option to resume from a checkpoint.

3.1.2 Reusability

To enable the use of my training loops both with or without KD, I simply move the calculation of loss from the training code to the model itself. I define an abstract classes `PretrainingModel` and `SequenceClassificationModel` whose forward methods return calculated loss alongside relevant predicted values. Now I can make two implementations of this class for pre-training and two implementations for finetuning:

- `BertForPretrainingWithLoss` and `BertForSequenceClassificationWithLoss`: Wrappers around a single BERT model, that calculates the relevant cross-entropy losses.
- `KDPretraining` and `KDSequenceClassification`: Classes that keep both a teacher and student model as internal state, and calculates relevant KD losses. See Section 3.3 for details.

3.1.3 Reproducibility

To ensure that others can independently verify my results I ensure full reproducibility of all experiments. This is done in two main ways:

- **Description of reproduction steps:** As different steps of my procedure has different requirements in terms of compute, i.e. CPU/GPU constrained calculations, it is not suitable to supply a single script that reproduces all experiments. Instead, I provide reproduction instructions in the `readme.md` file. Here, the full environment and exact order and arguments of commands that need to be run is specified.
- **Controlling non-determinism:** There are multiple sources of non-determinism in the pipeline, including: random masking for MLM, dataset shuffling, dropout, etc. To ensure the same pseudo-random values are generated across multiple runs, all scripts using randomness take `seed` as a command line argument. This single value is then used to initialised all global RNGs used as sources of randomness, including the RNGs of Python, NumPy, and PyTorch. The seed is set to 0 in all experiments if not otherwise specified.

I also ensure that I fulfill all the requirements for presenting experimental results from the NeurIPS 2021 Paper Checklist [**?**].

3.2 Dataset preparation

Procedures to generate training data need to be implemented for both the general distillation and the task-specific distillation step. As the datasets I am working with are large (20GB for pre-training/general distillation and 1GB after augmentation for task specific distillation), our approach to data preparation needs to be efficient.

Load dataset (s)	Generate Samples (s)	Run Training (s)
7.6 ± 0.5	21.6 ± 1.9	637.4 ± 6.1

Table 3.2: Time spent on loading the tokenised dataset, generate training samples (step 2 and 3 above), and running the training loop. Times are measured on a single dataset chunk on Sulis (see Section 4.2), taking the average over 5 repetitions.

3.2.1 BERT Pre-training Dataset Generation

Before we can run pre-training, we have to convert the text corpus into BERT pre-training samples. I split this process into three steps:

1. Split each document into sentences, and tokenise each sentence.
2. Combine tokenised sentences to create samples of desired number of tokens (As per Devlin et al. [18], I use 128). For the Next Sentence Classification task, 50% samples are single contiguous sequences of sentences, and 50% of samples are two segments from different documents.
3. Apply masking and random token replacement for the Masked Language Modelling task.

The output of this procedure is 5 tensors, so requiring more memory to store than the original dataset. Also, as step 2 and 3 are non-deterministic, they are ideally run one time for each epoch so the number of unique examples seen during training is maximised.

To combat difficulties with memory load and debugging iteration time arising from the large size of the dataset, I split it into 64 chunks before processing it. This is also useful for the training script, as it can load and train on a single chunk at a time instead of loading the entire dataset into memory.

In step 1, I use the HuggingFace library to load[1] and tokenise[2] the Wikipedia dataset. The tokeniser is well optimised and has support for parallelism, and takes about 10 minutes to run with 8 cores on a single chunk of the dataset. In total, this step takes around 10 hours and I run it once on Beyla (see Section 4.2), producing 64 `pickle` files of ~ 200 MB, for a total of 13 GB.

I implement step 2 and 3 from scratch, integrate them into the training pipeline, and run it every time a new dataset chunk is loaded. This way we never have to store the dataset in the memory inefficient pre-processed format, and the non-deterministic steps get run anew every epoch. However, this means the procedure needs to be efficient. I therefore implement support for parallelism, after which the dataset generation step only takes 20 seconds on average on 8 cores. A breakdown of time spent for training on a single chunk can be seen in Table 3.2.

The open source code from the TinyBERT paper [18] contains an implementation of this procedure. My implementation improves on it in the following ways:

[1] https://huggingface.co/datasets/wikipedia
[2] https://huggingface.co/bert-base-uncased

- The TinyBERT implementation combines the three steps to a single script, run before any training. The output is a 1GB JSON file for each chunk, or almost 200GB for 64 chunks over 3 epochs, 15x larger than the maximum storage required for my implementation (13 GB).

- Loading a single such JSON file takes 41.8s (± 4.7s, measured 5 times), longer than the time spent in my implementation for loading and generating the samples combined. This leads to a ~ 1.5% speedup of the training loop.

- My implementation is tested with unit tests. Testing is difficult due to the probabilistic nature of the procedure. I use a combination of very small inputs, where we can check that the output matches any of the possible correct outputs, and large inputs, where I verify statistical properties of the output.

3.2.2 Data Augmentation

The augmentation procedure (see Algorithm 1) requires multiple inferences of a pre-trained BERT model for each sample. It is therefore preferable to run the augmentation just once before training and save the result. The open source code from the TinyBERT [18] paper contains an implementation of this procedure. However, this takes prohibitively long to run for some of the larger datasets I use. By running the procedure for 1000 samples, I estimate it would take about 45 hours to run a full augmentation (20 augmented samples for each original training sample) of the MNLI [41] dataset.

In my own implementation, I do two things different to speed up the algorithm:

1. **Batching of BERT inputs**: For each of the single-token words in a sentence, we need to run one inference of a pre-trained BERT model. To more efficiently utilise parallelisation capabilities of the GPU, I collect multiple of these masked sentence inputs into a batch, and run inference on the whole batch at once.

2. **Multiprocessing**: As we are alternating between GloVe vector look-ups and BERT inferences, a single process doesn't fully utilise the GPU all the time. We might also have multiple GPUs available for running inference. Thus, by using the **torch.multiprocessing** to process different parts of the dataset in parallel, a significant speedup can be gained.

A comparison of speedups can be seen in Table 3.3.

Testing Unit testing is made difficult by the probabilistic nature of the algorithm. I first test functions with small inputs and arguments that limit parallelism. For larger inputs I test that statistical properties, such as average percentage of words replaced, hold within some error margin.

Implementation	TinyBERT	Mine	Mine	Mine
Num. processes	1	1	4	8
Num. GPUs	1	1	1	2
Time (s)	**413.3** ± 16.1	**300.0** ± 26.5	**163.3** ± 1.2	**111.7** ± 5.4

Table 3.3: Time taken to augment 1000 samples. Times are measured on Beyla (see Section 4.2), taking the average over 3 repetitions.

3.3 Knowledge Distillation

3.3.1 `KDLoss`

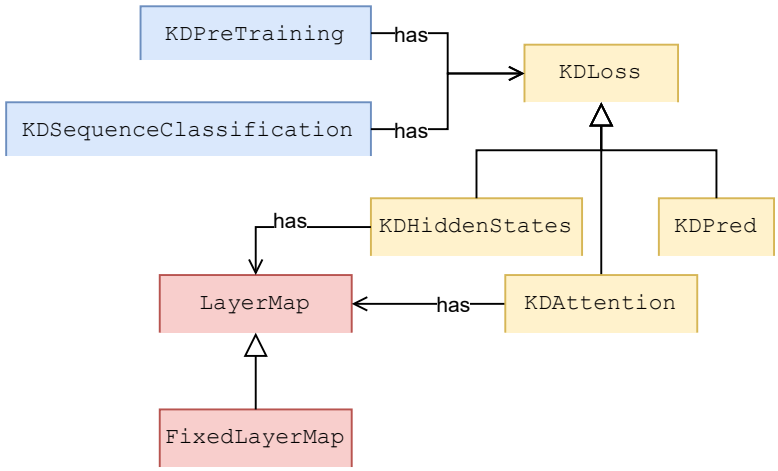


Figure 3.2: Structure of my KD implementation. Upward arrows indicate inheritance.

The main requirement of the knowledge distillation implementation is flexibility – experimentation and testing different losses should be easy and require minimal code changes. To this end, I define an abstract base class `KDLoss` representing a PyTorch module computing a component of the KD loss. A `KDLoss` class defines a `forward` method, taking two `ModelOutput` objects as input, one for the teacher and one for the student, and returns the loss. The `ModelOutput` objects contains hidden states, attention matrices and logits from a batch of input to the model. `KDLoss` extends the PyTorch `nn.Module` class as it may contain learnable parameters.

I implement four `KDLoss` classes:

- `KDPred`: Prediction layer distillation – computes the soft cross entropy with student logits as prediction and teacher logits as target.
- `KDAttention`: Attention based distillation – computes the MSE loss between student attention matrix and teacher attention matrix. It takes a `LayerMap` object that specifies how the layers of the student should be mapped onto the teacher layers.
- `KDHiddenStates`: Hidden states based distillation – computes the MSE loss between the hidden states of teacher and student. To compare the hidden states in case of differing

hidden dimension size, the class keeps a learnable FF layer, used to transform the student's hidden vector to the size of the teacher's. It also take a `LayerMap` object to map the teacher's layers onto the student's.

- `KDTransformerLayers`: Combines `KDAttention` and `KDHiddenStates`, by initialising an instance of each with sensible default layer maps and returning their sum.

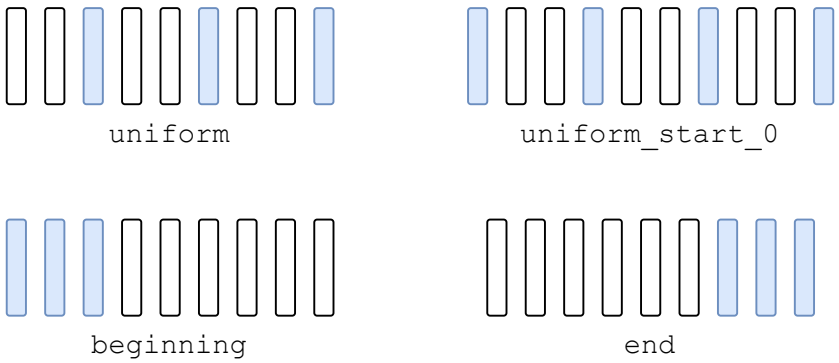


Figure 3.3: The four constant layer maps implemented by default, shown for a 9-layer teacher and 3-layer student. The blue columns indicate which teacher layers the student's layers are mapped to. `uniform_start_0` is used by default for hidden states based distillation, and thus the example has an extra column to account for the embedding layer

3.3.2 `LayerMap`

The `LayerMap` module takes a tensor containing some per-layer teacher data (e.g. a attention matrices or hidden states) and a student layer index. It then returns a single data item to be compared to the student layer of specified index. Initially I only implement a single layer map: `FixedLayerMap`. In Section 4.7 I implement and evaluate a more general learnable layer map.

`FixedLayerMap` accepts four different options for how to construct the mapping: `uniform`, `uniform_start_0`,`beginning`, and `end`. See Figure 3.3 for an explanation of each mapping. The first two are used by default for `KDAttention` and `KDHiddenStates` respectively, while the latter two exist mainly for ablation purposes.

3.4 Quantisation

I implement each quantisation method as a `torch.autograd.Function`. This is an interface that let's you define a custom gradient calculation. It consisting of two static methods, `forward` and `backward`. The `forward` method takes in an autograd context and a number of tensors as input, does any calculation, and returns a resulting set of tensors. It also contains a backward method which takes in gradients w.r.t the output tensors of the forward method, and calculates the gradient w.r.t the input tensors of the forward method. By simply propagating the gradient through my backward methods without modification, PyTorch will calculate the gradient w.r.t

the full precision weight by equating them to the gradients w.r.t the quantised weights. When calling `optimizer.step()` we then get our desired behaviour of updating the full-precision weights using the gradient w.r.t the quantised weights. This is known as *straight-through estimation* of gradients [9].

In order to use the quantisers as objects that can store fixed parameters (such as clipping value, whether quantisation is layer- or row-wise), I also implement thin `Quantizer` classes that extend `nn.Module` and whose forward method simply calls the corresponding `torch.autograd.Function`.

Clipping As mentioned in 2.4.1, clipping model weights before quantising is helpful for stability. However, this means that the gradient w.r.t the full precision weight should be zero when the weight is clipped. Therefore, in the forward method we save a boolean mask indicating what values were clipped in the autograd context using `ctx.save_for_backward`. In the backward function this mask can be retrieved using `ctx.saved_tensors`. As clipping is done in multiple quantisation functions, I implement reusable functions `clip_and_save` and `gradient_apply_clipping`, that are called from the relevant forward and backward methods.

Quantisation across dimensions For a certain layer, we can either choose to learn a single set of quantisation parameters (e.g. α in the ternarisation quantiser) for the entire layer (*layer-wise quantisation*), or for example learn a separate parameter for each row (*row-wise quantisation*)or for each column to increase expressivity in trade for complexity and number of parameters. To allow full flexibility, I let all my quantisation functions take in a weight tensor of arbitrary shape. In addition, a tuple of dimension indices to perform the quantisation across is passed in.

3.4.1 Quantised BERT

In my quantised BERT, weight quantisation will be applied to all linear layers, the embedding layer, and all matrices used for matrix multiplication. We also quantise the inputs to all linear layers and matrix multiplication, which can speed up the matrix multiplication operation [25] when running on dedicated hardware.

Thus, in order to implement a quantised version of the BERT model we need to modify three parts of the model:

- All linear layers
- The embedding layer
- The matrix multiplications in the attention layer

To save us from unnecessarily re-implementing the entire BERT architecture, I opt to start with a standard HuggingFace BERT model, and replace some layers with my own implementation. Specifically, I replace all `Linear` and `Embedding` Layers with layers `QuantizedLinear` and

`QuantizedEmbedding` that I implement. I also implement a `CustomBertSelfAttention` where I add support for quantisation. A function `prepare_bert_for_quantization` takes in a BERT model, and traverses all layers replacing modules with their quantised counterparts.

`QuantizedLinear` and `QuantizedEmbedding` These two layers both replicate the functionality of the corresponding PyTorch layers. The constructors takes a non-quantised layer used for initialisation, a weight quantiser and an activation quantiser as input. In the forward method, weights and activations are quantised using the supplied quantisers, before the layer output is calculated (using `nn.functional.linear` and `nn.functional.embedding`).

`CustomBertSelfAttention` This module is a simple re-implementation of the BERT self attention layer, with the added functionality of using `QuantizedLinear` layers in place of the `Linear` layers W_Q,W_K and W_V. It also takes an activation quantiser, to use for quantising the Q and K matrices before multiplication, and the V matrix before multiplying it with the attention matrix.

The `CustomBertSelfAttention` is also necessary to use for non quantised KD, as instead of attention probabilities it returns the pre-softmax attention scores, which is what we use in attention based distillation.

3.5 Repository Overview

My project repository has the following layout:

- `/`
  - `datasets` ..... stores the Wikipedia and GLUE datasets
  - `output` ..... experiment logs, result files and model checkpoints.
  - `evaluation`
    - `plot_and_explore.ipynb` ..... plots and explores evaluation results.
  - `scripts` ..... scripts for launching batched experiments, fetching datasets, etc.
  - `src`
    - `modelling`
      - `models.py` ..... Model creation and loading utilities **241 loc**
      - `quantisation.py` ..... implements quantisation of BERT **254 loc**
      - `checkpoint_to_model.py` ..... creates a model file from a checkpoint **29 loc**
    - `tests`
      - `test_kd.py` ..... **56 loc**
      - `test_quantization.py` ..... **67 loc**
      - `test_prepare_pretraining_dataset.py` ..... **134 loc**
      - `test_dataset_augmentation.py` ..... **98 loc**
    - `args.py` ..... Shared command line arguments classes **41 loc**
    - `dataset_augmentation.py` ..... implements the dataset augmentation **254 loc**
    - `kd.py` ..... implements the KD loss functions **295 loc**
    - `load_glue.py` ... implements loading of GLUE datasets to common format **321 loc**
    - `prepare_pretraining_dataset.py` .... BERT pre-training data generation **211 loc**
    - `pretrain.py` ..... implements the pre-training training procedure **255 loc**
    - `kd_pretrain.py` ..... runs pre-training with KD **69 loc**
    - `finetune.py` ..... implements the finetuning training procedure **291 loc**
    - `kd_finetune.py` ..... runs finetuning with KD **100 loc**
    - `utils.py` ..... shared functions for e.g. logging and randomness control **105 loc**
  - `LICENSE` ..... MIT License
  - `README.md` ..... Usage and reproduction instructions

4 Evaluation

4.1 Success Criteria

Success criteria All success criteria from the project proposal (Appendix C) have been met:

- **An implementation of KD framework:** described in Section 3.1.
- **Implementation of KD losses**: described in Section 3.3.
- **Teacher fine-tuning with at least 90% accuracy on SST-2**: Described in Section 4.3.1
- **Running KD experiments**: Described in Section 4.4
- **Report accuracy and FLOPs**: Accuracy is reported for all datasets in Table 4.3, and FLOPs are reported in Table 4.5.

Extensions The following extensions have all also been implemented:

- **Evaluate on more datasets in GLUE**: For most experiments, including in Table 4.3, I report results on 7 GLUE datasets.
- **Implementation and evaluation of KD with quantisation**: Section 3.4 describes the implementation of quantisation, and experiments are performed in Section 4.5.
- **Experimenting with different layer map techniques**: Described in Section 4.7.

4.2 Experimental Setup

System specifications I use a workstation in the Computer Lab for testing during development, running data pre-processing, and for less resource intensive training runs such as the teacher fine-tuning. This workstation is referred to as *Beyla*, and has 4 Intel Xeon E5-2620 v4 CPUs @ 2.10GHz with 8 cores each, 4 NVIDIA GeForce GTX 1080 Ti GPUs and 132 Gigabyte of RAM.

For more resource intensive experiments, I got access to *Sulis*, a tier 2 HPC platform at Warwick University. It has 80 GPU nodes, each equipped with 3 NVIDIA A100 graphics cards, each with 40GB of VRAM. General distillation took about 300 GPU hours in total, and running task-specific distillation on all datasets takes around 100 GPU hours, 90 of which is spent on the larger datasets QQP, MNLI and QNLI.

Source	CoLA	MRPC	RTE	SST-2	MNLI	QNLI	QQP
Zhang et al. [51]	**58.1**	**86.5**	**71.1**	**93.1**	**84.5**	**92.0**	**87.3**
Mine	**59.4** (57.1±1.8)	**84.3** (84.1±0.2)	**71.1** (68.2±2.4)	**93.4** (93.0±0.3)	**84.7**	**91.2**	**87.3**

Table 4.1: Results of $\text{BERT}_{\text{base}}$ finetuning on the dev-splits of GLUE. My results roughly match Zhang et al. [51], who also publish their dev-split results of teacher finetuning. In parenthesis, I also report mean and standard deviation of rerunning with the best performing hyper-parameters 3 times.

Evaluation Metrics In line with previous literature I employ Accuracy (Acc), F1 score (F1) and Matthew coefficient (MTT) [28] (a binary variable correlation measure, ranging from -1 to 1), to evaluate the performance of trained models on the GLUE dataset.

Score is measured using Matthews Coefficient for the CoLA dataset, F1 score is used for QQP, and for all other datasets I use accuracy. When calculating average scores, MTT is first scaled to range from 0 to 1 instead of -1 to 1.

Optimiser For all model training I use the AdamW [19**?**] optimiser with β_1 = 0.9, β_2 = 0.999 and L2 weight decay of 0.01. I also use dropout with a probability of 0.1. These are all standard values for both pre-training and finetuning BERT [10], and the same values is used for Knowledge Distillation of BERT [18]. Batch size and learning rate are the main hyper-parameters that I vary across experiments.

4.3 Experiments

4.3.1 Teacher Training

Training setup To perform knowledge distillation, I first have to train teacher models for all tasks I intend to evaluate on, i.e. the seven tasks from GLUE specified in Section 2.5. I use $\text{BERT}_{\text{base}}$ as a teacher model for all experiments. It has 12 layers, 12 attention heads, and hidden dimension size 768. A pre-trained $\text{BERT}_{\text{base}}$ is downloaded using the HuggingFace library [44]. For every dataset in GLUE, we add a randomly initialised classifiation layer to the pre-trained model, and train for 5 epochs. After every epoch, we evaluate the model on the dev-split and save metrics to a log file.

Hyper-parameter optimisation To find suitable hyper-parameters I do a grid search with the following values, that were reported to work well across tasks by the BERT paper [10]: learning rate chosen from $\{1 \times 10^{-5}, 2 \times 10^{-5}, 5 \times 10^{-5}\}$, batch size chosen from $\{16, 32\}$. For each combination of hyper-parameters, we run the training for 5 epochs, and record the best evaluation metric that has been observed. Results can be seen in Table 4.1.

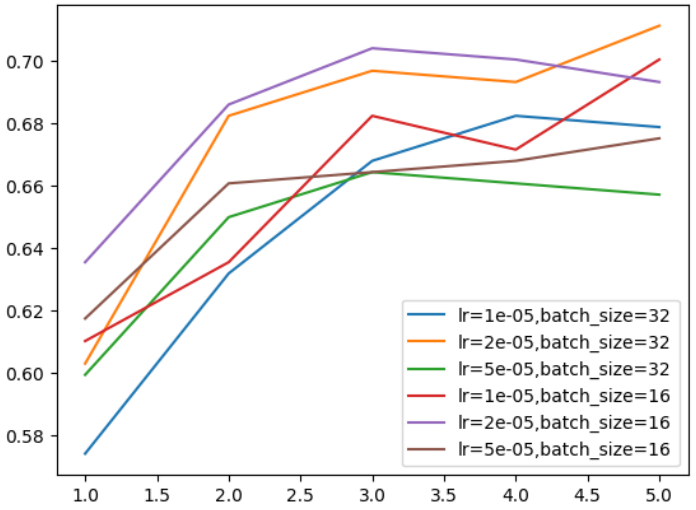


Figure 4.1: Hyper-parameter search of finetuning $\text{BERT}_{\text{base}}$, all with `seed=0`

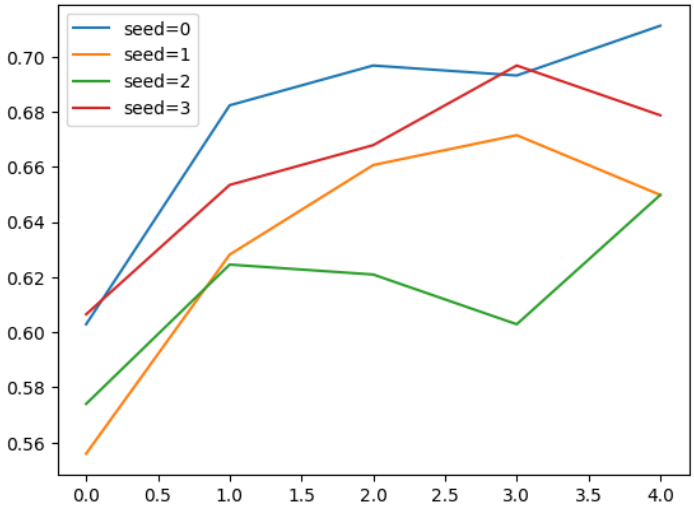


Figure 4.2: Four runs of fine-tuning of $\text{BERT}_{\text{base}}$ with `lr=2e-5,batch_size=32`

Figure 4.1 shows that the performance varies ~ 4% with choice of hyper parameters. However, as Figure 4.1 shows, random seed (affecting initialisation, random batching and dropout) also affects the score with similar variance. It is therefore hard to tell how much of the performance gain from the hyper-parameter search comes from finding more suitable parameters, and how much is simply noise from changing any parameter. This is why the 3x mean is significantly lower than the max score in Table 4.1.

In order verify that the scope of the hyper-parameter search is sufficient, I run a Bayesian hyper-parameter search with a larger number of samples, which did not produce an improvement in score. We also note that this range of learning rates is common in finetuning transformers [10, 34], and that AdamW is less sensitive to a correct choice of learning rate than other optimisers as it is a second order optimiser that learns a per-parameter step size. With this in mind, I say that the performed hyper-parameter search is sufficient.

Overfitting The noisiness of the hyper-parameter search also means instead of finding hyper-paraters that lead to good generalised performance, we might be implicitly overfitting on the development dataset. Therefore, the reported max scores might not accurately reflect the capabilities of my trained models. As we in this project only compare systems against each other we are not too concerned about generalisation ability. However, this shows that dev-set performance of a system can be increased by introducing more variance, e.g. searching over more hyper-parameters, and by increasing the number of times evaluation occurs e.g. running for more epochs. Reporting mean and standard deviation of multiple runs help rule out this factor when comparing systems.

Core takeaways:: The range of hyper-parameters considered is suitable, and within this range the effect on performance is similar to the generally high variance of fine-tuning.

4.4 Knowledge Distillation

Due to the expensive general distillation step, I only run Knowledge Distillation on a single BERT architecture with parameters $l = 4$ (number of layers), $d = 384$ (hidden vector size), $a = 12$ (num. attention heads). I denote this instance of BERT as BERT_{KD4} [1].

4.4.1 General Distillation

Training setup I run general distillation with BERT_{KD4} as student and a pre-trained $\text{BERT}_{\text{base}}$ as teacher on the Wikipedia dataset. The training is initially run for 3 epochs, with a batch size of 256 and learning rate linearly decaying from 1×10^{-4} to 0. These are the same parameters used in the TinyBERT paper. Towards the end of training, the loss was still decreasing. As the large dataset (~20GB) compared to the size of the model (~400MB) means that overfitting on the pre-training dataset is unlikely, I decided to continue training for more epochs. I first trained for 3 additional epochs resuming from the latest checkpoint, and then for 12 additional epochs. In each restart, the learning rate schedule reset, starting from 1×10^{-4} again. This explains the initial increase in loss for the first epoch after the first restart.

Results To demonstrate the effect of general distillation, I compare the student model after 3 epochs, 6 epochs and 18 epochs by running task-specific distillation on RTE and CoLA. I also compare against the *General* $BERT_{KD4}$ published together with the TinyBERT paper.

Core takeaways: Running the general distillation step is cruicial for performance, and running it for longer increases downstream performance.

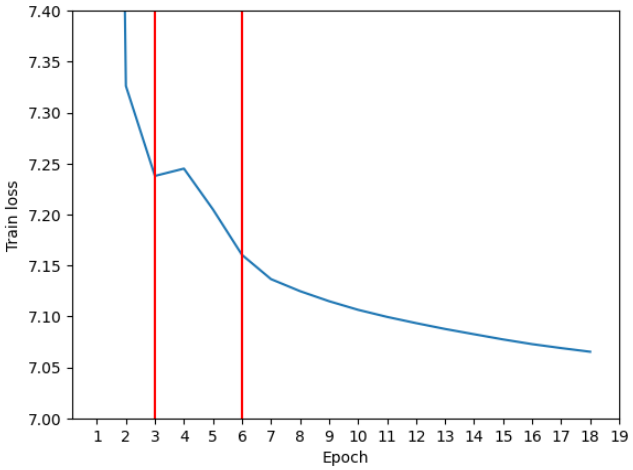


Figure 4.3: Average training loss per epoch during General Distillation. The red lines mark the points where training was resumed.

4.4.2 Task-specific Distillation

Training setup Before running task-specific KD, I pre-compute augmented datasets for all GLUE tasks by running Algorithm 1. As per Jiao et al. [18], I use the parameters $N = 20, K = 15, p_r = 0.4$.

For the Task-specific Distillation, I use the best fine-tuned $\text{BERT}_{\text{BASE}}$ on the corresponding dataset as teacher model. We use the generally distilled BERT_{KD4} produced from Section 4.4.1

[1] This model is called TinyBERT_4 in the TinyBERT paper [18], but I choose a different name to avoid confusion with $\text{BERT}_{\text{TINY}}$, a different model

Dataset	0 Epochs	3 Epochs	6 Epochs	18 epochs	TinyBERT Paper
RTE	**59.9** (57.3±1.2)	**65.3** (62.2±1.3)	**65.7** (63.8±0.9)	**66.1** (63.8±1.6)	**65.3** (63.8±1.1)
CoLA	**33.1** (31.0±1.7)	**37.7** (36.3±1.0)	**40.5** (36.6±2.0)	**41.1** (37.7±1.9)	**40.4** (38.3±1.5)

Table 4.2: Comparison of different number of general distillation epochs used for student initialisations for task-specific distillation on RTE and CoLA. In parenthesis: mean and standard deviation across all evaluations (15 per run).

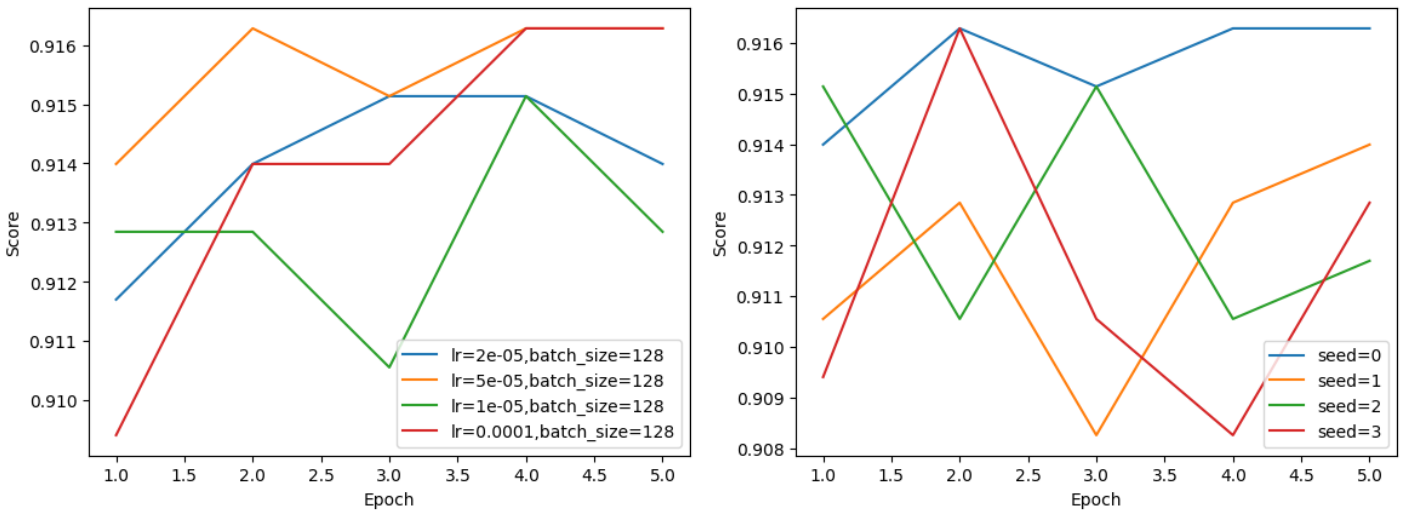
Model	#Params	CoLA	MRPC	RTE	SST-2	MNLI	QNLI	QQP
$\text{BERT}_{\text{BASE}}$	109.5M	**59.4** (57.1±1.8)	**84.3** (84.1±0.2)	**71.1** (68.2±2.4)	**93.4** (93.0±0.3)	**84.7**	**91.2**	**87.3**
$\text{BERT}_{\text{MEDIUM}}$	41.4M	**47.6** (44.5±1.8)	**81.4** (81.0±0.4)	**65.3** (63.3±1.9)	**90.0** (89.6±0.3)	**80.4**	**88.7**	**86.7**
$\text{BERT}_{\text{SMALL}}$	28.8M	**38.2** (37.5±0.7)	**78.4** (77.8±0.5)	**64.6** (64.1±0.5)	**88.5** (87.7±0.5)	**78.4**	**87.2**	**85.9**
BERT_{KD4} (Mine)	14.4M	**42.2** (41.6±0.5)	**82.0** (80.4±1.0)	**67.1** (65.3±1.7)	**91.6** (91.5±0.1)	**81.3**	**88.9**	**84.7**
BERT_{KD4} (Paper)	14.4M	50.8	85.8	-	-	82.8	-	-
BERT_{KD4} (GitHub)	14.4M	42.0	82.0	66.0	92.1	-	-	-

Table 4.3: Results from task specific finetuning. BERT_{KD4} refers to the model trained with KD. *Paper* refers to the values reported by the TinyBERT paper [18]. *Github* refers to the values I got from running their open sourced implementation. The TinyBERT paper doesn't report dev-split scores for RTE, SST-2, QNLI and QQP, and I only run their published code for the four smallest datasets.

as student. Jiao et al. [18], performs the distillation in two steps. First transformer layer distillation is run for 20 epochs with learning rate $\eta = 5\times10^{-5}$, and then prediction layer distillation is run with a grid search for learning rate using values $\{1\times10^{-5}, 2\times10^{-5}, 5\times10^{-5}\}$. Both steps use a batch size of 32. In order to speed up my experiments, and to properly utilise the 40GB VRAM A100s that I train on, I increase batch size to 128. This means fewer gradient updates take place in an epoch, so to retain the same effective learning rate should be increase. Krizhevsky [21] shows that learning rate should scale as the square root of batch size, and thus we double all learning rates.

As with the finetuning, I report a mean and confidence interval from rerunning with the best hyper-parameters. Due to compute constraints, I only rerun the prediction layer distillation. All results are presented in Table 4.3. Figure 4.4 shows loss and accuracy from task specific distillation on SST-2.

Comparison against reference implementation To verify the correctness of my implementation, I compare against the values published by TinyBERT [18]. They only publish dev-



(a) Hyperparameter search

(b) Repeats of the best run

Figure 4.4: Task specific distillation of BERT_{KD4} on SST-2

split results for CoLA, MRPC and MNLI. On all of these, my implementation perform significantly worse. To find the source of the discrepency, I try to reproduce their results by running their open sourced implementation. As they publish their model after general distillation, we only have to run the task-specific distillation. They do not publish exact instructions for reproducing results, so I make my best guesses based on the paper and default values in the scripts. We find that the results are significantly lower than the published results, and roughly match mine. A thorough discussion of the potential causes for this can be found in Appendix A. As my results roughly match the values we get from their implementation, it indicates correctness of my implementation.

Comparison against baselines We find that the distilled model significantly outperforms the $\text{BERT}_{\text{SMALL}}$ model on most datasets, differing by multiple standard deviation.

This shows that in most cases, the benefit from KD is big enough to outperform much bigger models. The reasons for this are the ones already outlined in Section 2.3: knowledge transfer from the teacher model through the embedding, transformer layer and prediction layer distillation, and the increased amount of training data from data augmentation. The TinyBERT paper [18] performs ablation studies, showing that removing intermediate layer leads to an average decrease in score of around 10 points, while removing data augmentation reduces score by around 2-3 points.

The one exception is QQP, where $\text{BERT}_{\text{SMALL}}$ outperforms the distilled model. Other than the fact that $\text{BERT}_{\text{SMALL}}$ is a larger model, a possible explanations for the worse score is that the embedding layer and intermediate layer distillation actually is detrimental to performance. It might be that, for this specific dataset, what makes for useful intermediate representations in a larger model are not similar enough to what makes for useful representations in a smaller model. Alternatively, the rigidity of the fixed layer map used might hinder performance by forcing each student layer to learn from a specific teacher layer. Introducing higher flexibility in

Model	Size	CoLA	MRPC	RTE	SST-2	MNLI	QNLI	QQP
$\text{BERT}_{\text{BASE}}$	437.9 MB	**59.4** (57.1±1.8)	**84.3** (84.1±0.2)	**71.1** (68.2±2.4)	**93.4** (93.0±0.3)	**84.7**	**91.2**	**87.3**
Distilled BERT_{KD4}	57.4 MB	**42.2** (41.6±0.5)	**82.0** (80.4±1.0)	**67.1** (65.3±1.7)	**91.6** (91.5±0.1)	**81.3**	**88.9**	**84.7**
TernaryBERT (mine)	30.3 MB	**56.4** (55.9±0.4)	**84.8** (84.6±0.2)	**70.8** (70.6±0.3)	**93.3** (93.3±0.1)	**83.5**	**90.5**	**86.7**
TernaryBERT (paper)	30.3 MB	**55.7**	**87.5**	**72.9**	**92.8**	**83.3**	**89.9**	**87.2**
TernaryBERT (PTQ)	30.3 MB	**0**	**67.1**	**47.2**	**74.3**	-	-	-
TernaryBERT (No KD)	30.3 MB	**0.3**	**68.3**	**54.2**	**78.0**	-	-	-

Table 4.4: Results from training TernaryBERT with KD. PTQ refers to using ternarisation as Post Training Quantisation, i.e. not doing any training. No KD refers to running the standard BERT fine-tuning procedure without knowledge distillation

the layer map could improve the result, which I investigate in Section 4.7. The already large size of the QQP dataset also means that the impact of data augmentation will not be as significant.

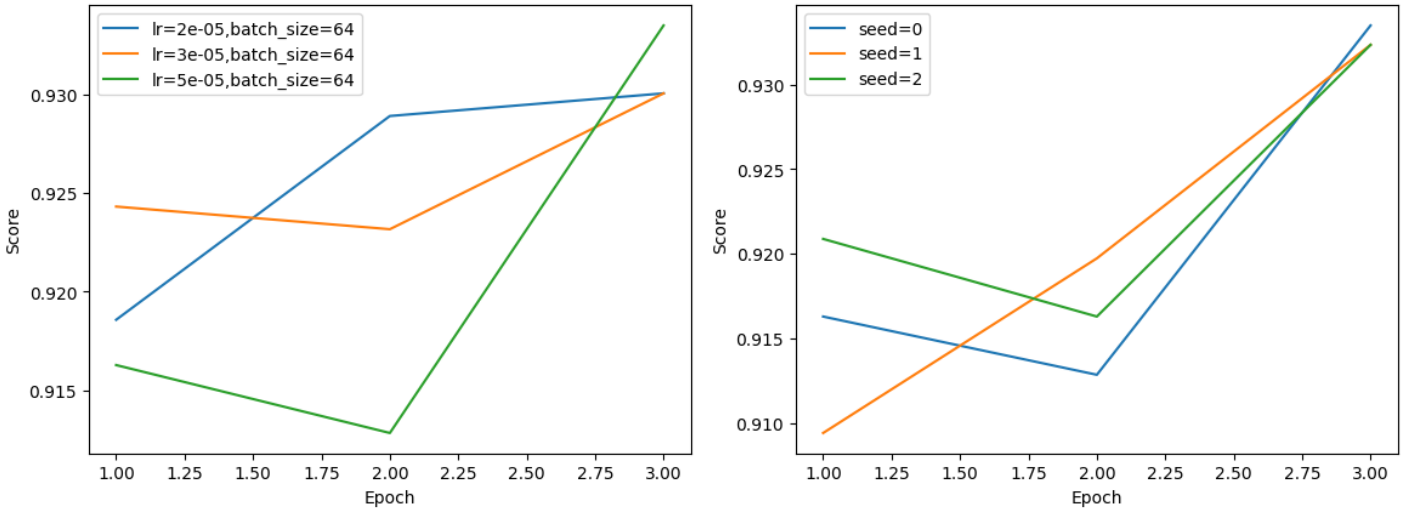
Core takeaways: Our results match the reference implementation and produces results in line with a 3x larger model on most datasets. However, the student performs worse on some dataset, and I hypotheise that rigidity in the layer map might hinder performance.

4.5 Quantisation

Training setup I also evaluate my implementation of quantisation by training TernaryBERT, a ternarised version of $\text{BERT}_{\text{BASE}}$. This model ternarises all dense weights, i.e. all matrices used for matrix multiplication, including in the embedding layer, linear layers and attention layers. Following the TernaryBERT paper, row-wise ternarisation is employed for the embedding layer, as the performance otherwise is significantly reduced. For all other layers, layer-wise ternarisation is used. We also quantise all activations, using 8 bit MinMax quantisation. This leads to a memory reduction of 93%.

In contrast to the default KD setup, the student model can now be initialised directly by quantising the teacher model. As the starting point is much closer to the teacher model, less epochs are required for training. I therefore train for 3 epochs on the augmented dataset for the smaller datasets CoLA, RTE, MRPC and SST-2, and only a single epoch on the larger datasets. The TernaryBERT paper suggest a linear learning rate decay from 2×10^{-5} and a batch size of 32. Similarly to my non quantised KD experiments, I opt to use a higher batch size of 64 and thus raise the learning rate to 3×10^{-5}. I run a hyper-parameter search on the smaller datasets CoLA, MRPC, RTE and SST-2, with learning rate chosen from $\{2 \times 10^{-5}, 3 \times 10^{-5}, 5 \times 10^{-5}\}$, and the best run is repeated 3 times with different seeds. Results can be seen in Table 4.4.

Discussion TernaryBERT outperforms BERT_{KD4} on all tasks, matching the score of the teacher model on most tasks and even exceeding it slightly for MRPC, all while using only 7% as much memory to store the model. This shows that KD with quantisation is a very effective



(a) Hyper-parameter search

(b) Repeats of the best run

Figure 4.5: Task specific distillation of TernaryBERT on SST-2

way of reducing the memory requirements of a model, and outperforms normal KD using less memory. I propose two main reasons for this:

1. Using 32 bits to represent 16 different parameters might be a more expressive than using 32 bits for a single parameter. Even though the total number of models representable with each approach is the same (given same total number of bits), the quantised version is able to express models spanning a wider range. (This is very handwavy, should I clarify my thoughts/find references or remove or say something simpler?)

2. As the architecture of the quantised model more closely matches that of the teacher, the restrictions imposed by the KD are more likely to be a useful ones. For instance, as the number of layers are the same the layer mapping becomes trivial.

In Table 4.4, I also report results from TernaryBERT before any training, and from training with a normal finetuning procedure instead of KD. The low scores show that the KD is crucial for the effectiveness of the quantisation. The most likely explanation is that the straight-through gradient approximation (see Section 3.4).

In Figure 4.5 I show the score over all training runs of SST-2. We notice that the final accuracy has relatively low variance, both across hyper-parameters and seeds. This is likely due to the linear decay in learning rate, meaning that regardless of initial learning rate, towards the end of the training the learning rate always approaches zero. The variance across seeds is also lower as the classification layer is not randomly initialised, but copied from the teacher, removing one source of non-determinism.

Core takeaways: Combining quantisation with KD gives a 14x model memory reduction with negligible performance loss, an improvement compared to the 3x reduction from standard KD, while taking much fewer epochs to train. We also note that KD is cruicial to high performance of the quantised model.

Model	#Layers	#Hidden Dimensions	#Params	Size	#FLOPs	Latency (ms)	Mean score
$\text{BERT}_{\text{BASE}}$	12	768	109.5M	437.9 MB	358.0B	178.1 ± 4.7 ms	84.5
$\text{BERT}_{\text{MEDIUM}}$	8	512	41.4M	165.5MB	107.6B	53.6 ± 0.9 ms	80.9
$\text{BERT}_{\text{SMALL}}$	4	512	28.8M	115.1 MB	53.8B	27.2 ± 0.9 ms	78.9
BERT_{KD4}	4	384	14.4M	57.4 MB	20.0B	16.9 ± 0.9 ms	81.0 (KD)
TernaryBERT	12	768	109.5M	30.3 MB	-	-	83.9 (KD)

Table 4.5: Memory and Compute requirements.

4.6 Memory and Compute

To compare the resource requirements of models, I calculate the following:

- The number of parameters in the model.
- The memory needed to store all weights.
- The number of floating point operations (FLOPs) required for inference of a single input batch[1]. This is a good proxy for inference latency, as FLOPs make up all calculation within the model after the embedding layer.
- Inference latency, i.e. time taken to process a single batch. I measure this by running inference 50 times on a single NVIDIA GeForce GTX 1080 Ti.

Results are shown in Table 4.5. BERT_{KD4} over 3 times as fast and takes up 3x less memory than $\text{BERT}_{\text{MEDIUM}}$, while having comparable performance.

The decrease in memory usage is especially significant for two reasons. Firstly, on modern GPUs, a significant portion of inference latency is due to data movement [**?**]. Secondly, it enables a reduction of RAM size in custom inference chips. RAM is a significant factor in multiple aspects of chip design, and affects power consumption, memory access latency and silicon area [**?**].

The quantised model achieves comparable performance to the teacher model, while only using 7% of the memory. The reduced memory is already a benefit, but an improvement in computational latency is less clear. In my implementation there is no improvement, as it is optimised for training. In practice, a low level implementation of the quantised model could be significantly faster if the hardware supports the necessary operations. For ternarisation, all expensive matrix multiplications (making up 99.9% of all FLOPs in $\text{BERT}_{\text{BASE}}$) could be replaced with a 8-bit integer additions [25]. This is not something typical GPUs are optimised for[1], but it would be possible to design an ASIC for this exact application. For 45nm technology, 8-bit additions require 0.8% the energy and 0.5% the area compared to 32-bit floating point multiplication [13].

[1] *input batch* refers to 32 sequences of length 128

[1] NVIDIA A100 GPUs has int8 tensor cores, enabling matrix multiplication. Using this for inference could already give a significant speedup but doesn't take advantage of the fact that weights are constrained to +-1

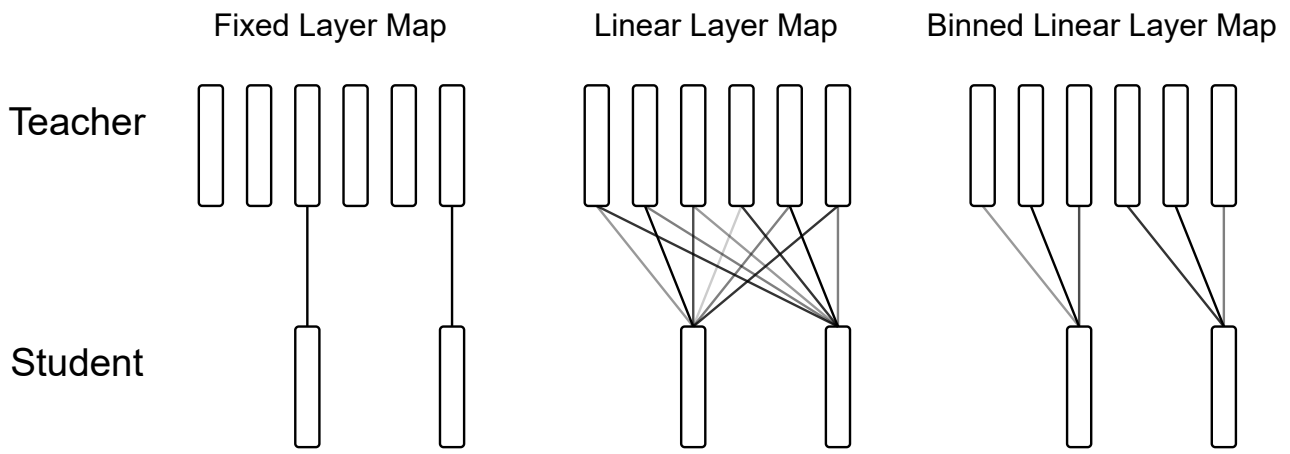


Figure 4.6: Visual comparison of the fixed, linear and binned linear layer maps, for a 6-layer teacher and 2-layer student.

Core takeaways: Reduced model size is beneficial for inference speed, energy efficiency and production costs for inference chips. With custom hardware, TernaryBERT could decrease latency and power consumption by orders of magnitude.

4.7 Linear Layer Map

Linear layer map As discussed in Section 4.4.2, one way to increase the performance of KD might be to increase the flexibility of the layer map. We also note that in my setup, only a third of the teacher layers are used for distillation, meaning that potentially useful information from the other layers are going to waste. To remedy this, I propose a *linear layer map*, where each student layer is compared against a linear combination of all teacher layers, with coefficients as learnable parameters.

Specifically, for each student layer i, we calculate a linear combination T_i^S of the teacher data T_j (can be attention scores, hidden states or something else) to compare it against. The coefficient are calculated as the softmax of learnable parameters m_{ij}, to ensure they sum to 1 and therefore do not change the magnitude of the data.

$$(m'_{i1}, m'_{i2}, ...) = \text{softmax}((m_{i1}, m_{i2}, ...))$$

$$T_i^S = \sum_j m'_{ij} T_j$$

T_i^S is then compared against S_i, the student data, to compute the distillation loss. One mapping is performed for attention matrices and one mapping for hidden states.

ALP-KD [31] uses a similar idea for their layer map, but instead of learning the weights for the layer map they are calculated by taking dot product between pairs of hidden states in the student and teacher. This is possible when the students hidden dimension is the same as the teacher, and when the student model is initialised such that its hidden vectors are already

Layer map	CoLA	RTE	SST-2
Fixed	42.2	67.1	91.6
Linear	26.4	60.0	91.9
Linear (higher prediction loss)	31.1	65.7	91.5
Masked linear	34.3	65.7	91.6

Table 4.6: Results of linear map

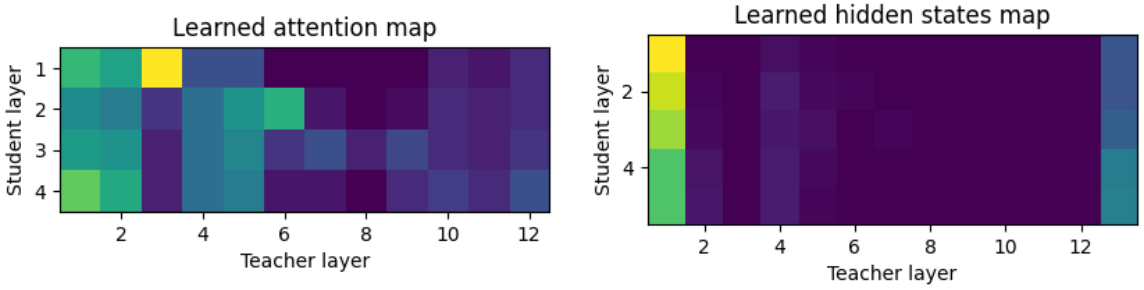


Figure 4.7: Final learned layer maps for attention and hidden state distillation on CoLA.

similar to the teacher's. ALP-KD uses a student model that only differs in number of layers from the teacher, 6 instead of 12, and initialises the student with the first 6 layers of the teacher. Our linear layer map approach is more general, and we can use it directly in the task-specific distillation of BERT_{KD4}

Training setup I test the linear layer map by running task-specific distillation on CoLA, RTE and SST-2. We train for 20 epoch using *both* transformer layer and prediction layer distillation.

We should note that as the layer map is only influencing the transformer layer distillation loss, it is not directly being optimised to make the student perform well. If we would begin by only using transformer layer distillation as done before, then the student would learn to match its intermediate states to whatever teacher layer is easiest to replicate, minimising the intermediate distillation loss. Instead, by employing both distillations at once the prediction distillation should ensure that intermediate student states remain useful, while the layer map will be optimised for mapping the most similar layers to each other.

Results Results are shown in Table 4.6 The linear layer map performs comparably to the constant layer map for SST-2, and worse on CoLA and RTE. Looking at the learnt layer maps (Figure 4.7), we see that all layers learn a very similar linear combination, heavily weighted towards the earlier teacher layers. We thus expect all the intermediate states for student's layers to be similar, which is shown to be true in Figure B.1. This means that my hypothesis that the prediction loss will force the student to keep useful intermediate representations, which in turn will guide the layer map to be useful, is false. This hints at a fundamental problem in KD loss design: if flexibility is increased to enable a more useful mapping, this introduces more learnable

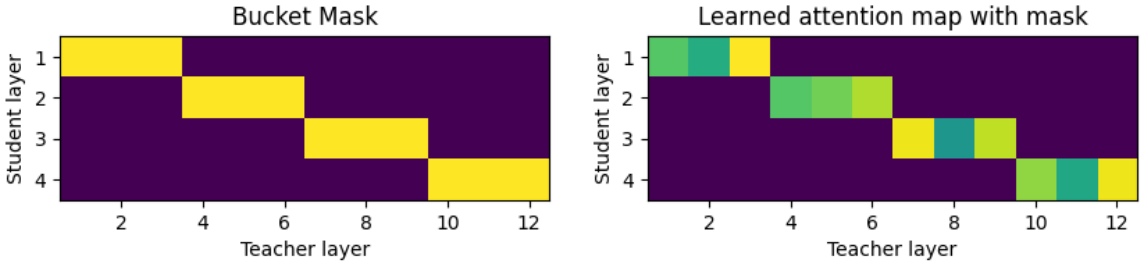


Figure 4.8: Left: Mask applied to the attention layer map for when using masked linear layer map. Right: final learned layer map for attention distillation on CoLA, using a masked linear layer map.

elements whose optimisation might not correlate to performance improvement. I was unable to find a thourough discussion of this problem in the KD literature, where learnable parameters in the loss function is common practice [18**?** , 31]. Thus, we either need to keep flexibility low, or use a a process more similar to Neural Architecture Search or hyper-parameter search to design the KD loss function. .

Improvement attempts I make two attempts to improve the performance. First attempt is multiplying prediction loss by a factor of 10, increasing its importance. This yields better results and the intermediate representations look more varied, but the layer map still only considers the early layers of the teacher and doesn't contribute to the performance. Second attempt is apply a mask to the layer mapping (Figure 4.8), to ensure that each student layer use information from different parts of the teacher. This also improves performance on RTE and CoLA, but the fixed layer map still wins.

Core takeaways: Introducing more learnable parameter in the KD loss function i not necessarily beneficial, as minimizing the loss might not be aligned with increasing performance. This problem can be partially remidied by tweaking relative importance of different losses and by decreasing flexability in the learnt parameters.

5 Conclusions

Summary The goal of this project was to reduce the size and inference cost of transformers for natural language processing. To this end, a framework for Knowledge Distillation of transformers was implemented. My framework includes support for general distillation on large text corpuses and task-specific distillation on tasks from the GLUE benchmark, a flexible library of KD loss definitions, and fast, parallellised data pre-processing and augmentation procedures.

To evaluate the implementation, I run experiments with a pre-trained $\text{BERT}_{\text{BASE}}$ as teacher and a 4 layer BERT_{KD4} student model as student. I first run general distillation on Wikipedia and then fine-tune by running task-specific distillation on each task from GLUE. The distilled model significantly outperforms a twice as large model and has similar performance to a three times as large model.

As an extension, I implemented quantised KD, where the a quantised student model is trained with a full precision model as teacher. With this technique, I train a model 15x smaller than $\text{BERT}_{\text{BASE}}$, while achieving comparable performance.

I also experiment with a more general way of mapping the student's layers to the teacher's, by learning linear combinations during training. These experiments come out negative, and do not demonstrate a performance gain over the fixed layer map used before.

Lessons Learned This work has shown that Knowledge Distillation improves the performance of small transformer models. Thus, when a larger teacher model is available, KD is a viable strategy to reduce the model size needed for a given performance. As my KD framework requires significantly fewer epochs for general distillation (~3) than BERT uses for pre-training (~40), using KD also reduces computation time for this step and enables faster and cheaper iteration on model architecture.

Furthermore, using quantisation together with KD yields even smaller models, with negligible performance decrease compared to the teacher model. However, to realise the full performance gain from the quantised model, special hardware is required.

5.1 Future Work

There are many possible directions for future work. One main area is improving the KD loss function, either by clever manual constructions similar to ALP-KD [31], or by using a search algorithm to find loss functions that produce high performing models. Training-less Neural Architecture Search algorithms, such as the one proposed by Mellor et. al [29], could perhaps be adapted for this use-case.

Future work could also further improve the training data used for distillation. This includes more extensive data augmentation procedures, or alternative data generation methods such as the adversarial data generation from COMB-KD [23].

Bibliography

[1] Afra Alishahi, Grzegorz Chrupała, and Tal Linzen. Analyzing and interpreting neural networks for nlp: A report on the first blackboxnlp workshop, 2019.

[2] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization, 2018.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[4] Colby R. Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. Benchmarking tinyml systems: Challenges and direction, 2021.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[6] Alon Brutzkus and Amir Globerson. Why do larger models generalize better? A theoretical perspective via the XOR problem. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 822–830. PMLR, 09–15 Jun 2019. URL `https://proceedings.mlr.press/v97/brutzkus19b.html`.

[7] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[8] Papers With Code. Image classification on imagenet. URL `https://paperswithcode.com/sota/image-classification-on-imagenet?metric=Number%20of%20params&dimension=Top%201%20Accuracy`.

[9] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

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

[11] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A survey of data augmentation approaches for nlp, 2021.

[12] Wikimedia Foundation. Wikimedia downloads. URL `https://dumps.wikimedia.org`.

[13] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[14] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. March 2015.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90009-T. URL `https://www.sciencedirect.com/science/article/pii/089360809190009T`.

[17] Lu Hou and James T. Kwok. Loss-aware weight quantization of deep networks, 2018.

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

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

[20] Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. Revealing the dark secrets of bert, 2019.

[21] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks, 2014.

[22] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. May 2016.

[23] Tianda Li, Ahmad Rashid, Aref Jafari, Pranav Sharma, Ali Ghodsi, and Mehdi Rezagholizadeh. How to select one among all? an extensive empirical study towards the robustness of knowledge distillation in natural language understanding, 2021.

[24] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E. Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers, 2020.

[25] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications, 2016.

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

[27] Chengqiang Lu, Jianwei Zhang, Yunfei Chu, Zhengyu Chen, Jingren Zhou, Fei Wu, Haiqing Chen, and Hongxia Yang. Knowledge distillation of transformer-based language models revisited, 2022.

[28] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, 1975. ISSN 0005-2795. doi: https://doi.org/10.1016/0005-2795(75)90109-9. URL `https://www.sciencedirect.com/science/article/pii/0005279575901099`.

[29] Joseph Mellor, Jack Turner, Amos Storkey, and Elliot J. Crowley. Neural architecture search without training, 2021.

[30] Patrick Mineault. How much energy does chatgpt use? URL `https://xcorr.net/2023/04/08/how-much-energy-does-chatgpt-use/`.

[31] Peyman Passban, Yimeng Wu, Mehdi Rezagholizadeh, and Qun Liu. Alp-kd: Attention-based layer projection for knowledge distillation, 2020.

[32] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[33] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning, 2017.

[34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2018. URL `https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf`.

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[36] Bita Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, and et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. In *NeurIPS 2020*. ACM, November 2020. URL `https://www.microsoft.com/en-us/research/publication/pushing-the-limits-of-narrow-precision-inferencing-at-cloud-scale-with-microsoft-fl`

[37] Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning, 2016.

[38] Md. Maruf Hossain Shuvo, Syed Kamrul Islam, Jianlin Cheng, and Bashir I. Morshed. Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proceedings of the IEEE*, 111(1):42–91, 2023. doi: 10.1109/JPROC.2022.3226481.

[39] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. June 2017.

[41] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[42] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision, 2019.

[43] Jason Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks, 2019.

[44] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi 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 M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[45] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, and Ellie Pavlick et al. Bloom: A 176b-parameter open-access multilingual language model, 2023.

[46] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. Unsupervised data augmentation for consistency training, 2020.

[47] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2020.

[48] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit BERT. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. IEEE, dec 2019. doi: 10.1109/emc2-nips53020.2019.00016. URL `https://doi.org/10.1109%2Femc2-nips53020.2019.00016`.

[49] Qingru Zhang, Simiao Zuo, Chen Liang, Alexander Bukharin, Pengcheng He, Weizhu Chen, and Tuo Zhao. Platon: Pruning large transformer models with upper confidence bound of weight importance, 2022.

[50] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[51] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. Ternarybert: Distillation-aware ultra-low bit bert. September 2020.

[52] Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books, 2015.

A Reference implementation discussion

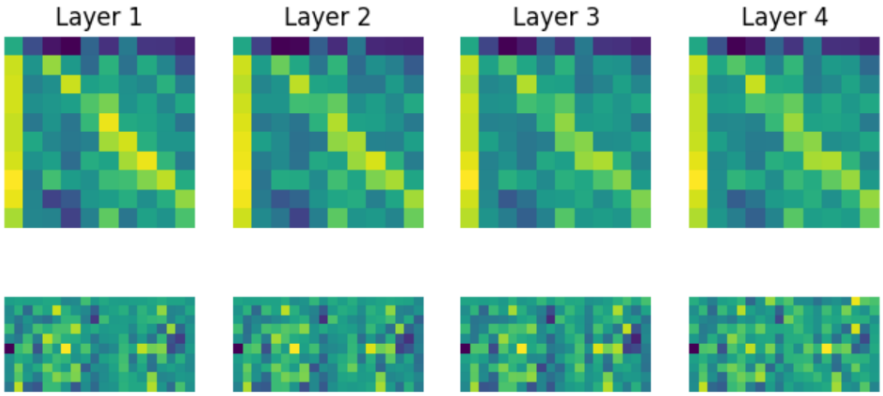
Running the open sourced implementation from the TinyBERT [18] paper, the results did not match the reported values from the paper. I run their training scripts with command line arguments set to the specified values from the paper where it exists, and leave it to default otherwise. The following are all potential differences I have identified between the TinyBERT paper and my run of their code:

- The code contains no control of randomness. We can therefore not ensure that the random initialisation, minibatching and dropout is the same as in the run in their paper. As they don't specify what seed their experiments are run with, they might also have run some training procedures multiple times and report the best result, as suggested by the BERT paper [10].
- Their code run evaluation every s steps instead of once per epoch, where s is a command line parameter. I run their script with s set to effectively evaluate once per epoch, but with the default value evalutaion happens around 5 to 50 times per epoch depending on dataset size. As mentioned in Section 4.3.1, this will increase the score, if it is taken as the maximum over all evaluations.
- They do not publish the teacher models used, and not their exact method for training them. I use the teacher models trained in Section 4.3.1. The performance of these is likely could be lower than the ones used by the TinyBERT authors. In particular, their distilled performance on MRPC is higher than my teacher model. Assuming the models are the same as used in the TernaryBERT paper [51] by the same authors, the MRPC teacher model in deed achieves higher accuracy (86.5%, compared to my 84.3%).

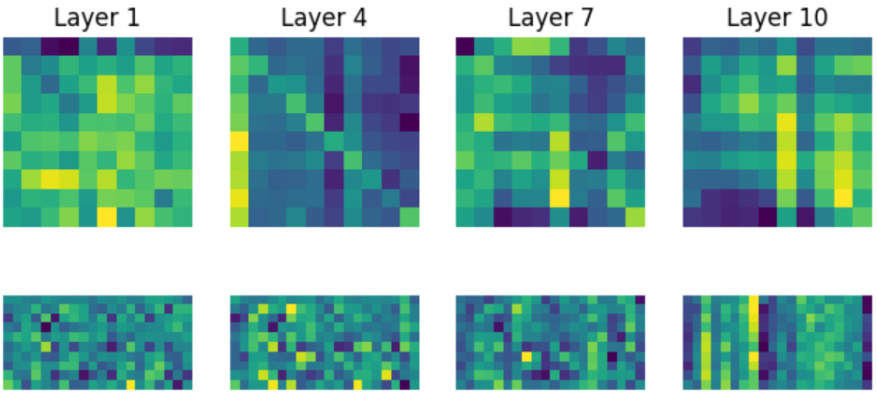
I also note that that there is some significant difference between my implementation and their, for instance in the results on SST-2, where their result is multiple standard deviations away from mine. I identified two differences in the training:

- Their code uses a linear learning rate decay for the prediction distillation step. As no other learning rate schedule is specified in the paper, I did not change this before running their code.
- As mentioned above, they use a smaller batch size of 32, compared to my batch size of 128, with learning rate scaled correspondingly.

B Linear Layer Map Activations



(a) Student model trained with linear layer map on CoLA



(b) Teacher model

Figure B.1: Internal states (attention maps on top, hidden vectors on bottom) for the first sample in the CoLA dataset. The internal states in the student model are all very similar, indicating poor utilisation of the layers.

C Project Proposal

Knowledge Distillation for Transformers

Fredrik Ekholm

October 2022

College: Trinity College
CRSID: fwe21
Director of Studies: Prof. Frank Stajano, Dr Holden
Project Supervisors: Prof. Robert Mullins, Aaron Zhao

1 Background

The field of deep learning has in recent years quickly developed to use larger and larger models. This has brought us state of the art systems in many different areas, including computer vision and natural language processing. As the models are getting bigger, more powerful and specialised systems are needed to cope with the computational costs of running them. Transformer models have been widely used in language processing tasks, and is one type of model where this has become a problem. OpenAI's GPT-3, an example of a successful transformer model, has 175 billion parameters and requires 800GB of storage to run [1]. The ability to decrease sizes of networks and computational costs of inference without a significant loss in accuracy would enable deployment of large transformer models on more resource-restricted devices, expanding the possible application areas. Knowledge Distillation (KD) is one technique that aims to achieve this.

1.1 Transformers

The transformer model [11] is a *sequence to sequence* model that transforms a sequence of vectors in multiple steps. It is built up of a series of transformer layers, each of which consist of two main sublayers: a fully connect feed-forward network (FFN) and Multi-Head Attention (MHA).

The Feed-forward network transforms each vector in the sequence independently by applying two linear layers with ReLU activation.

The Attention layer first computes values Q, K and V for each input vector through linear transformations. An attention matrix is calculated as

$$A = \frac{QK^T}{\sqrt{d_k}}$$

where d_k is a constant scaling factor. This attention matrix has been shown to capture significant linguistic knowledge. The output of the attention layer is calculated by

$$Y = softmax(A)V$$

1.2 Knowledge Distillation

The idea of Knowledge Distillation [5] is to use a larger model T to effectively extract structure from the training data, and then use this knowledge to guide the teaching of a smaller student model. More specifically, define f^T and f^S to be some *behaviour* functions of the teacher and student model respectively. These are functions from model input to some informative intermediate representation. This can for example be the activations of a specific layer. KD is formally modelled as minimising the following objective function

$$\mathcal{L}_{KD} = \sum_{x \in \mathcal{X}} L(f^S(x), f^T(x))$$

where $L(\cdot)$ is a loss function evaluating the difference in behaviour function output of the teacher and student model, and $\mathcal{X}$ is the training set.

2 Description of the Work

The first goal of the project is to investigate how effective knowledge distillation is with these models. The TinyBERT [6] paper performed Knowledge Distillation on BERT [2] achieving a student model with 96.8% of the performance while being 7.5x smaller and 9.4x faster. Reimplementing the method from this paper will serve as the core of the project. There are then many possible options for varying and extending the technique.

One design dimension is choosing how the behaviour and loss functions are defined. For transformers, we can for example consider intermediate sequence representations (*hidden states based distillation*), attention matrixes (*attention based distillation*), word embeddings (*embedding-layer distillation*) and prediction layer logits (*Prediction-layer Distillation*) as part of the behaviour functions. For the base of the project, all of these will be implemented. When the number of layers in the teacher and student models differ, there are different options available for the loss function as it isn't obvious which layer in the teacher model the features from a given layer in the student model should be compared to.

Another design dimension lies in how the reduction of size of the student network is done. Some potential choices here include decreasing the hidden vector sizes, decreasing the number of attention heads, decreasing the number of layers and introducing quantization. For the core of the project, we will only vary the hidden vector size and the number of attention heads.

The general paradigm for NLP nowadays is to pretrain a transformer on a large unlabelled text corpus and then finetune on a smaller task-specific dataset. The TinyBERT paper proposes a two step distillation process where the distillation is done both in the pretraining (General Distillation) and in the finetuning step (Task-specific Distillation).

In this project, a pretrained BERT_{base} model will be used as the teacher, and a BERT_{tiny} model will be used as the student. We use English Wikipedia (2500M words) as a pretraining dataset, and for finetuning I will be using SST [10], a sentiment dataset where the task is to classify whether sentences from movie reviews are positive or negative. This is one of the datasets in the GLUE benchmark [12].

The second core goal of the project is to implement an reproducible end-to-end experimentation framework for Knowledge Distillation of transformers. Here we mean this to be a system where all the results in the report can be replicated exactly by running a single command. The framework will take care of loading models and datasets, training, and evaluation. The system should also allow easy configuration and creation of new experiments to run.

3 Main steps of the project

1. Load a pretrained BERT_{base} and finetune on SST.
2. Build end-to-end experimentation pipeline for Knowledge Distillation on transformers, with support for loading the relevant datasets and support for two stage distillation process proposed in TinyBERT.
3. Implement the four types of distillation (*hidden states based distillation*, *attention based distillation*, *embedding-layer distillation* and *Prediction-layer Distillation*) from TinyBERT in this pipeline.
4. Run experiments.
5. Work on extensions.

4 Starting point

I will be using Python and PyTorch as the main deep learning framework. I have experience with both of these from personal projects and internships. This is mostly training simple feed forward models and some convolutional models, but I have also experimented a bit with transformers.

I am not previously familiar with the area of knowledge distillation, but have been reading some of the relevant papers leading up to this project.

Some of the papers mentioned have existing open source implementations available. However, I will reimplement them from scratch in order to integrate them into a common experimentation framework.

The Huggingface [1] library will be used for loading pretrained models, tokenizers and datasets.

The English Wikipedia dataset will be downloaded from Wikimedia[2] and pre-processed using WikiExtractor[3]

5 Success Criteria

The project is a success if it achieves the following:

- Finetuning of BERT_{base} on SST with at least around 90% accuracy.
- An implementation of an end-to-end framework for training models with Knowledge Distillation and running reproducible experiments.
- Implement the four types of knowledge distillation for transformers from TinyBERT (*hidden states based distillation*, *attention based distillation*, *embedding-layer distillation* and *Prediction-layer Distillation*)
- Experiments running the distillation procedure and evaluating the performance of the distilled models.
- Report accuracy on the test set, along with number of parameters and FLOPs (Floating Operators) of the distilled model.

6 Extensions

More datasets We can try to replicate results across more datasets. QNLI and MNLI, two dataset that also are part of the GLUE benchmark[12] would be good to start with. We can further extend to all the datasets in GLUE.

Quantization Investigating quantization as another method of reducing size and decreasing computational costs in the student model is a possible extension. Quantization has been successfully applied to transformers [8]. I can and compare different methods of quantization, such as fixed-point quantization, MSFloat[9], ternarization [7] such as loss-aware ternarization [13].

Varying knowledge distillation behaviour and loss functions As outlined in the background section, there are many design choices involved in how exactly the distillation is performed. Heo et al. [4] compares some of these, and can serve as inspiration for possible choices.

- For *Hidded state distillation*, if the dimensions of intermediate sequence vectors differ we have to perform some tranformation in the behaviour functions to bring them to a comparable format. The most basic solution is to compare the magnitude of the vectors, but this way a lot of information is lost. Another approach is to either pass the teacher's or the student's vectors through an extra single linear layer to bring them to the same dimension.
- For both *Attention matrix distillation* and *Hidded state distillation* we can choose to train the student on the representation before or after the acivation function (soft-max and ReLU respectively).
- The distance function used for the distillation loss can be varied. Heo et al.[4] introduces *partial* L_2, which ignores negative values when the student's value is lower than the teacher's. This is argued to work well when ReLU is used as activation function.

[1] https://huggingface.co/

[2] https://en.wikipedia.org/wiki/Wikipedia:Database_download

[3] https://github.com/attardi/wikiextractor

Varying number of layers of student model. If we start varying the number of layers we run into some further challenges in designing the loss function for distillation. The distillation of TinyBERT is hand-designed - the output of every n layers connects to the output of the student layer. We could try different options here, such as comparing each layer in the student model to the average accross a set of layers in the teacher model, or perhaps learning a mapping from the teacher layers to the student layers as part of the distillation process. There is also an approach where a GNN is used to guide the distillation, detailed in [14].

Other tasks We can investigate knowledge transfer on different tasks where transformers are used, such as image classification with vision transformers [3]. For this, I would train a ViT on ImageNet.

7 Timetable and Milestones

Oct 17 - Oct 30

- Setup github, install required packages on development machines.
- Ensure access to compute resources (research group's local GPU server, JADE, Wilkes3), familiarize myself with routine for running jobs on these systems.
- Run finetuning of BERT$_base$ on SST.
- **Milestone:** Have a complete and tested setup for loading datasets and finetuning BERT.

Oct 31 - Nov 13

- Implement dataset loading and perform necessary pre-processing for Wikipedia Text.
- Start implementing the General Distillation step, starting with only Prediction-layer Distillation
- **Milestone:** Being able to load the Wikipedia dataset as a sequence of tokens.

Nov 14 - Nov 27

- Finish implementing the General Distillation step
- Run some sanity check evaluation of the trained student model on the fill-in-the-missing-words task that BERT is pretrained on.
- **Milestone:** A working implementation of the General Distillation step.

Nov 28 - Dec 11

- Implement the Task-specific Distillation step. This should not be too difficult already having a code for finetuning BERT and code for General Distillation.
- Implement the other three distillation types (*hidden states based distillation*, *attention based distillation*, *embedding-layer distillation*). This will also be fairly straightforward as it only requires modification of the KD behaviour and loss functions.
- **Milestone:** A working implementation of the whole Knowledge Distillation pipeline from TinyBERT, with the two distillation steps and all the four distillation types.

Dec 12 - Dec 25

- Run evaluation and produce results
- Buffer time for catch-up.

Dec 26 - Jan 8

- Buffer time for catch-up.
- Start writing dissertation

Jan 9 - Jan 22

- Run KD from BERT$_base$ to BERT$_tiny$ on all the datasets in GLUE benchmark (first extension).
- **Milestone:** Results on all datasets in the GLUE benchmark.

Jan 23 - Feb 5

- Write progress report
- Implement fixed-point quantization (second extension).
- As much as time allows, implement different types of quantization: MSFloat, ternarization, loss-aware ternarization.
- **Milestone:** Submit progress report, report results from KD with fixed-point quantization.

Feb 6 - Feb 19

- Compare performance of the different design choices for KD behaviour and loss functions, as outlined in the third extension.
- Finish writing the two first chapters of the dissertation
- **Milestone: Empirical results showing the optimal design choices for KD, rough draft of the first two chapters of the dissertation.**

Feb 20 - Mar 5

- Buffer time and time for working more on extensions
- Write implementation chapter for dissertation

Mar 6 - Mar 19

- Buffer time and time for working more on extensions
- Finish writing implementation chapter for dissertation
- **Milestone:** rough draft of the first 3 chapters of the dissertation

Mar 20 - Apr 2

- Write evaluation and conclusion chapters of dissertation.

Apr 3 - Apr 16

- Finish write evaluation and conclusion chapters of dissertation.
- **Milestone:** rough draft dissertation finished.

Apr 17 - Apr 30

- Collect feedback and iterate on the dissertation.
- **Milestone:** Final draft of dissertation finished

May 1 - May 14

- Buffer time for any final tweaks
- **Milestone:** Hand in the final dissertation

8 Resource Declaration

I have two personal laptops where I will be doing all the development and running small scale experiments. The first one is an Acer Predator with 16GB ram, an Intel i5 7th gen processor and a GTX 1060 graphics card. The second one is a Lenovo Yoga with 16GB ram and an AMD Ryzen 5 5600H.

I accept full responsibility for these machines and I have made contingency plans to protect myself against hardware and/or software failure. In case one of these fails, I will be able to continue working using the other one.

I will use GitHub as a repository for my code, that I will keep in frequent sync with the local repository. This servers as a backup in case of machine failure.

Robert Mullins and Aaron Zhao have agreed to give me access to three different compute resources where larger experiments will be run:

- A local server of the research group with 4 GTX 2080 GPUs
- JADE at Oxford (https://www.arc.ox.ac.uk/jade)
- Wilkes3 at Cambridge (https://www.hpc.cam.ac.uk/systems/wilkes-3)

If one of these systems fails or I am unable to use it, I can use the others as a backup.

References

[1] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: (May 2020). arXiv: 2005.14165 [cs.CL].

[2] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: (Oct. 2018). arXiv: 1810.04805 [cs.CL].

[3] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: (Oct. 2020). arXiv: 2010.11929 [cs.CV].

[4] Byeongho Heo et al. "A Comprehensive Overhaul of Feature Distillation". In: (Apr. 2019). arXiv: 1904.01866 [cs.CV].

[5] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: (Mar. 2015). arXiv: 1503.02531 [stat.ML].

[6] Xiaoqi Jiao et al. "TinyBERT: Distilling BERT for Natural Language Understanding". In: (Sept. 2019). arXiv: 1909.10351 [cs.CL].

[7] Fengfu Li, Bo Zhang, and Bin Liu. "Ternary Weight Networks". In: (May 2016). arXiv: 1605.04711 [cs.CV].

[8] Gabriele Prato, Ella Charlaix, and Mehdi Rezagholizadeh. "Fully Quantized Transformer for Machine Translation". In: (Oct. 2019). arXiv: 1910.10485 [cs.CL].

[9] Bita Rouhani et al. "Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point". In: *NeurIPS 2020*. ACM. Nov. 2020. URL: https://www.microsoft.com/en-us/research/publication/pushing-the-limits-of-narrow-precision-inferencing-at-cloud-scale-with-microsoft-floating-point/.

[10] Richard Socher et al. "Recursive deep models for semantic compositionality over a sentiment treebank". In: *EMNLP* 1631 (Jan. 2013), pp. 1631–1642.

[11] Ashish Vaswani et al. "Attention Is All You Need". In: (June 2017). arXiv: 1706.03762 [cs.CL].

[12] Alex Wang et al. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: (Apr. 2018). arXiv: 1804.07461 [cs.CL].

[13] Wei Zhang et al. "TernaryBERT: Distillation-aware Ultra-low Bit BERT". In: (Sept. 2020). arXiv: 2009.12812 [cs.CL].

[14] Sheng Zhou et al. "Distilling Holistic Knowledge with Graph Neural Networks". In: (Aug. 2021). arXiv: 2108.05507 [cs.CV].