Teacher-student knowledge distillation from BERT for sentence classifiers

Sam Sučík

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

TO-DO

Acknowledgements

I thank Steve Renals of University of Edinburgh and Vova Vlasov of Rasa for supervising me throughout the academic year; patiently listening to my never-ending reports and providing helpful and optimistic comments.

Many thanks also to Ralph Tang whose work inspired this project, and to Slávka Heželyová who constantly supported me and motivated me to explain all of my work in non-technical stories and metaphors.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

- After the deep learning hype started, NLP went through an era of LSTMs. Since 2017, the area has been becoming dominated by Transformer models pre-trained on large unlabelled corpora.

- As newer and bigger Transformer-based models were proposed in 2018 and 2019, improving on the SOTA, it was becoming clearer that their big size and low speed was rendering them difficult to use (both train and deploy) in practice outside of research labs.

- Recently, we've seen various early attempts at making Transformers – in particular BERT (Devlin et al., 2018) – smaller by removing attentional heads (Michel et al., 2019), quantisation and pruning (Cheong and Daniel, 2019; Sucik, 2019). In terms of actually down-sizing and accelerating the models, knowledge transfer using teacher-student knowledge distillation has led to the most attractive results (Mukherjee and Awadallah, 2019; Tang et al., 2019b; Jiao et al., 2019; Sanh et al., 2019).

- However, these studies focus only on using knowledge distillation as a tool. Important questions about the nature of this technique and how it interacts with properties of the teacher and student models remain generally unexplored.

- In line with the increasing demand for explainable AI, it is desirable that, for the beginning, at least the researchers better understand the tools they use, in this case distillation of NLP knowledge from Transformer models. Indeed, such understanding is also useful for overcoming the limitations and designing new variants of this method for smaller and better classifiers.

1.2 Aims

I aim to better understand knowledge distillation by exploring its use for knowledge transfer from BERT into different student architectures on various NLP tasks.

This can be further broken down into three aims:

- Explore the effectiveness of knowledge distillation in very different NLP tasks. To cover a broad variety of tasks, I use sentence classification datasets ranging from binary sentiment classification to 57-way intent classification to linguistic acceptability.
- Explore how distilling knowledge from a Transformer varies with different student architectures. I limit myself to using the extremely popular BERT model (Devlin et al., 2018) as the teacher architecture. As students, I use two different architectures: a BiLSTM, building on the successful work of Ralph Tang (Tang et al., 2019b,a), and a down-scaled BERT architecture.
- Explore how successfully can different types of NLP knowledge and capabilities be distilled. Since NLP tasks are often possible for humans to reason about, I analyse the models' behaviour (e.g. the mistakes they make) to learn more about knowledge distillation. I also probe the models for different linguistical capabilities, inspired by previous successful probing studies (Conneau et al., 2018; Tenney et al., 2019a).

1.3 Contributions

My actual findings. To be added later.

Chapter 2

Background [READY FOR REVIEW]

In this chapter, the Transformer models are introduced and set into the historical context; knowledge distillation is introduced, in particular its recent applications in NLP; and an overview of some relevant work in model understanding is given.

2.1 NLP before Transformers

By the very nature of the natural language, its processing has always meant processing sequences of variable length: be it written phrases or sentences, words (sequences of characters), spoken utterances, sentence pairs, or entire documents. Very often, NLP tasks boil down to making simple decisions about such sequences: classifying sentences based on their intent or language, assigning a score to a document based on its formality, deciding whether two given sentences form a meaningful question-answer pair, or predicting the next word of an unfinished sentence.

As early as 2008, artificial neural networks started playing a key role in NLP: Collobert and Weston (2008)[1] successfully trained a deep neural model to perform a variety of tasks from part-of-speech tagging to semantic role labelling. However, neural machine learning models are typically suited for tasks where the dimensionality of inputs is known and fixed. Thus, it comes as no surprise that NLP research has focused on developing better models that encode variable-length sequences into fixed-length representations. If any sequence (e.g. a sentence) can be embedded as a vector in a fixed-dimensionality space, a simple classification model can be learned on top of these vectors.

One key step in the development of neural sequence encoder models has been the idea of *word embeddings*: rich, dense, fixed-length numerical representations of words. When viewed as a lookup table – one vector per each supported word –

[1] See also Collobert et al. (2011).

such embeddings can be used to "translate" input words into vectors which are then processed further. Mikolov et al. (2013) introduced an efficient and improved way of learning high-quality word embeddings: *word2vec*. The embeddings are learnt as part of a larger neural network, which is trained to predict the next word given several previous words, and the previous words given the current word[2]. Such training can easily leverage large amounts of unlabelled text data and the embeddings learn to capture various properties from a word's morphology to its semantics. Released word2vec embeddings became very popular due to their easy use and performance improvements in many NLP tasks. **TO-DO: Add a simple illustration of learning word2vec.**

While word embeddings were a breakthrough, they themselves do not address the issue of encoding a sequence of words into a fixed-size representation. This is where Recurrent neural networks (RNNs) (Rumelhart et al., 1986) and later their improved variant – Long Short-Term Memory neural networks (LSTMs) (Hochreiter and Schmidhuber, 1997) – come into play. Although originally proposed long ago, they became popular in NLP, and in text processing in particular, only later (see e.g. Mikolov et al. (2010) and Graves (2013)). These recurrent encoders process one word at a time (see Fig. 2.1) while updating an internal ("hidden") fixed-size representation of the text seen so far. Once the entire sequence is processed, the hidden representation (also called "hidden state") is outputted and used make a simple prediction.

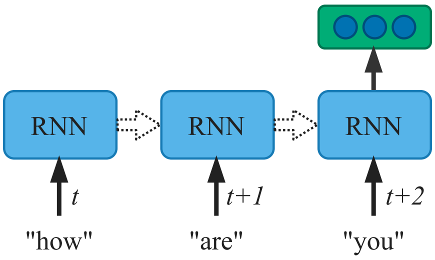


Figure 2.1. A recurrent neural network (RNN) consumes at each timestep one input word. Then, it produces a single vector representation of the inputs.

As various recurrent models started dominating NLP, one particularly influential architecture emerged, addressing tasks such as machine translation, where the output is a new sequence rather than a simple decision. This was the *encoder-decoder* architecture (Kalchbrenner and Blunsom, 2013; Sutskever et al., 2014), see Fig. 2.2. It uses a recurrent encoder to turn an input sentence into a single vector, and a recurrent decoder to generate an output sequence based on the vector.

Bahdanau et al. (2014) improved encoder-decoder models by introducing the concept of *attention*. The attention module helps the decoder produce better output by selectively focusing on the most relevant parts of the input at each

[2] These are the so-called Continuous bag-of-words (CBOW) and Skip-gram (SG) tasks, respectively.

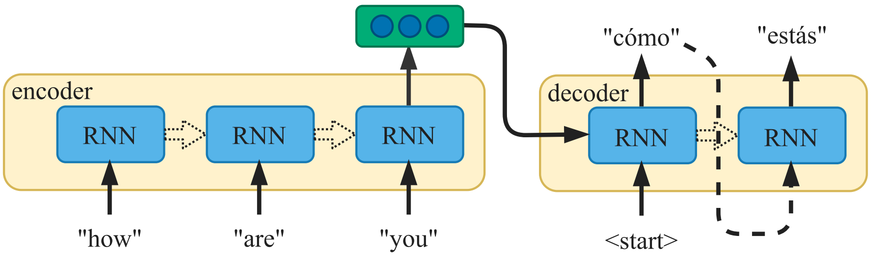


Figure 2.2. An encoder-decoder model for machine translation. Notice how the decoder initially takes as input the special `<start>` token and at later time consumes the previous output word.

decoder timestep. This is depicted in Fig. 2.3, showing the decoder just about to output the second word ("estás"). The steps (as numbered in the diagram) are:

1. the decoder's hidden state passed to the attention module,
2. the intermediate hidden states of the encoder also passed to the attention module,
3. the attention module, based on information from the decoder's state, selecting relevant information from the encoder's hidden states and combining it into the attentional *context vector*,
4. the decoder combining the last outputted word ("cómo") with the context vector and consuming this information to better decide which word to output next.

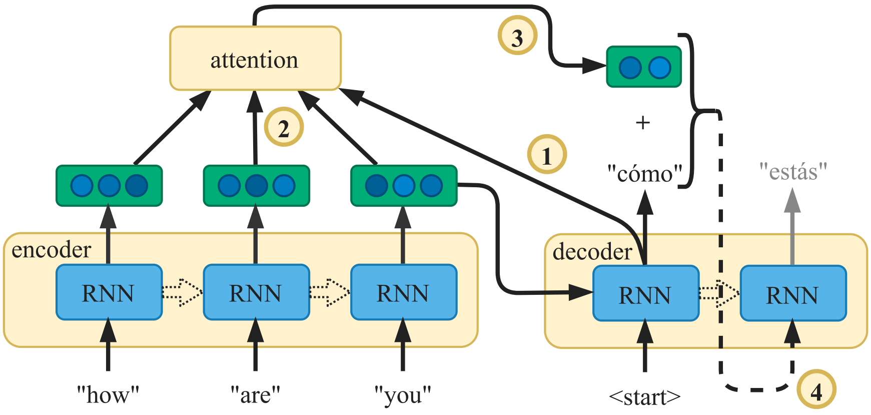


Figure 2.3. An encoder-decoder model for machine translation with added attention mechanism.

The attention can be described more formally[3]: First, the decoder state $\boldsymbol{h_D}$ is processed into a *query* $\boldsymbol{q}$ using

$$\boldsymbol{q} = \boldsymbol{h_D} W_Q \tag{2.1}$$

[3] My description does not exactly follow the original works of Bahdanau et al. (2014) and Luong et al. (2015). Instead, I introduce concepts that will be useful in later sections of this work.

and each encoder state $\boldsymbol{h}_E^{(i)}$ is used to produce the *key* and *value* vectors, $\boldsymbol{k}^{(i)}$ and $\boldsymbol{v}^{(i)}$:

$$\boldsymbol{k}^{(i)} = \boldsymbol{h}_E^{(i)} \boldsymbol{W_K}, \quad \boldsymbol{v}^{(i)} = \boldsymbol{h}_E^{(i)} \boldsymbol{W_V} \ . \tag{2.2}$$

Then, the selective focus of the attention is computed as an *attention weight* $w^{(i)}$ for each encoder state i, by combining the query with the i-th key:

$$w^{(i)} = \boldsymbol{q}^\top \boldsymbol{k}^{(i)} \ . \tag{2.3}$$

The weights are normalised using softmax and used to create the context vector c as a weighted average of the values:

$$\boldsymbol{c} = \sum_i a^{(i)} \boldsymbol{v}^{(i)} \quad \text{where} \quad a^{(i)} = \mathrm{softmax}(w^{(i)}) = \frac{\exp\left(w^{(i)}\right)}{\sum_j \exp\left(w^{(j)}\right)} \ . \tag{2.4}$$

Note that W_Q, W_K, W_V are matrices of learnable parameters, optimised in training the model. This way, the attention's "informed selectivity" improves over time.

For years, recurrent models with attention were the state of the art in many NLP tasks. However, as we will see, the potential of attention reached far beyond recurrent models.

2.2 Transformer-based NLP

2.2.1 Transformers

We saw how the attention mechanism can selectively focus on parts of a sequence to extract relevant information from it. This raises the question of whether processing the inputs in a sequential fashion with the recurrent encoder is still needed. In particular, RNN models are slow as a results of this sequentiality, with no room for parallelisation. In their influential work, Vaswani et al. (2017) proposed an encoder-decoder model based solely on attention and fully parallelised: the *Transformer*. The core element of the model is the *self-attention* mechanism, used to process all input words in parallel.

In particular, a Transformer model typically has multiple self-attention layers, each layer processing separate representations of all input words. Continuing with the three-word input example from Fig. 2.3, a high-level diagram of the workings of a self-attention layer is shown in Fig. 2.4. Importantly, the input word representations evolve from lower to higher layers such that they consider not just the one input word, but also all other words – the representation becomes *contextual* (also referred to as a *contextual embedding* of the word within the input sentence).

As for the internals of self-attention, the basic principle is very similar to standard attention. Self-attention too is used to focus on and gather relevant information

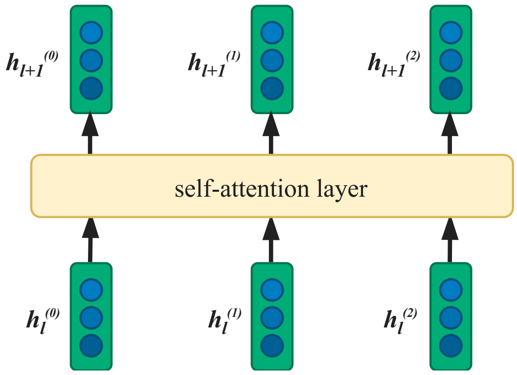


Figure 2.4. A high-level diagram of the application of self-attention in Transformer models. Three hidden states are shown for consistency with the length of the input shown in Fig. 2.3; in general, the input length can vary.

from a sequence of elements, given a query. However, to produce a richer contextual embedding $\boldsymbol{h}_{l+1}^{(i)}$ in layer $l+1$ of the i-th input word, self-attention uses the incoming representation $\boldsymbol{h}_{\boldsymbol{l}}^{(i)}$ for the query, and considers focusing on all representations in layer l, including $\boldsymbol{h}_{\boldsymbol{l}}^{(i)}$ itself. Fig. 2.5 shows this in detail for input position $i = 0$. Query $\boldsymbol{q}^{(0)}$ is produced and matched with every key in layer l (i.e. $\boldsymbol{k}^{(0)}, \dots,$ $\boldsymbol{k}^{(2)}$) to produce the attention weights. These weights quantify how relevant each representation $\boldsymbol{h}_{\boldsymbol{l}}^{(i)}$ is with respect to position $i = 0$. Then, the new contextual embedding $\boldsymbol{h}_{l+1}^{(i)}$ is constructed as a weighted sum of the values $\boldsymbol{v}^{(0)}, \dots,$ $\boldsymbol{v}^{(2)}$ (same as constructing the context vector in standard attention).

Notice that, even though each contextual embedding considers all input positions, the next-layer contextual embeddings $\boldsymbol{h}_{l+1}^{(0)}, \dots,$ $\boldsymbol{h}_{l+1}^{(2)}$ can be computed all at the same time, in parallel: First, the keys, queries and values for all input positions are computed; then, the attention weights with respect to each position are produced; finally, all the new representations are produced. It is this parallelism that allows Transformer models to run faster. As a result, they can be much bigger (and hence create richer input representations) than recurrent models while taking the same time to train.

Due to their parallel nature, self-attentional layers have no notion of an element's position within the input sequence. This means no sensitivity to word order. (Recurrent models sense this order quite naturally because they process input text word by word.) To alleviate this downside of self-attention, Transformers use *positional embeddings*. These are artificially created numerical vectors added to each input word, different across input positions, thus enabling the model's layers to learn to be position- and order-sensitive.

As an additional improvement of the self-attentional mechanism, Vaswani et al. introduce the concept of multiple self-attention heads. This is very similar to having multiple instances of the self-attention module in Fig. 2.5 (each instance being one *head*). The motivation behind multiple self-attention heads is to enable each head a to learn different "focusing skills" by learning its own $W_{Q,a}$, $W_{K,a}$,

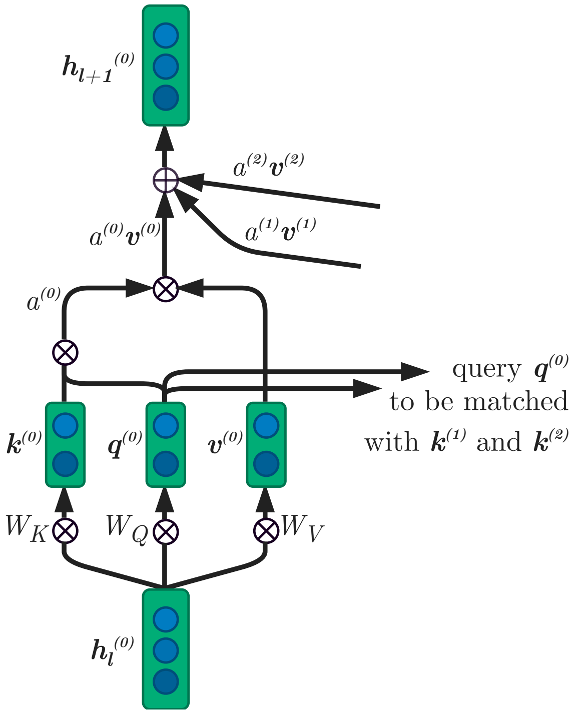


Figure 2.5. The internals of self-attention, illustrated on creating the next-layer hidden representation of the input position $i = 0$, given all representations in the current layer. Note that $\otimes$ stands for multiplication (where the multiplication involves a learnable matrix like W_K, this is written next to the $\otimes$), and $\oplus$ denotes summation.

$W_{V,a}$. Each head produces its own output:

$$O_{att,a} = \text{softmax}(\frac{\boldsymbol{q}\boldsymbol{k}^\top}{\sqrt{d_k}})\boldsymbol{v} = \text{softmax}(\frac{(\boldsymbol{h_l}^\top W_{Q,a})^\top(\boldsymbol{h_l}^\top W_{K,a})}{\sqrt{d_k}})(\boldsymbol{h_l}^\top W_{V,a}) \tag{2.5}$$

which matches Fig. 2.5 (but notice the detail of the additional scaling by $\frac{1}{\sqrt{d_k}}$, introduced by Vaswani et al., where d_k is the dimensionality of the key). The outputs of the A individual attentional heads are then concatenated and dimensionality reduced with a trainable linear transformation W_{AO}, to produce the final output, which replaces $\boldsymbol{h}_{l+1}$ in Fig. 2.5:

$$O_{att} = [O_{att,1},\ \ldots,\ O_{att,A}]W_{AO}\ . \tag{2.6}$$

Besides the self-attention-based architecture, there is one more important property that makes today's Transformer models perform so well on a wide variety of NLP tasks: the way these models are trained. First used for Transformers by Radford et al. (2018)[4], the general procedure is:

1. *Unsupervised pre-training*: The model is trained on one or more tasks, typically language modelling, using huge training corpora. For example, Radford et al. pre-train their model to do next word prediction (the standard language modelling task) on a huge corpus of over 7,000 books.

[4] The idea was previously used with recurrent models by Dai and Le (2015).

2. *Supervised fine-tuning*: The pre-trained model is trained on a concrete dataset to perform a desired downstream task, such as predicting the sentiment of a sentence, translating between languages, etc.

This two-step procedure is conceptually similar to using pre-trained word embeddings. In both cases, the aim is to learn general language knowledge and then use this as a starting point for focusing on a particular task. However, in this newer case, the word representations learned in pre-training are be better tailored to the specific architecture, and they are inherently contextual – compared to pre-trained word embeddings like word2vec which are typically context-insensitive.

Importantly, pre-trained knowledge makes models more suitable for downstream tasks with limited amounts of labelled data. The model no longer needs to acquire all the desired knowledge just from the small dataset; it contains pre-trained high-quality general language knowledge which can be reused in various downstream tasks. This means that large, powerful Transformer models become more accessible: They are successfully applicable to a wider array of smaller tasks than large models that have to be trained from scratch.

2.2.2 BERT

Perhaps the most popular Transformer model today is BERT (Bidirectional Encoder Representations from Transformers), proposed by Devlin et al. (2018). Architecturally, it is a sequence encoder, hence suited for sequence classification tasks. While being heavily based on the original Transformer (Vaswani et al., 2017), BERT adds a few further tricks:

1. The model learns bidirectional representations: It can be trained on language modelling that is not next-word prediction (prediction given left context), but word prediction given both the left and the right context.

2. It uses two very different pre-training classification tasks:

   (a) The *masked language modelling* (MLM) task encourages BERT to learn good contextual word embeddings. The task itself is to correctly predict the token at a given position in a sentence, given that the model can see the entire sentence with the target token(s) masked out[5], with a different token, or left unchanged.

   (b) The *next-sentence prediction* (NSP) task encourages BERT to learn good sentence-level representations. Given two sentences, the task is to predict whether they formed a consecutive sentence pair in the text they came from, or not.

   The pre-training was carried out on text from books and from the English Wikipedia, totalling to 3,400 million words (for details see Devlin et al.

[5] I.e. replaced with the special `[MASK]` token.

(2018)). The MLM and NSP tasks were both used throughout the pre-training, forcing the model to learn both at the same time.

3. The inputs are processed not word by word, but are broken down using a fixed vocabulary of sub-word units (*wordpieces*, introduced by Wu et al. (2016)). This way, BERT can better deal with rare words. (In word-level models, words that are not found in the model's vocabulary are replaced with a special `UNKNOWN` token, which means disregarding any information carried by the words.) The tokeniser module of BERT uses the wordpiece vocabulary to tokenise (segment) the input text before it is further processed. Fig. 2.6 shows an example; notice how my surname ("Sucik") gets split into three wordpieces whereas the other, much more common words are found in the wordpiece vocabulary.

4. To enable the different pre-training tasks as well as two-sentence inputs, BERT uses a special input sequence representation, illustrated in Fig. 2.6. Given the two input sentences S_A, S_B, they are concatenated and separated by the special `[SEP]` token. The overall sequence is prepended with the `[CLS]` (classification) token. To explicitly capture that certain tokens belong to S_A and others to S_B, simple *token type embeddings* (which only take on two different values) are added to the token embedding at each position. Then, for tasks like NSP, only the output representation of the `[CLS]` token (i.e. $\boldsymbol{o_0}$) is used, whereas for token-level tasks like MLM the output vector from the desired position is used (in Fig. 2.6, the MLM task would use $\boldsymbol{o_3}$ to predict the correct token at this position).

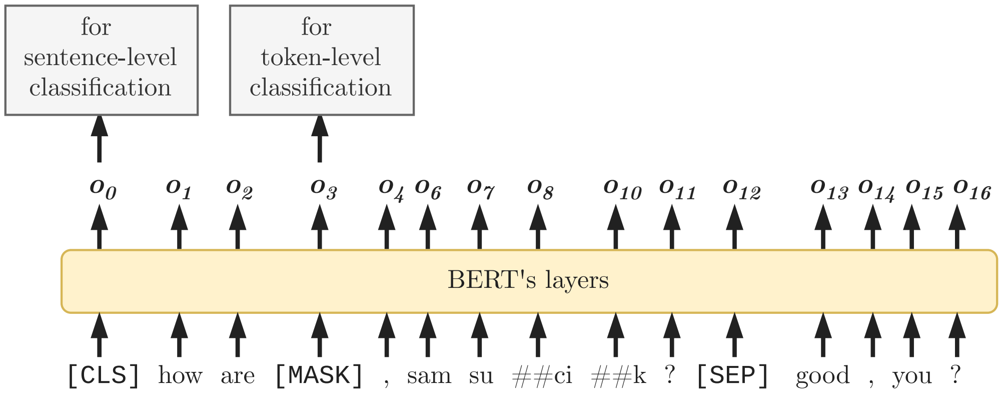


Figure 2.6. BERT's handling of input for sentence-level and token-leven tasks. The input sentences ($S_A = \mathit{How\ are\ you,\ Sam\ Sucik?}$ and $S_B = \mathit{Good,\ you?}$) are shown as split by BERT's tokeniser, with the first instance of "you" masked out for MLM.

The overall architecture of BERT is shown in Fig. 2.7. The tokeniser also adds the special tokens like `[CLS]` and `[SEP]` to the input, while the trainable token embedding layer also adds the positional embedding and the token type embedding to the wordpiece embedding of each individual token. The pooler takes the appropriate model output (for sequence level classification the first output $\boldsymbol{o_0}$ as discussed above) and applies a fully-connected layer with the tanh activation function. The external classifier is often another fully-connected layer with the tanh activation, producing the logits[6]. These get normalised using softmax to produce a probability distribution over all classes. The most probable class is outputted as the model's prediction.

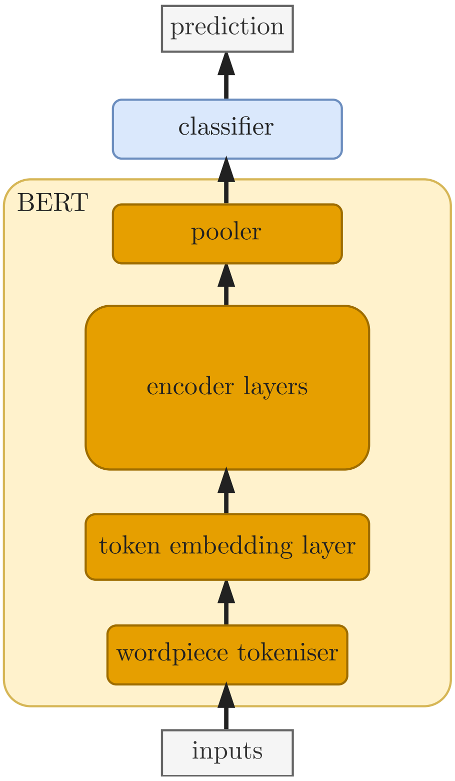


Figure 2.7. High-level overview of the modules that make up the architecture of BERT as used for sequence-level classification.

To complete the picture of BERT, Fig. 2.8 shows the internals of an encoder layer. Besides the multi-headed self-attention submodule, it also contains the fully-connected submodule. This uses a very wide intermediate fully-connected transformation with parameters W_I, inflating the representations up to the dimensionality d_I, and the layer output fully-connected transformation with parameters W_O, which reduces the dimensionality. Each submodule is also by-passed by a residual connection (shown with dashed lines). The residual information is summed with the submodule's output, and layer normalisation is applied to the sum. Note that this structure is not new in BERT; it was used already by the original Transformer of Vaswani et al. (2017). Conveniently, Transformers are designed such that all of the intermediate representations (especially the encoder inputs and outputs, and the self-attention layer inputs and outputs) have the same dimensionality d_h – this makes any residual by-passing and summing easy.

When training BERT, artificial corruption of internal representations is done using dropout, which acts as a regulariser, making the training more robust. In particular, dropout is applied to the outputs of the embedding layer, to the

[6] For a classifier, the logits are the (unnormalised) predicted class probabilities.

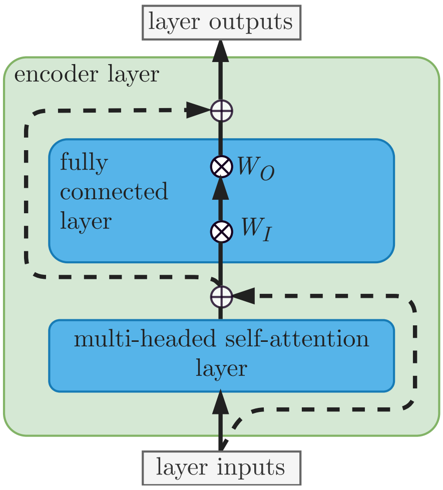


Figure 2.8. The modules making up one encoder layer in BERT; residual connections highlighted by using dashed lines.

computed attention weights, just before residual summation both to the self-attention layer output and to the fully connected layer output (see Fig. 2.8 for the summation points), and to the output of the pooler module (before aplying the external classifier, see Fig. 2.7). The typical dropout rate used is 0.1.

Originally, pre-trained BERT was released in two sizes: BERT$_{\text{Base}}$ with 110 million parameters, 12 encoder layers and 12-head self-attention, and BERT$_{\text{Large}}$ with 340 million parameters, 24 encoder layers and 16-head self-attention. The models quickly became popular, successfully applied to various tasks from document classification (Adhikari et al., 2019) to video captioning (Sun et al., 2019). Further pre-trained versions were released too, covering, for example, the specific domain of biomedical text (Lee et al., 2019) or multilingual text (Pires et al., 2019).

2.2.3 Newer and larger Transformer models

Following the success of the early Transformers and BERT (Vaswani et al., 2017; Radford et al., 2018; Devlin et al., 2018), many further model variants started emerging, including:

- The OpenAI team releasing GPT-2 (Radford et al., 2019), a larger and improved version of their original, simple Transformer model GPT (Radford et al., 2018).
- Lample and Conneau (2019) introducing XLM, which uses cross-lingual pre-training and is thus better suited for downstream tasks in different languages.
- Transformer-XL (Dai et al., 2019), which features an improved self-attention that can handle very long contexts (across multiple sentences/documents).

All these open-sourced, powerful pre-trained models were a significant step towards more accessible high-quality NLP (in the context of downstream tasks with limited data). However, the model size – often in 100s of million trainable parameters – meant these models could not be applied easily in practice (outside of research): They were memory-hungry and slow.

Naturally, this inspired another stream of research: Compressing large, well-performing Transformer models (very often BERT) to make them faster and resource-efficient. I turn my focus to one compression method that worked particularly well so far: the teacher-student knowledge distillation.

2.3 Teacher-student knowledge distillation

2.3.1 A brief introduction to knowledge distillation

Knowledge distillation was introduced by (Bucila et al., 2006) as a way of knowledge transfer from large models into small ones. The aim is to end up with a smaller – and hence faster – yet well-performing model. The steps are 1) to train a big neural classifier model (also called the *teacher*), 2) to let a smaller neural classifier model (the *student*) learn from it – by learning to mimic the teacher's behaviour. Hence also the name *teacher-student knowledge distillation*, often simply *knowledge distillation*.

There are different ways of defining the teacher's "behaviour" which the student learns to mimic. Originally, this was realised as learning to mimic the teacher's predictions: A dataset would be labelled by the teacher, and the student would be trained on these labels (which are in this context referred to as the *hard labels*). The dataset used for training the student (together with the teacher-generated labels) is referred to as the *transfer dataset*.

Later, Ba and Caruana (2014) introduced the idea of learning from the teacher-generated *soft labels*, which are the teacher's logits. The idea is to provide the student with richer information about the teacher's decisions: While hard labels only express which class had the highest predicted probability, soft labels also describe how confident the prediction was and which other classes (and to what extent) the teacher was considering for a given example.

When soft labels were first used, the student's training loss function was the mean squared distance between the student's and the teacher's logits:

$$E_{MSE} = \sum_{c=1}^{C} (z_t^{(c)} - z_s^{(c)})^2 \qquad (2.7)$$

where C is the number of classes and z_t, z_s are the the teacher's and student's logits. Hinton et al. (2015) proposed a more general approach, addressing the issue of overconfident teachers with very sharp logit distributions. The issue with such distributions is that they carry little additional information beyond the hard label (since the winning class has a huge probability and all others have negligibly small probabilities). To "soften" such sharp distributions, Hinton et al. proposed using the *cross-entropy loss* (2.8) in combination with softmax with temperature (2.9) (instead of the standard softmax) in training both the teacher and the student.

$$E_{CE} = \sum_{c=1}^{C} z_t^{(c)} \log z_s^{(c)} \tag{2.8}$$

$$p_c = \frac{\exp\left(z^{(c)}/T\right)}{\sum_{c'=1}^{C} \exp\left(z^{(c')}/T\right)} \tag{2.9}$$

The temperature parameter T determines the extent to which the distribution will be "unsharpened" – two extremes being the completely flat, uniform distribution (for $T \rightarrow \infty$) and the maximally sharp distribution[7] (for $T \rightarrow 0$). When $T > 1$, the distribution gets softened and the student can extract richer information from it. Today, using soft labels with the cross-entropy loss with temperature is what many refer to simply as knowledge distillation.

Since 2015, further knowledge distillation variants have been proposed, enhancing the vanilla technique in various ways, for example:

- Papamakarios (2015, p. 13) points out that mimicking teacher outputs can be extended to mimicking mimicking the *derivatives* of the teacher's loss with respect to the inputs. This is realised by including in the student's loss function also the term: $\frac{\partial \boldsymbol{o}_s}{\partial \boldsymbol{x}} - \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{x}}$ ($\boldsymbol{x}$ being an input, e.g. a sentence, and $\boldsymbol{o}$ being the output, e.g. the predicted class).

- Romero et al. (2015) proposed to additionally match the teacher's internal, intermediate representations of the input. Huang and Wang (2017) achieved this by learning to align the distributions of neuron selectivity patterns between the teacher's and the student's hidden layers. Unlike standard knowledge distillation, this approach is no longer limited only to classifier models with softmax outputs (see the approach of Hinton et al. (2015) discussed above).

- Sau and Balasubramanian (2016) showed that learning can be more effective when noise is added to the teacher logits.

- Mirzadeh et al. (2019) showed that when the teacher is much larger than the student, knowledge distillation performs poorly, and improved on this by "multi-stage" distillation: First, knowledge is distilled from the teacher into an intermediate-size "teacher assistant" model, then from the assistant into the final student.

2.3.2 Knowledge distillation in NLP

The knowledge distillation research discussed so far was tied to the image processing domain. This is not surprising: Image processing was the first area to

[7] I.e. having the preferred class's probability 1 and the other classes' probabilities 0.

start taking advantage of deep learning, and bigger and bigger models had been researched ever since the revolutional AlexNet (Krizhevsky et al., 2012).

In NLP, the (recurrent) models were moderately sized for a long time, not attracting much research in model compression. Still, one early notable work was on adapting knowledge distillation for sequence-to-sequence models (Kim and Rush, 2016), while another pioneering study (Yu et al., 2018) distilled a recurrent model into en even smaller one – to make it suitable for running on mobile devices.

Understandably, the real need for model compression started very recently, when the large pre-trained Transformer models became popular. Large size and low speed seemed to be the only downside of these – otherwise very successful and accessible – models.

Perhaps the first decision to make when distilling large pre-trained models is at which point to distill. In particular, one can distill the general knowledge from a pre-trained teacher and use such general student by fine-tuning it on downstream tasks, or one can fine-tune the pre-trained teacher on a task and then distill this specialised knowledge into a student model meant for the one task only. Each of these approaches has its advantages and disadvantages.

In the first scenario (distilling pre-trained knowledge), a major advantage is that the distillation happens once and the small student can be fine-tuned fast for various downstream tasks. Since the distillation can be done on the same data that the teacher was pre-trained on – large unlabelled text corpora –, lack of transfer data is not a concern. A possible risk is that the large amount of general pre-trained language knowledge will not "fit" into the small student, requiring the student itself to be considerably large. Sanh et al. (2019) took this approach and, while their student is successfully fine-tuned for a wide range of tasks, it is only 40% smaller than the BERT$_{\text{Base}}$ teacher.

In the second scenario, only the task-specific knowledge needs to be transferred to the student – potentially allowing very small students. However, teacher fine-tuning and distillation have to be done anew for each task and this is resource-hungry. Additionally, there may be a lack of transfer data if the downstream task dataset is small. Various ways of addressing this issue by *augmenting* small datasets have been proposed, with mixed success. Mukherjee and Awadallah (2019) use additional unlabelled in-domain sentences with labels generated by the teacher – this is limited to cases where such in-domain data are available. Tang et al. (2019b) create additional sentences using simple, rule-based perturbation of existing sentences from the downstream dataset. Finally, Jiao et al. (2019) and Tang et al. (2019a) use large Transformer models generatively to create new sentences. In the first case, BERT is applied repeatedly to an existing sentence, changing words into different ones one by one and thus generating a new sentence. In the second case, new sentences are sampled token-by-token from a GPT-2 model fine-tuned on the downstream dataset with the next-token-prediction objective.

Clearly, each approach is preferred in a different situation: If the requirement is to compress the model as much as possible, and there is enough transfer data, distilling the fine-tuned teacher is more promising. If, on the other hand, one wants to make available a re-usable, small model, then distilling the broader, pre-trained knowledge is preferred.

2.4 Understanding NLP models

Neural models are by their very nature opaque or even black boxes, and (not) really understanding the models is a serious concern. Despite the typical preferrence of performance over transparence, recently, the demand for explainable artificial intelligence (XAI) has been increasing, as neural models become widely used.

The area of image processing has seen the most attempts at interpreting neural models, partly because reasoning about images is easy for humans. Various techniques shed light into the workings of image classifiers, e.g. creating images that maximally excite certain neurons (Simonyan et al., 2013) or highlighting those parts of an image that a particular neuron "focuses" on (Zeiler and Fergus, 2013).

In NLP, interpreting is more difficult and historically came with a delay similar to the delay with which large neural models became popular for NLP tasks. In their review, Belinkov and Glass (2018) observe that many methods for analysing and interpreting models are simply adapted from image processing, in particular the approach of visualising a single neuron's focus, given an input. In attentional sequence-to-sequence models, the attention maps can be visualised to explore the soft alignments between input and output words (see, e.g., Strobelt et al. (2018)). However, these methods are mostly qualitative and suitable for exploring individual input examples, thus not well suited for drawing statistically backed conclusions or for model comparison.

More quantitative and NLP-specific are the approaches that explore the linguistic knowledge present in a model's internal representations. Most often, this is realised by *probing* the representations for specific linguistic knowledge: trying to automatically recover from them specific properties of the input. When such recovery works well, the representations must have contained the linguistic knowledge tied to the input property in question. First used by Shi et al. (2016) for exploring syntactic knowledge captured by machine translation models, this general approach was quickly adopted more widely. Adi et al. (2017) explored sentence encodings from recurrent models by probing for simple properties like sentence length, word content and word order. More recently, Conneau et al. (2018) curated a set of 10 probing tasks ranging from easy surface properties (e.g. sentence length) through syntactic (e.g. the depth of the syntactic parse tree) to semantic ones (e.g. identifying semantically disrupted sentences). Focusing on Transformers, Tenney et al. (2019b) proposed a set of *edge probing* tasks, examining how much contextual knowledge about an entire input sentence is captured within the contextual representation of one of its words. Their tasks correspond to the typical steps of a text processing pipeline – from part-of-speech (POS) tagging to identifying dependencies and entities to semantic role labelling. Tenney et al. (2019a) managed to localise the layers of BERT most important for each of these "skills". They showed that the ordering of these "centres of expertise" within BERT's encoder matches the usual low- to high-level order: from simple POS tagging in the earlier layers to the most complex semantic tasks in the last layers.

While the discussed approaches provide valuable insights, they merely help us intuitively describe or quantify the kinds of internal knowledge/expertise present in the models. Gilpin et al. (2018) call this level of model understanding *interpretability* – comprehending what a model does. However, they argue that what we should strive to achieve is *explainability*: the ability to "summarize the reasons for neural network behavior, gain the trust of users, or produce insights about the causes of their decisions". In this sense, today's methods achieve only interpretability because they enable researchers to describe but not explain – especially in terms of causality – the internals and decisions of the models. Still, interpreting models is an important step not only towards explaining them, but also towards understanding the properties of different architectures and methods and improving them.

Chapter 3

Datasets [READY FOR REVIEW]

In this chapter, I introduce the downstream tasks and how I augmented their data to create large transfer datasets. I also introduce the probing tasks used later for analysing the teacher and student models.

3.1 Downstream tasks

The downstream task datasets I use to fine-tune the teacher model. The tasks are chosen to be diverse so that the knowledge distillation analysis later in this work is set in a wide NLP context. At the same time, all the datasets are rather small and therefore well representing the type of use case where pre-trained models like BERT are desirable due to the lack of labelled fine-tuning data.

Today, perhaps the most widely used collection of challenging NLP tasks[1] is the GLUE benchmarking collection (Wang et al., 2018). This collection comprises 11 tasks which enable model benchmarking on a wide range of NLP tasks from sentiment analysis to detecting textual similarity, all framed as single-sentence or sentence-pair classification. Each task comes with an official scoring metric (such as accuracy or F1), labelled training and evaluation datasets, and a testing dataset with labels not released publicly. The test-set score accumulated over all 11 tasks forms the basis for the popular GLUE leaderboard[2].

In this work, I use single-sentence classification tasks (i.e. not sentence-pair tasks). Therefore, only two GLUE tasks are suitable for my purposes – the Corpus of Linguistic Acceptability (CoLA) and the Stanford Sentiment Treebank in its binary classification variant (SST-2). As the third task, I use an intent classification dataset called Sara, which is not part of the GLUE collection.

[1] Challenging by the nature of the tasks and by the small dataset size.

[2] `gluebenchmark.com/leaderboard`

3.1.1 Corpus of Linguistic Acceptability

The CoLA dataset (Warstadt et al., 2018) comprises roughly 8,500 training sentences, 1,000 evaluation and 1,000 testing sentences. The task is to predict whether a given sentence represents acceptable English or not (binary classification). All the sentences are collected from linguistic literature where they were originally hand-crafted to demonstrate various linguistic principles and their violations.

The enormous variety of principles, together with many hand-crafted sentences that comply with or violate a principle in a niche way, make this dataset very challenging even for the state-of-the-art Transformer models. As a non-native speaker, I myself struggle with some of the sentences, for instance:

- **The car honked down the road.* (unacceptable)
- *Us, we'll go together.* (acceptable)

There are many examples which are easy for humans to classify but may be challenging for models which have imperfect understanding of the real world. Sentences like "Mary revealed himself to John." require the model to understand that "Mary", being a typical female name, disagrees with the masculine "himself".

The scoring metric is Matthew's Correlation Coefficient (MCC) (Matthews, 1975), a correlation measure between two binary classifications. The coefficient is also designed to be robust against class imbalance, which is important because the dataset contains many more acceptable examples than unacceptable ones.

3.1.2 Stanford Sentiment Treebank

The SST-2 dataset (Socher et al., 2013) is considerably bigger than CoLA, with roughly 67,000 training examples, 900 evaluation and 1,800 testing examples. It contains sentences and phrases from movie reviews collected on `rottentomatoes.com`. The main SST dataset comes with human-created sentiment annotations on the continuous scale from very negative to very positive. SST-2 is a simplified version with neutral-sentiment phrases removed, only containing binary sentiment labels (positive and negative).

Unlike the hand-crafted examples in CoLA, many examples in SST-2 are not the best-quality examples. In particular, sentences are sometimes split into somewhat arbitrary segments[3], such as:

- *should have been someone else -* (negative)
- *but it could have been worse.* (negative)

The labels are also sometimes unclear, see:

[3] This is due to the use of an automated parser in creating the dataset.

- *american chai encourages rueful laughter at stereotypes only an indian-american would recognize.* (negative)
- *you won't like roger, but you will quickly recognize him.* (negative)

Despite the problematic examples, most are straightforward (e.g. "delightfully cheeky" or "with little logic or continuity"), making this task a relatively easy one. With accuracy being the official metric, best models in the GLUE leaderboard score over 97%, very close to the official human baseline of 97.8%[4].

3.1.3 Sara

As the third task, I use an intent classification dataset created by Rasa, a start-up building open-source tools for conversational AI[5].

The dataset is named Sara after the chatbot deployed on the company's website[6] The Sara chatbot is aimed for holding conversations with the website visitors on various topics, primarily answering common questions about Rasa and the tools that it develops (the same tools were used to build Sara). To support diverse topics, Sara internally classifies each human message as one of 57 intents and then generates an appropriate response. The Sara dataset is a collection of human-generated message examples for each of the 57 intents, e.g.:

- *what's the weather like where you are?* (ask_weather)
- *what is rasa actually* (ask_whatisrasa)
- *yes please!* (affirm)
- *i need help setting up* (install_rasa)
- *where is mexico?* (out_of_scope)

TO-DO: List of all intents with brief descriptions or representative examples? Perhaps in the appendix...

In the early days of the chatbot, it supported fewer intents, and several artificial examples per intent were first hand-crafted by Rasa employees to train the initial version of Sara's intent classifier. After Sara was deployed, more examples were collected and annotated from conversations with the website's visitors[7]. Inspired by the topics that people tended to ask about, new intent categories were added. Today, the dataset still evolves and can be found – together with the implementation of Sara – at github.com/RasaHQ/rasa-demo (accessed April 3,

[4] See the GLUE leaderboard at gluebenchmark.com/leaderboard

[5] For transparency: I did a Machine learning research internship with Rasa in the summer of 2019.

[6] See the bot in action at rasa.com/docs/getting-started/.

[7] To get a consent for such use of the conversations, each visitor was shown the following before starting a conversation with Sara: "Hi, I'm Sara! By chatting to me you agree to our privacy policy.", with a link to rasa.com/privacy-policy/

2020). It contains both the original hand-crafted examples as well as the (much more abundant) "real" examples.

The Sara dataset version I use dates back to October 2019, when I obtained it from Rasa and pseudonymised the data[8]. In particular, I removed any names of persons and e-mail addresses in any of the examples, replacing them with the special tokens `__PERSON_NAME__` and `__EMAIL_ADDRESS__`, respectively. The dataset comprises roughly 4,800 examples overall, and was originally split into 1,000 testing examples and 3,800 training examples. I further split the training partition into training and evaluation, with roughly 2,800 and 1,000 examples, respectively. All three partitions have the same class distribution.

In line with how the dataset is used for research at Rasa, I use as the main scoring metric the multi-class micro-averaged F_1 score. This variant of multi-class $F1$ considers all examples combined and is computed from the overall precision P and recall R in the usual way: $F_{1micro} = \frac{2PR}{P+R}$. Here, the overall P and R are calculated from the overall true-positives TP, false-positives FP and false-negatives FN:

$$P = TP/(TP+FP), \qquad R = TP/(TP+FN) \tag{3.1}$$

where TP is in this case the total number of correctly predicted examples (hits H), and both FN and FP equal the number of incorrect predictions (misses M). Hence:

$$F_{1micro} = H/(H+M) = P = R = accuracy\ . \tag{3.2}$$

Compared to another popular multi-class metric – the macro-averaged $F1$, the F_{1micro} score takes into account the sizes of different classes by looking at all examples in combination. This way, the overall score cannot be harshly pulled down just because of low recall or precision on some small, rare class.

3.2 Data augmentation for larger transfer datasets

As discussed in Section 2.3.2, the transfer datasets in knowledge distillation often need to be augmented; if they are too small, they don't provide enough opportunity for the teacher to "demonstrate its knowledge" to the student, and the student learns little.

Tang et al. (2019a) demonstrated on several GLUE tasks that using an augmented training portion for distillation leads to much better student performance than using just the original small training portion. For CoLA in particular, using just the small training set led to very poor student performance (see Table 1 in Tang et al.).

I take the augmentation approach that Tang et al. found to work the best: Generating additional sentences using a GPT-2 model (Radford et al., 2019)

[8] As a former employee (intern) of Rasa, I got access to the data under the NDA I had signed with the company.

fine-tuned on the training set[9]. The steps for creating the transfer dataset from the training portion are:

1. Fine-tune the pre-trained GPT-2 model (the 345-million-parameter version) on the training portion for 1 epoch with the language-modelling objective (i.e. predicting the next subword token given the sequence of tokens so far).

2. Sample from the model a large number of tokens to be used as the beginnings (*prefixes*) of the augmentation sentences. This sampling can be done as one-step next-token prediction given the special `SOS` (start-of-sentence) token.

3. Starting from each sampled prefix, generate an entire sentence token by token by repeatedly predicting the next token using the GPT-2 model. The generation of a sentence stops when the special `EOS` (end-of-sentence) token is generated or when the desired maximum sequence length is reached (in this case 128 tokens).

4. Add the generated augmentation sentences to the original training data, and generate the teacher logits for each sentence.

For consistency with Tang et al. (2019a), I added 800,000 augmentation sentences to the training data of each of the three downstream tasks, resulting in the transfer datasets comprising roughly 808,500, 867,000, and 802,800 sentences for CoLA, SST-2, and Sara, respectively.

3.3 Probing tasks

The probing tasks (discussed in Section 2.4) I use after knowledge distillation to analyse the linguistic capabilities of the students and the teacher. In particular, I use the probing suite curated by Conneau et al. (2018), consisting of 10 tasks[10].

Each probing task is a collection of 120,000 labelled sentences, split into training (100,000), evaluation (10,000) and test (10,000) set. The label refers to a property of the sentence, such as the sentence's length. The aim is to recover the property from an encoding of the sentence, produced by the model being probed. Fig. 3.1 shows the basic workflow. First, the model is used to produce an encoding of each sentence. Then, a light-weight classifier is trained, taking the training sentences' encodings as inputs and learning to produce the labels. The evaluation sentence encodings are used to optimise the hyperparameters of the classifier. Finally, a probing score (accuracy) is produced on the test encodings. The score quantifies how well the sentence property in question is recoverable (and thus present) in the encodings. This serves as a proxy measure of the linguistic knowledge tied

[9] I used the code for Tang et al. (2019a) which is available at `github.com/castorini/d-bert` (accessed April 4, 2020).

[10] The data, along with code for probing neural models, are publicly available as part of the SentEval toolkit for evaluating sentence representations (Conneau and Kiela, 2018) at `github.com/facebookresearch/SentEval` (accessed April 4, 2020).

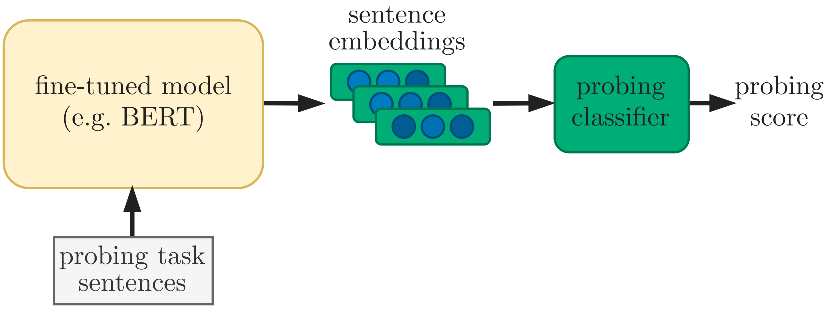


Figure 3.1. A high-level diagram of the probing process.

to the property. If, for instance, the property to be recovered is the depth of a sentence's syntactic parse tree, the score hints at the model's (un)capability to understand (and parse) the syntax of input sentences.

Regarding the linguistic capabilities explored by the probing suite, each task falls into one of three broad categories – surface properties, syntax, and semantics:

1. Surface information:
   - **Length** is about recovering the length of the sentence. The labels are somewhat simplified: The actual sentence lengths grouped into 6 equal-width bins – making this task a 6-way classification.
   - **WordContent** is about identifying which words are present in the sentence. A collection of 1000 mid-frequency words was curated, and sentences were chosen such that each contains exactly one of these words. The task is identify which one (1000-way classification).
2. Syntactic information:
   - **Depth** is about classifying sentences by their syntactic parse tree depth, with depths ranging from 5 to 12 (hence 8-way classification).
   - **BigramShift** is about sensitivity to (un)natural word order – identifying sentences in which the order of two randomly chosen adjacent words has been swapped (binary classification).
   - **TopConstituents** is about recognising the top syntactic constituents – the nodes found in the syntactic parse tree just below the S (sentence) node. This is framed as 20-way classification, choosing from 19 most common top-constituent groups + the option of "other".
3. Semantic information:
   - **Tense** is binary classification task, identifying the tense (present or past) of the sentence – given by the main verb of the sentence (the verb in the main clause). At the first sight, this is mainly a morphological task (in English, most verbs have the past tense marked by the "-d/ed"

suffix). However, the model first has to identify the main verb within a sentence, which makes this task also semantic.

- **SubjNumber** is about determining the number (singular or plural) of the sentence's subject (binary classification). Similar to the previous task, this one (and the next one too) is arguably about both morphology and semantics.
- **ObjNumber** is the same as SubjNumber, applied to the direct object of a sentence.
- **OddManOut** is binary classification, identifying sentences in which a randomly chosen verb or noun has been replaced with a different random verb or noun. Presumably, the random replacement in most cases makes the sentence semantically unusual or invalid (e.g. in "He reached inside his persona and pulled out a slim, rectangular black case." the word "persona" is clearly odd). To make this task more difficult, the replacement word is chosen such that the frequency of the bigrams in the sentence stays roughly the same. (Otherwise, in many cases, the random replacement would create easy hints for the probing classifier, in the form of bigrams that are very unusual.)
- **CoordinationInversion** works with sentences that contain two coordinate clauses (typically joined by a conjunction), e.g. "I ran to my dad, but he was gone." In half of the sentences, the order of the two clauses was swapped, producing sentences like: "He was gone, but I ran to my dad." The task is to identify the changed sentences (which are often semantically broken).

When choosing from the existing probing suites, I considered that of Tenney et al. (2019a) as well. As the authors showed, their tasks and methods can effectively localise different types of linguistic knowledge in a Transformer model like BERT. However, the task data are not freely available, the tasks have a relatively narrow coverage with heavy focus on the most complex NLP tasks like entity recognition and natural language inference, and the probing is done on single-token representations. The suite of Conneau et al., on the other hand, is publicly available, better covers the easier tasks (surface and syntactic information), and examines whole-sentence representations. In the future, however, I expect the two probing suites to be combined and used frequently together, as they complement each other.

Chapter 4

Methods and Implementation [READY FOR REVIEW]

In this chapter, I re-iterate on the main objectives of this work, the approach I took, and go into detail in describing the design and implementation work underlying my experiments and analyses.

4.1 Methods and objectives

My main aim is to explore knowledge distillation by using it. In particular, I use it on three different NLP tasks (CoLA, SST-2, Sara) and with two different student architectures (a bi-directional LSTM student and a BERT student). I further analyse and compare the teacher and students on each task. I do not aim for beating someone else's reported scores or for finding the best hyperparameter values for a model; I aim to learn more about knowledge distillation.

Because this work is heavily inspired by mi internship at Rasa on *compressing* BERT[1], I aim to produce student models as small as possible. Therefore, I take the approach of first fine-tuning a teacher model and then distilling only the fine-tuned knowledge into tiny students (see the discussion back in Section 2.3.2).

The most technical and time-demanding part of my work is setting up the system's implementation and using knowledge distillation to create small yet well-performing students for further analysis. Naturally, this involves optimisation of the student models' hyperparameters to make the students perform at least somewhat comparably with today's models like BERT. (There is little point in trying to get insights by analysing models that perform poorly in the first place.) However, I try to carry out only the essential optimisation, ending up with *a* well-performing student instead of *the* best one possible.

[1] See `blog.rasa.com/compressing-bert-for-faster-prediction-2/` and `blog.rasa.com/pruning-bert-to-accelerate-inference/`, accessed April 4, 2020.

In order to reasonably constrain the amount of time and work put into optimisation of students' hyperparameters, I carry out a more thorough parameter exploration only on the CoLA task. Subsequently, I use the best found student parameters on the other two tasks, only adapting the most essential parameters – like the student model size – on each task separately.

Only once good students are trained, the key part of this work begins. I analyse what the students learnt well and what they did not, how they differ from their teacher and from each other. Where possible, I try to generalise across the three datasets. However, some findings are simply specific to a particular downstream task, and each task can provide different insights.

As the first step of the analysis, I probe all models. This is relatively straightforward and produces clear, quantitative results. (Indeed, it may not be easy to draw clear conclusions from the results.)

As the second step, I carry out a (mostly qualitative) analysis of the models' predictions on concrete sentences. This analysis is much more time-consuming and is not widely used. However, I still believe that it can provide some useful insights. In particular, I look at predictions both through correctness – e.g. manually analysing sentences which were classified correctly by one model but not by another – and through confidence – which models are more confident, on what sentences are they (un)confident, how this relates to their (in)correctness...

Finally, I try to combine the probing and prediction analysis results into overall conclusions, seeing whether the two methods agree or disagree with each other and whether they help to describe the same or different aspects of the models and of knowledge distillation.

Because of the unavailability of test labels in CoLA and SST-2, I carry out the prediction analysis on all three tasks on the evaluation sentences. This can be understood as sheding light on the model qualities being optimised when one tunes a model's hyperparameters on the evaluation portion. Another option would be to carry out the analysis on a held-out set not used in training.

4.2 System overview and adapted implementations

Because a lot of research around Transformers is open-sourced, I make use of existing works and implementations. Fig. 4.1 shows the high-level pipeline of my work. It is inspired by the best pipeline of Tang et al. (2019a), with the exception that they only used the recurrent bi-directional LSTM (BiLSTM) student and did not carry out probing or prediction analysis.

For most of my implementation, I use the `transformers` open-source PyTorch library (Wolf et al., 2019)[2], which provides tools for working with pre-trained Transformers like BERT. For knowledge distillation, I adapt the code of Sanh

[2] `github.com/huggingface/transformers`, accessed April 4, 2020

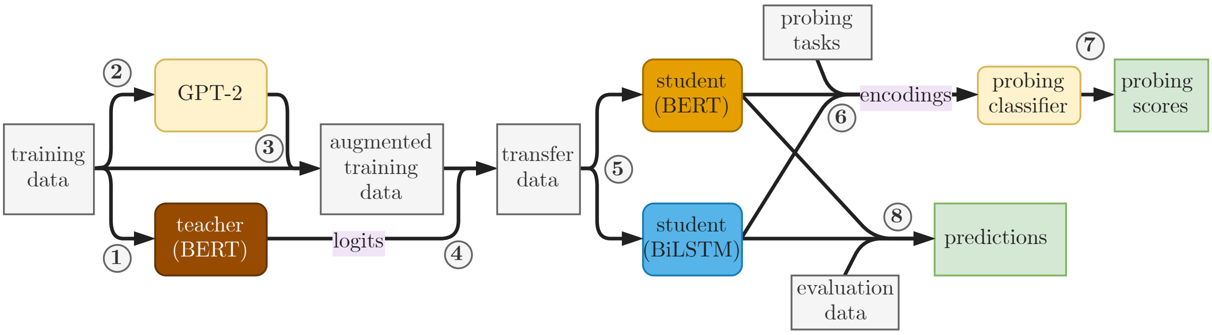


Figure 4.1. The main pipeline of this work: ① teacher fine-tuning, ② GPT-2 fine-tuning, ③ generating augmentation sentences, ④ adding teacher logits to the augmented training dataset, ⑤ knowledge distillation into students, ⑥ producing probing sentence encodings, ⑦ training the probing classifier and producing probing scores, ⑧ producing predictions on evaluation sentences.

et al. (2019), which is today also part of `transformers`[3]. (Note that they do knowledge distillation *before* downstream fine-tuning.) For augmenting the training data using GPT-2 and for knowledge distillation with the BiLSTM student, I adapt the code of Tang et al.[4], which uses an early version of `transformers`. For probing the two students, I use the SentEval framework (Conneau and Kiela, 2018)[5].

In terms of implementation, the most of my own work lies in adapting and integrating the different codebases into one, and in adding the possibility for optimising various student parameters (see the next section). I also make the code more general compared to the original codebases, which in many cases contained hard-coded decisions relating to the respective studies (be it Sanh et al. or Tang et al.). The core of my implementation can be found in my forked version of `transformers` at `github.com/samsucik/pytorch-transformers/`, with code for running experiment and analyses, as well as results and reporting, found in the project-specific repository at `github.com/samsucik/knowledge-distil-bert`.

4.3 Implementation details

4.3.1 Teacher fine-tuning

Following Tang et al. (2019a), I use the case-insensitive pre-trained BERT$_{\text{Large}}$ as the teacher. From now, I will refer to this model as BERT$_{\text{T}}$. With $L = 24$ encoder layers, $A = 16$ self-attention heads, the hidden dimension $d_h = 1024$ and the indermediate dimension $d_I = 4096$, the model has 340 million trainable parameters

[3] `github.com/huggingface/transformers/tree/master/examples/distillation`, accessed April 4, 2020

[4] `github.com/castorini/d-bert`, accessed April 4, 2020

[5] `github.com/facebookresearch/SentEval`

(as discussed in more detail back in Section 2.2.2). This large BERT generally performs better than the 110-million-parameter $\mathrm{BERT}_{\mathrm{Base}}$ variant and is therefore more desirable, but also slower, with a greater incentive for compression (in this case using knowledge distillation).

For teacher fine-tuning on each downstream task, I also use the procedure of Tang et al.: 3-epoch training with batch size $B = 36$ on the training set, using the cross-entropy loss and the Adam optimizer with $\eta = 5 \times 10^{-5}$ and the usual β values ($\beta_1 = 0.9$, $\beta_2 = 0.999$). The learning rate η was linearly "warmed up" (annealed from 0 to the target value) over the first 10% of the total training steps, and decayed linearly to 0 over the remaining steps. For regularisation, I use the standard dropout of 0.1 (see Section 2.2.2 for an overview of where dropout is used inside BERT).

Somewhat surprisingly, while the performance of $\mathrm{BERT}_{\mathrm{T}}$ converged (plateaued) within the 3-epoch budget on CoLA and SST-2 (as measured by the task-relevant metric on the evaluation set), the convergence was much slower for Sara. Hence, I empirically found a more suitable number of epochs within which the teacher would converge on Sara: 10. See Fig. 4.2 for the evaluation-set performance of the teacher models and how they converge during fine-tuning.

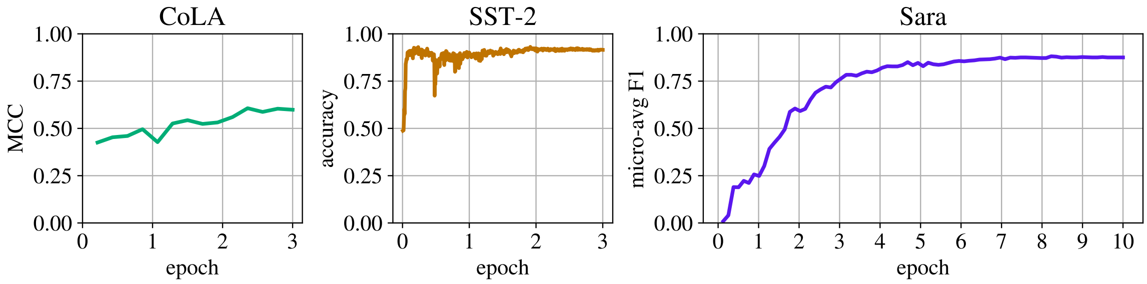


Figure 4.2. The evaluation-set performance of teacher models across fine-tuning. Unintentionally, I used different logging frequencies in fine-tuning the teachers, hence the SST-2 plot is dense (and appears more noisy) while the CoLA plot is sparse.

4.3.2 Augmentation with GPT-2

In fine-tuning the GPT-2 model, I again repeat the procedure of Tang et al.. In particular, the model is trained over 1 epoch using the cross-entropy loss, the AdamW learning algorithm (Adam with weight decay, see Loshchilov and Hutter (2019)) with learning rate $\eta = 5 \times 10^{-5}$, weight decay $\lambda = 1 \times 10^{-3}$, and the usual β values. The same linear scheduling is used as the one for fine-tuning $\mathrm{BERT}_{\mathrm{T}}$. The only parameter I choose differently from Tang et al. is the batch size B: They used batches of 48 whereas I use smaller batches of 16 examples to make the fine-tuning possible with my limited memory resource. **TO-DO: I did not note the perplexity numbers when fine-tuning GPT-2. Should I report these? Fine-tuning GPT-2 does not take long.**

4.3.3 BiLSTM student model

As the first student, I use the bi-directional LSTM (BiLSTM) from Tang et al., see Fig. 4.3, comprising in particulas one hidden BiLSTM layer with 300 units. The last hidden states for either of the two processing directions are concatenated and passed to a fully connected layer with 400 output units[6], the ReLU activation function and dropout. A final (linear) layer follows, projecting to the number of target classes, i.e. producing the logits. The model is topped with a softmax classifier for normalising the logits.

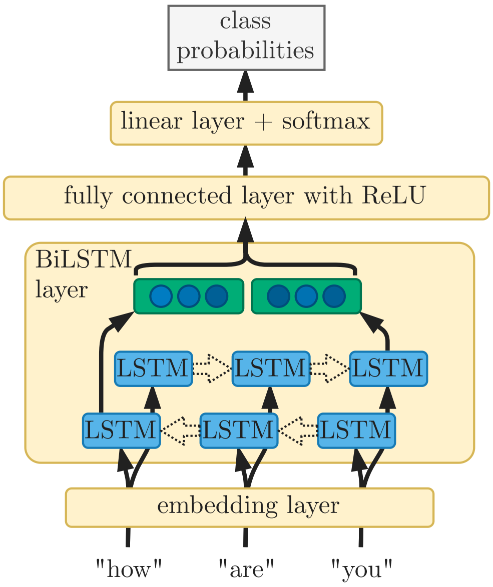


Figure 4.3. The bi-directional LSTM student. Diagram adapted from Figure 1 of Tang et al. (2019b).

The original model was built to process sentences word by word, encoding each word using the pre-trained word2vec embeddings[7] before passing it to the LSTM layer. Words for which there is no embedding (*out-of-vocabulary* (OOV) words) are embedded using a vector initialised with random numbers drawn uniformly from [-0.25, 0.25]. The embedding layer supports three *embedding modes*, based on Kim (2014):

1. **Static**: the embedding parameters are frozen and do not change during training.
2. **Non-static**: the embedding parameters are allowed to be further changed (fine-tuned) during training.
3. **Multichannel**: two embedding instances are used in parallel, one is frozen, the other one is allowed to be fine-tuned. For each input word, the two

[6] Even though Tang et al. tried also other, slightly different layer dimensions, these are the ones that worked the best on CoLA.

[7] The 300-dimensional version trained on Google News, see `code.google.com/archive/p/word2vec/`.

embeddings produced are concatenated together for further processing. The multichannel mode is the one used by Tang et al..

One significant change I made to this model is enabling the use of wordpiece embeddings (instead of word-level ones). This way, the fine-tuned embedding layer from BERT_T can be plugged into the student, giving it some of the teacher's knowledge even before knowledge distillation begins.

In total, this model – from now referred to as LSTM_S – has 2.41 million trainable parameters (excluding the embedding layer), making it 140x smaller than BERT_T.

4.3.4 BERT student model

For the second student, I use a scaled-down version of BERT, matched for size with LSTM_S. In particular, I scaled all the dimensions of BERT_Large down roughly by a factor of 5, ending up with a smaller BERT with $L = 5$ encoder layers, hidden dimension $d_h = 204$, intermediate dimension $d_I = 750$ and $A = 3$ attentional heads – amounting to 2.42 million trainable parameters (embedding parameters excluded). I will refer to this model as BERT_S.

4.3.5 Knowledge distillation

During knowledge distillation, BERT_S was trained using the usual cross-entropy loss. The softmax temperature I fixed at $T = 3$.[8]

LSTM_S uses the mean squared error (MSE) loss, following Tang et al. who compared this with the cross-entropy loss (with $T = 3$) and report that MSE led to slightly better performance. Following preliminary experiments on CoLA, I set the training budget to be 30 epochs for LSTM_S (same as Tang et al.) as it enabled the evaluation-set score to comfortably converge. BERT_S, on the other hand, took longer to converge and I decided for a 60-epoch budget. Importantly, for each training, I observe the evaluation-set score and keep the best-performing parameters. This way, even if the student's performance eventually starts decreasing during training, I retain the best version for any further analysis and comparison.

Originally, both students were initialised randomly from scratch before training, with the exception of the embedding layer of LSTM_S (initalised from word2vec). Later, I explore different ways of initialising the embedding layers in both students.

[8] Usual values are from 1 (no effect) to 3. For instance, Sanh et al. (2019) use $T = 2$. In a work that is much close to my situation, Tsai et al. (2019) apply knowledge distillation from BERT_Base into a 18-million-parameter smaller BERT, observing that from $T = \{1, 2, 3\}$ the best one was $T = 3$.

The learning algorithm is Adam for BERT_S, with $\beta_1 = 0.9$ and $\beta_2 = 0.98$. The learning rate I initially set to 5×10^{-4} and later optimise. Following Tang et al., Adadelta (Zeiler, 2012) is used for LSTM_S, with $\eta = 1.0$ and $\rho = 0.95$. Note that Adam is a generalised version of Adadelta and is much more widely used; later, I explore the use of Adam for LSTM_S. Originally, I do not anneal the learning rate for LSTM_S. For BERT_S, I originally anneal η linearly from 0 over the first 10 epochs, and let it linearly decay to 0 during the remaining 50 epochs. Later, I explore other annealing schedules.

I use the original gradient norm clipping for LSTM_S, with the maximum norm constrained to 30. However, throughout my experiments, I never observe the norm to reach the limit, and I set a limit to the gradients of BERT_S only symbolically, to make sure that the gradients in any case do not explode.

For regularisation, I use the standard dropout rate of 0.1 throughout all training – this rate is used by Devlin et al. (2018) for BERT and also by Tang et al. for LSTM_S.

Tang et al. observe that small batch sizes work well for the BiLSTM student, and I use their recommended value of $B = 50$. For BERT_S, I start with somewhat larger batches: $B = 256$. In any case, I explore various batch sizes later.

While LSTMs can process sequences of any lengths, Transformer models like BERT impose a maximum sequence length for practical reasons, with all sequences within a batch padded to the maximum length. While BERT_T allows sequences of up to 512 wordpieces in length, extremely few sentences reach this length (especially in my case, when the inputs are single sentences, not pairs). Hence, to speed up training, I constrain sentences to be at most 128 token long for BERT_S.

4.3.6 Probing

For the light-weight probing classifier, Conneau et al. (2018) use a simple neural network with one hidden layer with the sigmoid activation function and dropout, followed by a linear layer projecting to the desired number of classes. The cross-entropy training loss is used, with the Adam learning algorithm with $\eta = 0.001$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Training uses minibatches of 64 examples and employs early stopping for when the accuracy does not improve over 5 consecutive iterations.

For consistency with the exact method of Conneau et al., I tune the dropout rate (choosing from [0.0, 0.1, 0.2]) and the hidden layer width (choosing from [50, 100, 200]) using the evaluation set. Each probing score is reported as the one for the best dropout and layer width values.

When probing, an important design decision is how to extract sentence embeddings from the given model's layers. The BiLSTM layer of LSTM_S can produce two hidden states at each timestep (though only the last hidden state in each direction is used in training). Conneau et al. experiment with:

1. Creating a *BiLSTM-max* encoding such that each of its elements is the maximum over the values for each timestep. (The encoding has the same dimensionality as the BiLSTM layer output.)

2. Creating a *BiLSTM-last* encoding by simply taking the last hidden state in each direction – the encoding is the same as the BiLSTM layer output.

Conneau et al. report mixed results with BiLSTM-max encodings leading to better probing scores on some of the probing tasks. I am constrained to using BiLSTM-last since the PyTorch implementation of LSTMs does not give access to intermediate hidden states, only to the last one in each direction.

In BERT$_{\mathrm{S}}$, I can access all of the hidden representations produced by each encoder layer. I try three different ways of combining these into a single encoding:

1. Maximum pooling, equivalent to BiLSTM-max, i.e. taking the maximum value for each element over all hidden representations.

2. Single-position encoding (the equivalent fo BiLSTM-last), i.e. taking the hidden representation that is used in the model for the final classification. While in BiLSTM, the last hidden state in each direction is used for further processing and classification, in BERT, it is the first hidden representation, corresponding to the `[CLS]` input token.

3. Average pooling (not explored by Conneau et al.), which is similar to maximum pooling but instead of maximum takes the average of each element across all representations.

After conducting simple preliminary probing experiments with BERT$_{\mathrm{T}}$on each downstream task, I observed that the differences between the three approaches are mostly inconsistent and not large. However, in many cases, maximum pooling produced worse probing scores than the other two techniques, and average pooling slightly outperformed single-position representation overall. In all further probing experiments with BERT$_{\mathrm{T}}$ and BERT$_{\mathrm{S}}$, I use average pooling. **TO-DO: Should I include detailed results here? This is not a crucial design decision and the results don't show clearly interpretable differences between the three techniques.**

Inspired by Tenney et al. (2019a), I probe various encoder layers across the BERT models in order to see not just how well a certain kind of linguistic knowledge is captured in the model, but also how its concentration varies across the model layers.

4.4 Computing environment and runtimes

All major parts of my experiments – teacher and GPT-2 fine-tuning, augmentation data sampling, teacher logits generation, knowledge distillation and probing – are run in the environment of the University's Teaching cluster[9]

Each job uses its own one GPU – mostly either GeForce GTX TITAN X or GeForce RTX 2080 Ti – with 12-13GB of memory, and additional 30GB of RAM for use with the CPU.

The only processes that were parallelised were the augmentation data sampling and the teacher logit generation. Using 4 parallel processes, each with its own GPU with 6GB of memory, generating the 800,000 augmentation sentences took almost 17 hours for CoLA, 15 hours for SST-2, and 4 hours for Sara. (Following the original Sara training sentences being mostly short, the augmentation sentences are also short, which means they can be sampled faster.) Generating teacher logits in batches of 2048 examples, using four 12GB-memory GPUs, took slightly over 1 hour (measured on CoLA). **TO-DO: I did not note the runtime of fine-tuning GPT-2. Should I re-run the fine-tuning to be able to report these numbers?**

Teacher-finetuning took only 0.5-1h for CoLA and Sara, and 4 hours for SST-2 (due to it having much more trainind examples than the other two tasks). Knowledge distillation runtimes were much higher for BERT$_S$ than for LSTM$_S$ (both in the ~2.4-million-parameter variants). While BERT$_S$ took 15-26 hours to train, LSTM$_S$ took only 5-8 hours. In both cases, the training took the longest on SST-2 and the shortest on CoLA, due to different characteristic lengths of sentences in the augmentation data.

While the cluster was mostly unused by others during my experiments, in some cases, I ran multiple experiments in the same cluster node. This could have impacted the runtimes when the jobs had to share the CPU resources (even though each cluster node had as many as 32 or 20 CPU cores).

Because of the role restricions in the cluster, I could not run more than 20 jobs at the same time. This had a significant impact especially on the time it took to run the hyperparameter exploration experiments. It was also the main reason why I did not – with a few exceptions – run my experiments repeatedly with many different random seeds for more credible results.

[9] computing.help.inf.ed.ac.uk/teaching-cluster.

Chapter 5

Training student models

In this chapter, I use knowledge distillation to train student models BERT$_\mathrm{S}$ and LSTM$_\mathrm{S}$ from the fine-tuned teacher BERT$_\mathrm{T}$ on each of the three downstream tasks. My aim is to end up with students that are small but perform well. Ideally, the student size will stay at the initial 2.4 millions of trainable non-embedding parameters while the evaluation-set performance will be above 90% of the teacher's performance.

As discussed in Section 4.1, my aim is not to find the best possible student hyperparameters, but I still explore some of them to gain an intuition for the reasonable ranges of values and for their behaviour in knowledge distillation. In particular, I find a well-performing configuration of each student on CoLA, looking for good evaluation-set score and fast convergence. Then, on the remaining tasks, I use the same configuration, only tailoring a small number of parameters to the need of the concrete dataset at hand. Most importantly, I adjust the size of each student separately for each task because some tasks are known to be more or less difficult than others, requiring the students to be more or less complex.

After obtaining well-performing students for each task, I compare them with one another and with the respective teacher and discuss my observations.

5.1 General hyperparameter exploration on CoLA

I restrict myself to exploring the following essential hyperparameters (in both students):

1. η – the learning rate. Additionally, for LSTM$_\mathrm{S}$, I compare the original Adadelta learning algorithm with the more general Adam.
2. Learning rate scheduling, more concretely the warmup duration (in epochs) E_w of gradual warmup of the learning rate (Goyal et al., 2017), and the optional use of linear decay following the warmup.
3. B – the minibatch size.

4. Embedding type – word-level vs wordpiece.

Following on the discussion of implementation details in Section 4.3, the initial configurations of the parameters to be explored are:

- For BERT$_S$: Adam with $\eta = 5 \times 10^{-5}$, $B = 256$, $E_w = 10$ – warmup over the first 10 epochs, followed by linear decay of η, and wordpiece embedding layer with the non-static mode.

- For LSTM$_S$: Adadelta with $\eta = 1.0$, $B = 50$, no η warmup or decay, and the word-level embeddings with the multichannel mode.

Note that the embedding mode is explored separately for each task in the next section.

Initially, BERT$_S$ was initialised entirely from scratch, including the embedding layer. This poses a disadvantage for BERT$_S$ because the BiLSTM student starts with embeddings initialised from the pre-trained word2vec. To eliminate this disparity, I initialise BERT$_S$'s embedding layer with the parameters from the fine-tuned BERT$_T$. **TO-DO: explain how linear layer is used to project down to BERT$_S$'s hidden dimension.** Even though this idea is by no means revolutional, to the best of my knowledge, I am the first one to initialise a student in knowledge distillation from Transformers in this way. In exploring the parameters of BERT$_S$, I report results for the variant with the teacher's embedding knowledge. Results for the variant initialised entirely from scratch are in **TO-DO: APPENDIX X**. In general, the difference is not large, but the version initialised from scratch performs slightly worse.

5.1.1 Choosing learning algorithm and learning rate

Because Tang et al. (2019a) report not tuning their BiLSTM hyperparameters, I challenge their choices. In particular, I see no reason to not use the Adam learning algorithm, which is a widely used and more general version of the Adadelta algorithm they use.

For both students, I try a wide range of η values with Adam: 5×10^{-3}, 1.5×10^{-3}, 5×10^{-4}, 1.5×10^{-4}, 5×10^{-5}, 1.5×10^{-5}, 5×10^{-6}.

Fig. 5.1 shows that for all students the ideal η is around 5×10^{-4}. Much larger and much smaller values leading to poor learning, in particular the largest $\eta = 5 \times 10^{-3}$ "kills" the learning of BERT$_S$ entirely due to gradient explosion. As expected, BERT$_S$ initialised from scratch performs worse than when initialised from the wordpiece embeddings of BERT$_T$. However, the differences are not large.

As discussed previously, BERT$_S$ converges much slower than LSTM$_S$, hence the 30-epoch and 60-epoch training budgets for LSTM$_S$ and BERT$_S$, respectively. Additionally, it is apparent that LSTM$_S$ performs significantly better than BERT$_S$ even though the sizes of the models are comparable.

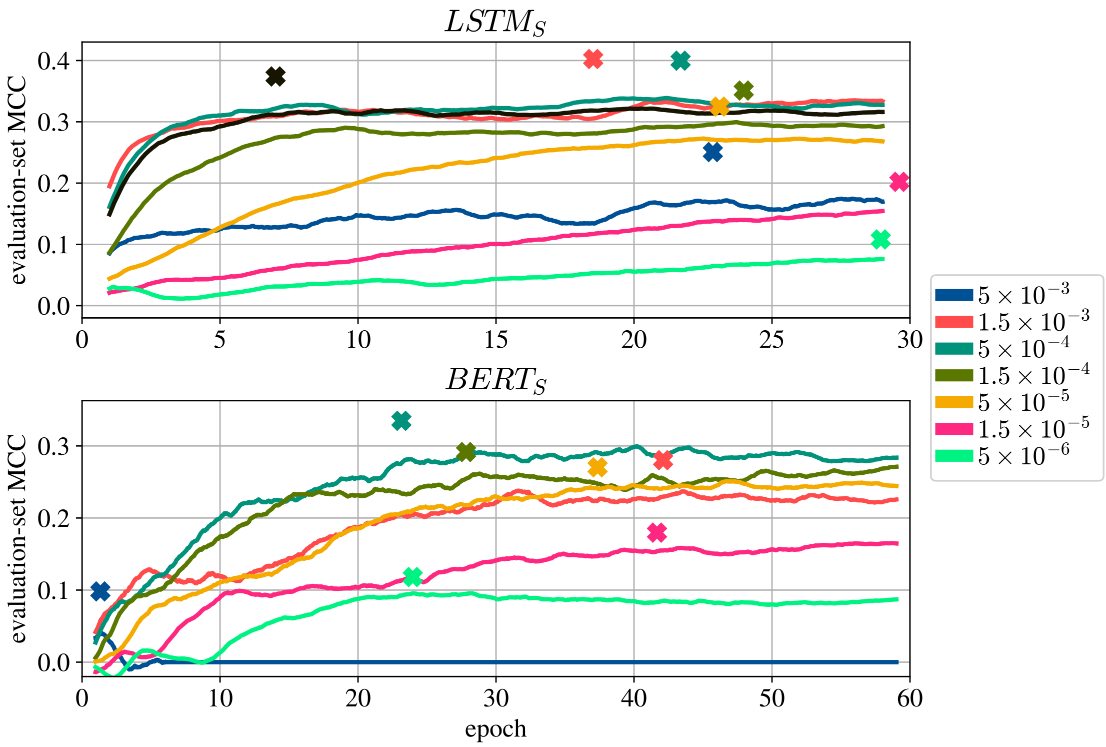


Figure 5.1. Comparing various η values on CoLA. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

In the case of LSTM_S, the Adadelta algorithm is outperformed by Adam. From now onwards, I use Adam with both students, in all cases with $\eta = 5 \times 10^{-4}$.

5.1.2 Choosing learning rate scheduling and batch size

TO-DO: Should I justify the order in which I'm exploring the parameters? η coming first, followed by its scheduling, then the batch size...

Tang et al. (2019b) use no learning rate scheduling, but they report small batch sizes ($B = 50$) to work better than the usual, larger batches. Hence, for LSTM_S I first verify their claims and subsequently move on to η scheduling. For BERT_S, inspired by Sanh et al. (2019) who take advantage of scheduling (both in terms of warmup and decay), I explore η scheduling first (as a continuation from exploring η values), and then look at various B values.

As Fig. 5.2 shows, LSTM_S clearly does prefer small batch sizes. However, training with tiny minibatches takes very long – compare ~25min for $B = 512$ with ~7h for $B = 32$. Hence, I restrain from trying even smaller batch sizes and use $B = 32$ for LSTM_S in all further experiments.

Fig. 5.3 shows the results of exploring various warmup durations for both students. Note that for LSTM_S, whose training budget is only 30 epochs, I did not try the long warmup duration of 20 epochs, only up to 15 epochs.

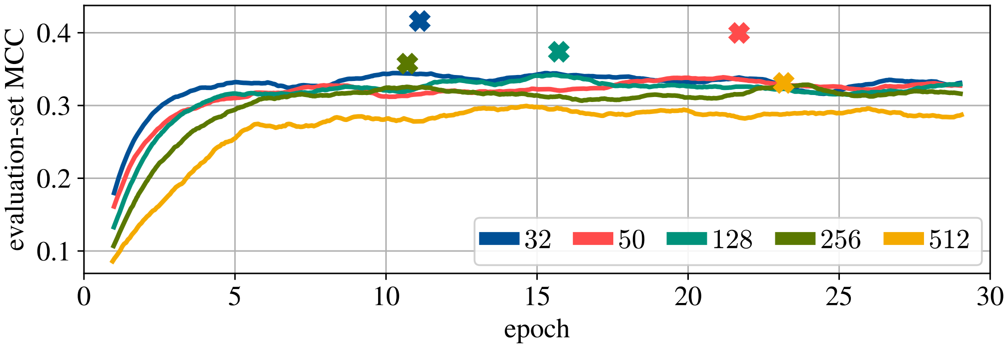


Figure 5.2. Comparing various batch sizes for LSTM$_S$ on CoLA. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

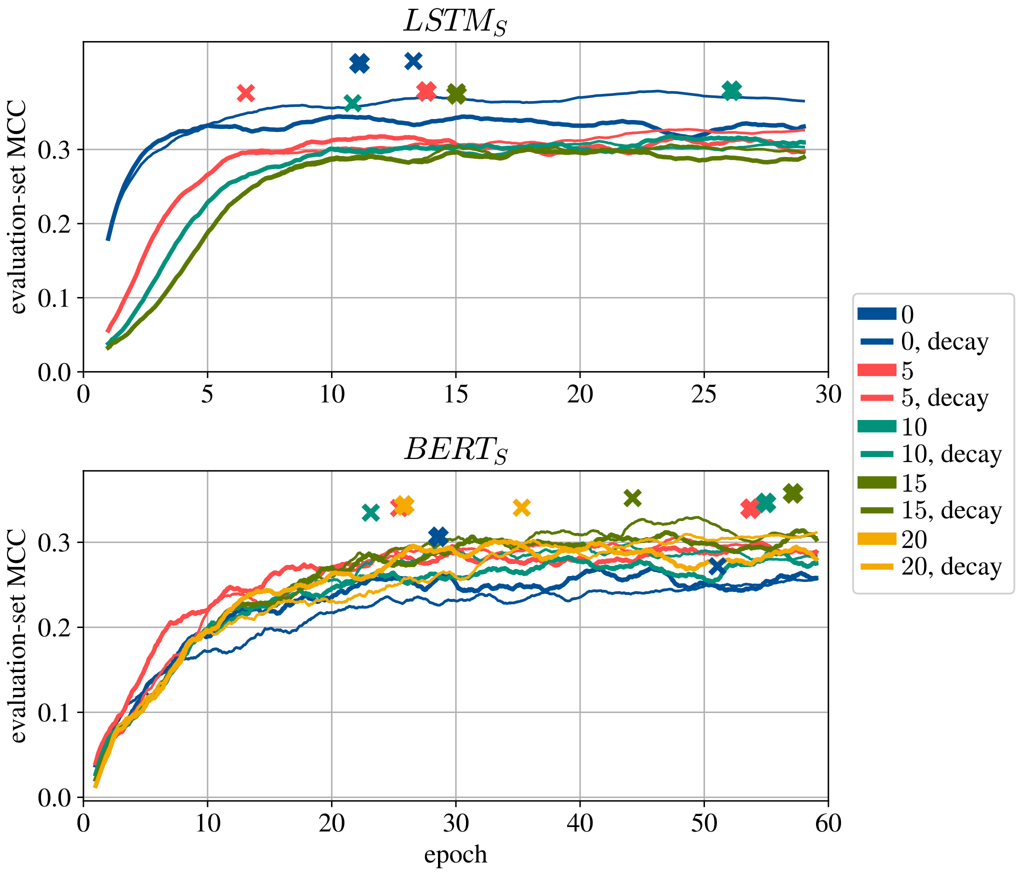


Figure 5.3. Comparing warmup durations E_w on CoLA, with the optional learning rate decay to 0 over the remaining training epochs. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

In the case of LSTM$_S$, Fig. 5.3 shows that the longer the warmup duration, the slower the model converges. This is understandable because during warmup, learning happens less aggressively – and hence more slowly – due to the smaller learning rate. More importantly, the graph shows that η decay does not significantly affect training, but it can help to prevent the model from overfitting the training data. This is most visible for $E_w = 0$ where LSTM$_S$'s performance starts to slowly decrease after 20 epochs in the absence of η decay. All in all, using the full η from the beginning of training is the best option, and η decay can only improve things. $E_w = 0$ with decay is used for LSTM$_S$ in all further experiments.

In the case of BERT$_S$, the only clear result visible from Fig. 5.3 is that BERT$_S$ performs poorly without η warmup. For non-zero warmup durations, there are no significant differences in the best-performance points (marked by crosses) or in the convergence speed. In all further experiments with BERT$_S$, I use $E_w = 15$ and η decay – the configuration which shows the highest stable performance level in Fig. 5.3 in later epochs (beyond epoch 35).

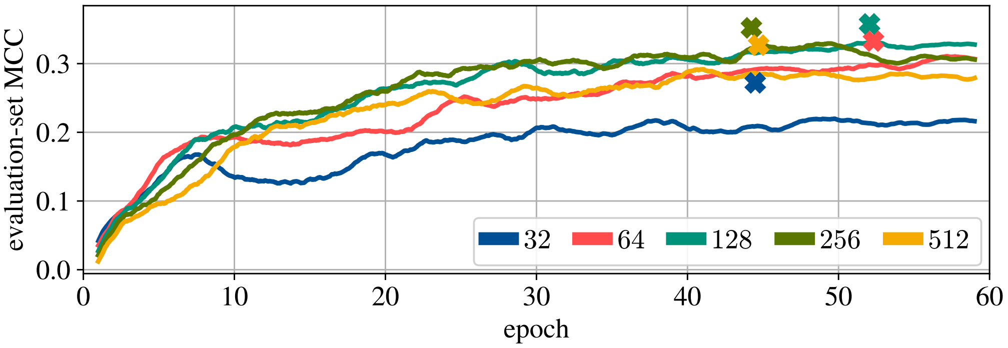


Figure 5.4. Comparing various batch sizes for BERT$_S$ on CoLA. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

The batch size exploration in Fig. 5.4 shows that BERT$_S$ performs best with mid-sized batches of 128-256 examples. Too large batch size ($B = 512$) as well as very small batches of 32-64 make BERT$_S$ underperform (with tiny batches of 32 being particularly detrimental). In all further experiments, $B = 128$ is used with BERT$_S$.

5.2 Optimising students for each downstream task

In this section, I explore different ways of initialising student models with language knowledge, and the effect of model size, separately for each downstream task.

5.2.1 Choosing embedding type and mode

In the previous section, LSTM$_S$ consistently outperformed BERT$_S$. This can be due to different model architectures or due to different embedding types and modes – while LSTM$_S$ has its word-level embedding layer initialised using word2vec and uses the multichannel mode, BERT_S uses wordpiece-level embeddings from the fine-tuned BERT_T in the non-static mode.

I explore combinations of the two embedding types and of different embedding modes. Following preliminary experiments, I do not use the static (embedding freezing) mode as it has detrimental effects on student learning. Left for exploration are the two modes that do allow embedding parameters to be tuned during training: the multichannel and non-static modes.

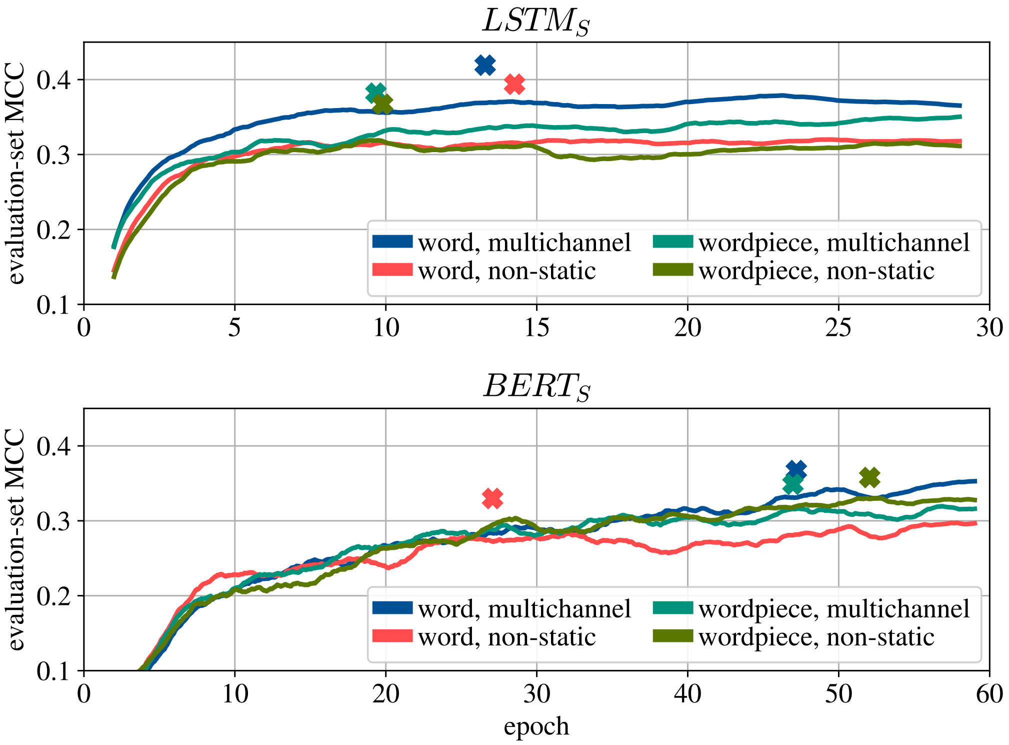


Figure 5.5. Comparing embedding types and modes on CoLA. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

Fig. 5.5 shows how different type and mode combinations affect knowledge distillation on CoLA. With LSTM_S, the multichannel mode is preferred to non-static and word2vec embeddings are preferred to wordpieces; hence, I use the multichannel mode with word2vec in further experiments (same as Tang et al. (2019a)). For BERT_S, the differences are smaller, yet it is clear that word-level embeddings benefit from using the frozen and unfrozen versions provided by the multichannel mode. In further experiments, the multichannel mode combined with word-level word2vec embeddings is used.

For SST-2, I only compare the best word-level and the best wordpiece-level combination for each model as observed from Fig. 5.5. The results are shown in Fig. 5.6. Notice the scale of the y axis: In particular, the students perform roughly the same (unlike on CoLA) and any relative differences observed in Fig. 5.6 are much smaller than the differences observed on CoLA in Fig. 5.5. Hence, I refrain from making conclusions about which embedding type and mode works better; I merely choose to use the word-level embeddings with the multichannel mode in all further experiments on SST-2 (my decision is based on the best evaluation scores marked by crosses in Fig. 5.6).

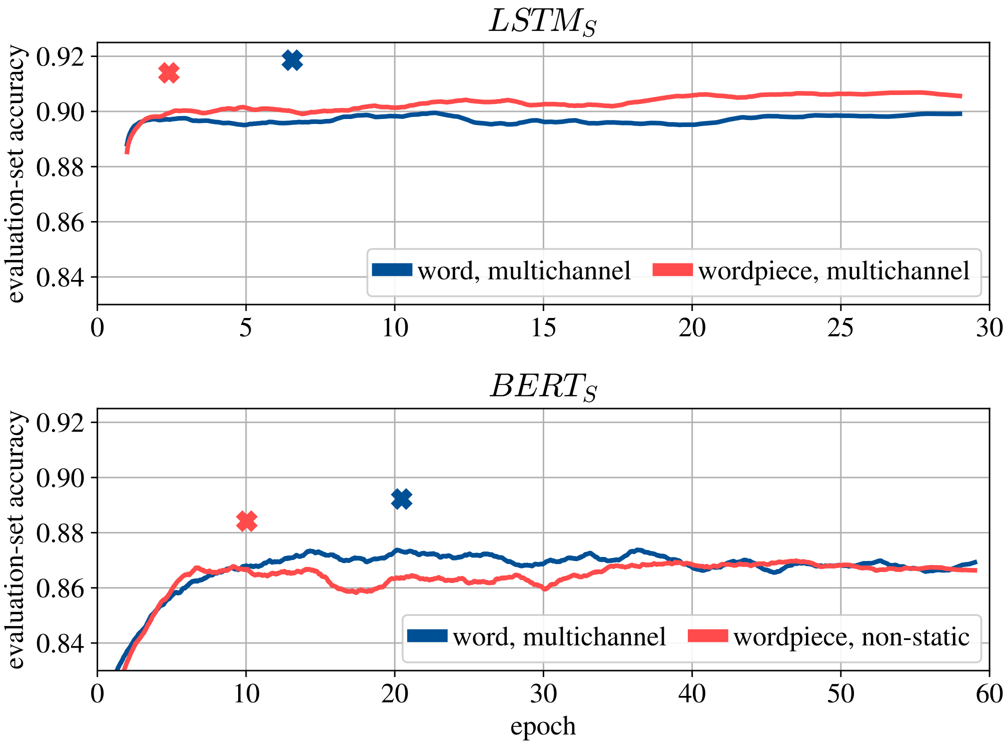


Figure 5.6. Comparing embedding types and modes on SST-2. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

Results on Sara are shown in Fig. 5.7. Here again, the relative differences in performance are small, but using wordpieces helps both students converge faster and reach slightly better performance levels. This can be a result of the word2vec vocabulary not capturing well the conversational language in Sara examples, for instance the utterance "yesyesyes" would be treated simply as one out-of-vocabulary word, whereas wordpieces have the potential to encode it as the word "yes" repeated 3x. In all further experiments on Sara I use the wordpiece embeddings, using the multichannel mode in LSTM_S and the non-static mode in BERT_S.

All in all, the multichannel mode seems to be generally superior to the non-static mode which lacks the frozen version of embeddings. Initialisation from the word-level word2vec embeddings also works better than wordpieces where examples tend to contain legitimate English (CoLA and SST-2). As for the performance gap between LSTM_S and BERT_S, I conclude that it cannot be explained by differences in embedding type/mode, and is most likely a consequence of different student architectures.

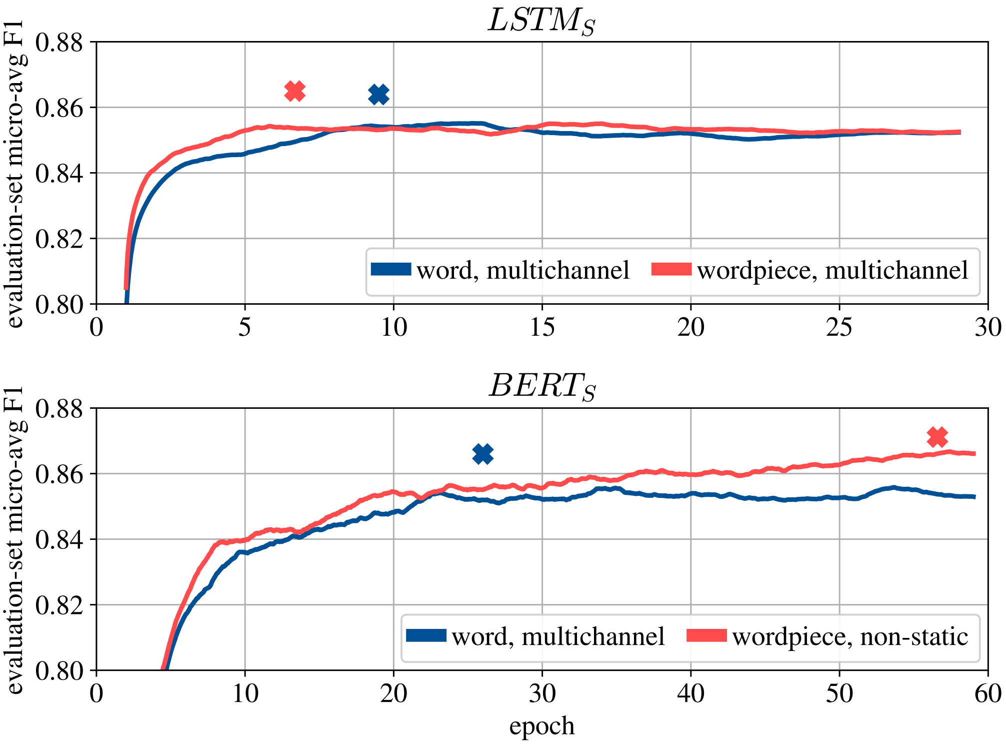


Figure 5.7. Comparing embedding types and modes on Sara. Crosses mark the maximum scores. The lines are smoothed using sliding average with a 2-epochs-wide window.

While wordpiece embeddings have the advantage of being fine-tuned on the particular downstream task (as part of teacher fine-tuning), word2vec contains more general knowledge stored at the word level. Importantly, the wordpiece vocabulary contains the most frequent words in their entirety; only the less frequent ones are split into pieces. Thus, for frequent words, their word2vec and wordpiece embeddings will differ only in the way they are trained. Naturally, since the wordpiece vocabulary has only 30,522 tokens while word2vec has 3,000,000, there are many words covered by word2vec for which the wordpiece embeddings have to be assembled from multiple pieces. On these words – frequent enough to be in the word2vec vocabulary, but not the most frequent ones – word2vec could have an advantage. Once we move beyond the words covered by word2vec to rare tokens like “yesyesyes”, wordpieces become the preferred approach. Clearly, no one approach is generally the best, and the decision should ideally be made individually for each downstream task.

5.2.2 Choosing student size

As the last parameter, I explore the size of each student. In particular, I try to reduce the performance gap on CoLA between BERT_T and both students by making the students larger. On SST-2 and Sara, the 2.4-million-parameter students already achieve scores very close to those of the teacher, and I refrain from exploring larger students – instead, I explore smaller student sizes.

There are two main ways of increasing the student size: By increasing the model “width” and the “depth”. By width, I mean the dimensionality of the model’s layers and internal representations. Depth means adding more layers. While larger width allows the models to extract and maintain richer token representations, adding layers adds more steps to the models’ processing pipelines, allowing for more abstract and task-specific representations to be extracted in the end.

In BERT$_S$, I manipulate width by increasing by a set factor W the hidden dimensionality d_h, the intermediate dimensionality d_I, and the number of self-attentional heads A. Depth is manipulated by changing the number of encoder layers L by the factor D. In LSTM$_S$, I change model width by scaling by W the LSTM layer width d_{LSTM} and the fully-connected layer width d_{FC}, which are originally set to 300 and 400, respectively. Depth is changed by increasing the number of LSTM layers (originally just one) by the factor D. The concrete dimensions of up- and down-scaled students are shown in Tab. 5.1 (BERT$_S$) and in Tab. 5.2 (LSTM$_S$).

W	d_h	d_I	A
1/16	13	47	1
1/8	26	94	1
1/4	51	188	1
1/3	68	250	1
1/2	102	375	2
1	204	750	3
2	408	1500	6
3	612	2250	9
4	816	3000	12

D	L
1/4	1
1/3	2
1/2	3
1	5
2	10
3	15

Table 5.1. BERT$_S$ dimensions for different width scalings (left) and depth scalings (right). The default size with 2.4 million parameters corresponds to $W = 1$, $D = 1$.

5.2.2.1 CoLA

With the teacher’s evaluation-set MCC of 0.599 being much higher than the student performance observed so far (around 0.4), I up-scale both students, aiming for 90% of the teacher performance while keeping the student size smaller than the 340-million-parameter teacher.

As observed in preliminary experiments with large BERT$_S$ versions, their learning suffers from gradient explosion due to the learning rate being too large for the models. For an example, see Fig. 5.8 where the gradient explosion happens around epoch 7 and the model score (MCC) then falls to 0 and stays there.

Without further extensive exploration of optimal learning rate values for each

W	d_{LSTM}	d_{FC}
1/32	9	13
1/16	19	25
1/8	37	50
1/4	75	100
1/2	150	200
1	300	400
2	600	800
3	900	1200
4	1200	1600
5	1500	2000

D	L
1	1
2	2
3	3
4	4
5	5

Table 5.2. LSTM$_S$ dimensions for different width scalings (left) and depth scalings (right). The default size with 2.4 million parameters corresponds to $W = 1$, $D = 1$.

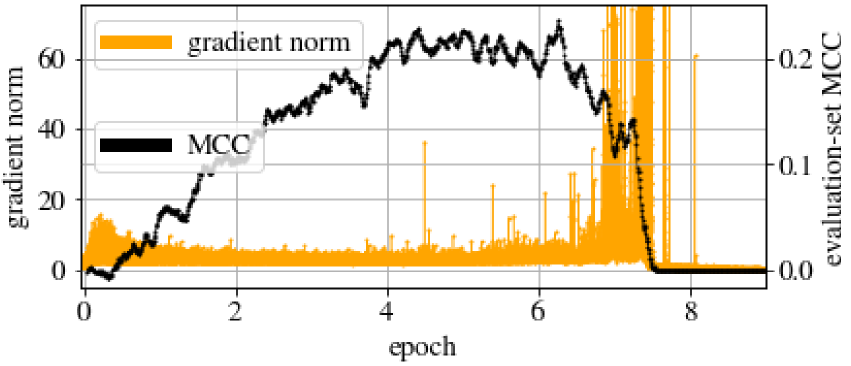


Figure 5.8. Gradient explosion in BERT$_S$ with $W = 2$ and $D = 2$. The MCC values have been smoothed with a 0.1-epoch-wide sliding average window.

BERT$_S$ size[1], I choose better learning rate values manually. Because of the use of η warmup, I can monitor the learning progress for varying η values in the early training epochs, as shown in Fig. 5.10. I approximately identify the point in training beyond which the learning slows down (and later falls to 0) due to large gradients. This way, I approximately identify the largest learning rate that still leads to learning, not to gradient explosion. In the concrete example in Fig. 5.10, I choose the beginning of epoch 3, at which point the learning rate is approximately $\eta = 8 \times 10^{-5}$, and use this learning rate with this concrete model size.

With the manually estimated learning rates, none of the larger versions of BERT$_S$ experiences gradient explosion. The LSTM students all use the same learning rate as this does not lead to any issues. Fig. 5.11 presents the results. While some larger students outperform the original, 2.4-million-parameter ones, the trends

[1] This would be extremely time-consuming because the larger versions were taking over 3 days to train.

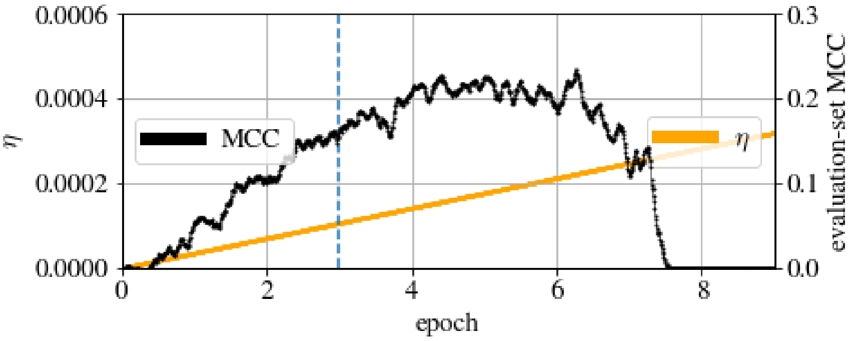


Figure 5.9. Learning progress (MCC over time) vs η for BERT$_\text{S}$ with $W = 2$ and $D = 2$ – the model experiencing gradient explosion in Fig. 5.8. The dashed line shows the approximate position I identify as the approximate latest point of training before the learning starts to slow down. The MCC values have been smoothed with a 0.1-epoch-wide sliding average window.

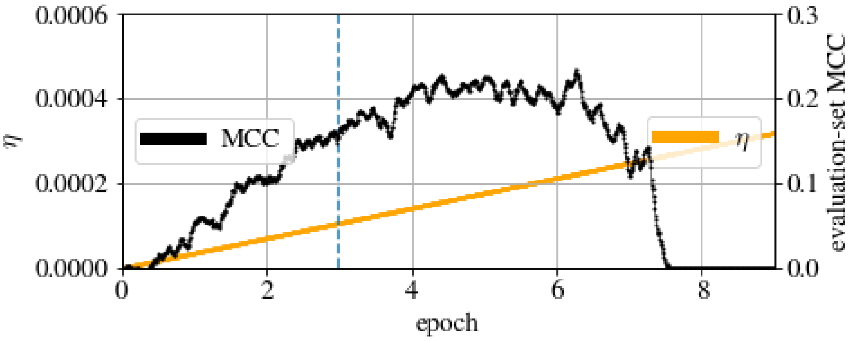
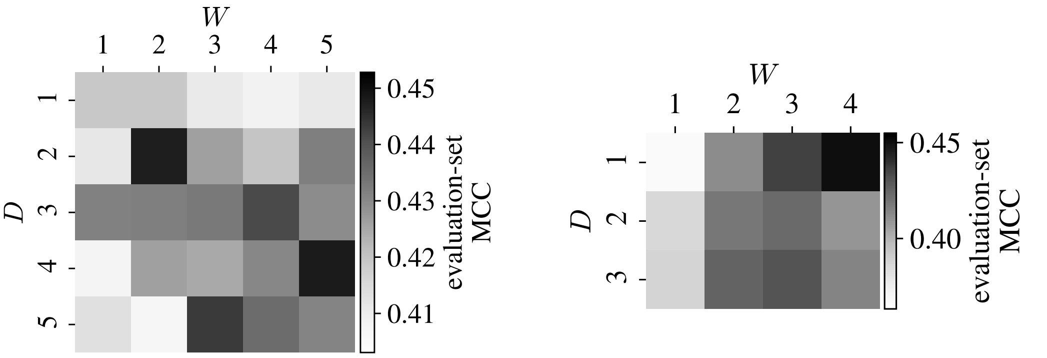


Figure 5.10. Learning progress (MCC over time) vs η for BERT$_\text{S}$ with $W = 2$ and $D = 2$ – the model experiencing gradient explosion in Fig. 5.8. The dashed line shows the approximate position I identify as the approximate latest point of training before the learning starts to slow down. The MCC values have been smoothed with a 0.1-epoch-wide sliding average window.

are not consistent. For LSTM$_\text{S}$ in particular, there is no clear correlation between student width or depth and the performance. For BERT$_\text{S}$, which starts as a relatively deep model with 5 layers, making it wider rather than deeper is helpful. For LSTM$_\text{S}$ which originally has only 1 hidden LSTM layer, increasing both the width and the depth can lead to better-performing models. Overall, both student architectures reach the best evaluation score of ~0.45, far from the teacher performance level of 0.599.

I refrain from exploring students even larger as the biggest students are already approching the teacher size: LSTM$_\text{S}$ with $W = 5$, $D = 5$ has ~247 million parameters and takes over 60h to train, and BERT$_\text{S}$ with $W = 4$, $D = 3$ has ~114 million parameters and takes over 6 days to train. As the best-performing students, I establish BERT$_\text{S}$ with $W = 4$, $D = 1$, and LSTM$_\text{S}$ with $W = 2$, $D = 2$.

Model size has impact on other model hyperparameters – in particular, larger models typically require smaller learning rates. In line with the hyperparameter



(b) Different sizes of BERT$_S$.

(a) Different sizes of LSTM$_S$.

Figure 5.11. Best evaluation-set performance for the different student sizes on CoLA.

exploration so far, I ignore these interactions.

mention exploding gradients

give concrete numbers (teacher vs students so far)

say something about Cola being difficult (students lag behind teacher, even huge students) vs other tasks being easy

5.3 Choosing task-specific parameters

5.3.1 CoLA

I tried expanding $LSTM_S$ by increasing the number of LSTM layers from 1 to [2, 3, 4, 5] and the width of the LSTM and fully-connected layers from the original 300 and 400 by 2x, 3x, 4x and 5x. In case of $BERT_S$, I tried increasing the number of layers from the original 5 to 10 and 15, and the other parameters (# of attentional heads, hidden and intermediate size) by 2x, 3x and 4x.

TO-DO: add graphs

For $LSTM_S$, the results are indecisive, without clear correlation between student size and performance. The best student was 2x deeper and 2x wider than the initial configuration.

For $BERT_S$, I soon observed that bigger models started dying after the first few epochs. TO-DO: add diagnostic graphs. This dying surfaced as the dev-set score flattening out and then decreasing towards 0, at which point the gradients in the model would explode. I identified the cause of this to be too big learning rate (thanks to the warm-up, the model was experiencing η starting from 0 and increasing towards the target value). Hence, by comparing the graphs for the linearly increasing η and the dev-set performance, I tried to manually identify η values as big as possible that were still enabling the model to improve fast (without making it plateau or die). These values were in the range from the originally chosen 5e-4 down to 8e-5 (generally lower η for bigger models). After this modification, all students trained normally, without dying. TO-DO: add graphs and conclusions once all of these have finished training.

5.3.2 SST-2

TO-DO: run experiments and report

5.3.3 Sara

TO-DO: run experiments and report recap parameters of best students on each dataset, the eval scores compare performance with teacher, comment comment on preference for word-level vs wordpiece embeddings (especially Sara and wordpieces) comment on multichannel being kinda like a residual connection on top of non-static would be interesting to see word-level embeddings fine-tuned in teacher – could be even better!

Chapter 6

Analysing the models

6.1 Analysing the models' predictions

This will involve for each downstream task:

1. gathering dev-set predictions made by the two best students and the teacher
2. looking at the mistakes made by each model (most confident mistakes and least confident mistakes first, maybe taking first 10-20 mistakes from each end), trying to describe then in intuitive terms if possible
3. comparing the mistakes made by different models, again trying to describe what I observe
4. looking at how confident the predictions were in general for each model (maybe the distribution of confidences?) and if possible, comparing this across models

My aim is to make at least some observations describable in human-understandable terms about how the models compare: how the 2 architecturally different students compare, and how their predicting behaviour compares to that of their teacher. Ideally, I will be also able to form some (even if weak) hypotheses that are further testable using probing tasks.

6.2 Probing the models for linguistic knowledge

Here, I will for each downstream task:

- probe each of the 3 models in different layers (e.g. the 24-layer $BERT_T$ in layers 1, 6, 12, 18, 24) to see how much linguistic knowledge and where is present in each model
- probe also the pre-trained BERT (not finetund on any downstream task) and look at how knowledge is preserved/thrown away when the teacher gets

finetuned vs when students with little prior knowledge are trained during distillation

- compare how knowledge is present in the two architecturally different students, try to relate this to other differences noticed when analysing the students' predictions

For now, I have one graph summarising the results of probing just the teacher models, see Fig. 6.1.

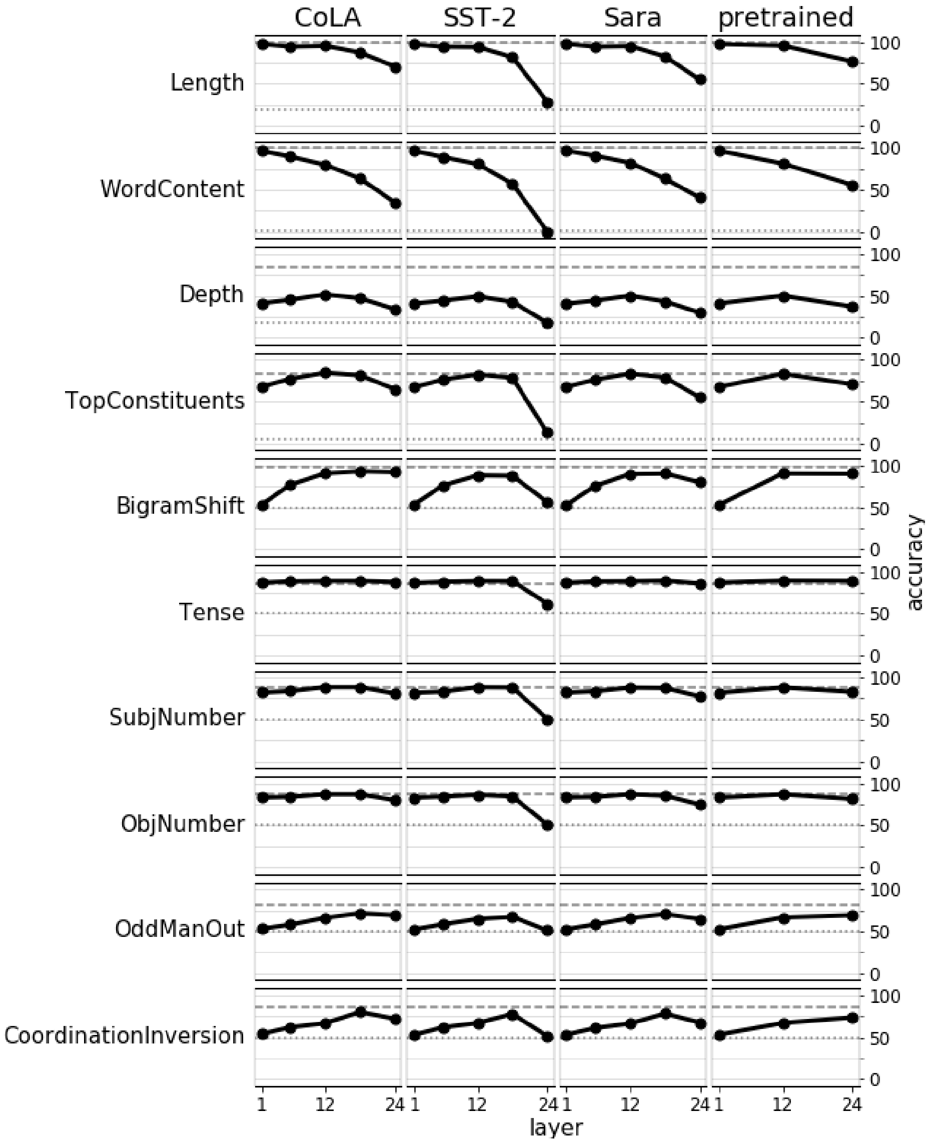


Figure 6.1. Probing the 3 finetuned teachers and the pre-trained one. The dotted line shows baseline accuracy levels achieved by majority-class guessing, the dashed line shows human performance.

Chapter 7

Overall discussion and conclusions

Chapter 8

Future work

Bibliography

Adhikari, A., Ram, A., Tang, R., and Lin, J. (2019). DocBERT: BERT for document classification.

Adi, Y., Kermany, E., Belinkov, Y., Lavi, O., and Goldberg, Y. (2017). Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. *ICLR*, abs/1608.04207.

Ba, J. and Caruana, R. (2014). Do deep nets really need to be deep? In *NIPS*.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate.

Belinkov, Y. and Glass, J. R. (2018). Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72.

Bucila, C., Caruana, R., and Niculescu-Mizil, A. (2006). Model compression. In *KDD '06*.

Cheong, R. and Daniel, R. (2019). transformers.zip: Compressing transformers with pruning and quantization.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML 2008, page 160âĂŞ167, New York, NY, USA. Association for Computing Machinery.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12(null):2493âĂŞ2537.

Conneau, A. and Kiela, D. (2018). SentEval: An evaluation toolkit for universal sentence representations. *arXiv*, abs/1803.05449.

Conneau, A., Kruszewski, G., Lample, G., Barrault, L., and Baroni, M. (2018). What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *ACL*.

Dai, A. M. and Le, Q. V. (2015). Semi-supervised sequence learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Ad-*

vances in Neural Information Processing Systems 28, pages 3079–3087. Curran Associates, Inc.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-XL: Attentive language models beyond a fixed-length context. In *ACL*.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding.

Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., and Kagal, L. (2018). Explaining explanations: An overview of interpretability of machine learning.

Goyal, P., DollÃąr, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch SGD: Training ImageNet in 1 hour.

Graves, A. (2013). Generating sequences with recurrent neural networks.

Hinton, G. E., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735âĂŞ1780.

Huang, Z. and Wang, N. (2017). Like what you like: Knowledge distill via neuron selectivity transfer. *arXiv*, abs/1707.01219.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. (2019). TinyBERT: Distilling BERT for natural language understanding.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1700–1709.

Kim, Y. (2014). Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar. Association for Computational Linguistics.

Kim, Y. and Rush, A. M. (2016). Sequence-level knowledge distillation. *arXiv*, abs/1606.07947.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *NIPS*.

Lample, G. and Conneau, A. (2019). Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*.

Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C. H., and Kang, J. (2019). BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*.

Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization. In *ICLR*.

Luong, T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Matthews, B. W. (1975). Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451.

Michel, P., Levy, O., and Neubig, G. (2019). Are sixteen heads really better than one?

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In Bengio, Y. and LeCun, Y., editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Mikolov, T., Karafiát, M., Burget, L., Černocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH*.

Mirzadeh, S., Farajtabar, M., Li, A., and Ghasemzadeh, H. (2019). Improved knowledge distillation via teacher assistant: Bridging the gap between student and teacher. *CoRR*, abs/1902.03393.

Mukherjee, S. and Awadallah, A. H. (2019). Distilling transformers into simple neural networks with unlabeled transfer data.

Papamakarios, G. (2015). Distilling model knowledge (MSc thesis). *arXiv*, abs/1510.02437v1.

Pires, T., Schlinger, E., and Garrette, D. (2019). How multilingual is multilingual BERT?

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9.

Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., and Bengio, Y. (2015). FitNets: Hints for thin deep nets. In *In Proceedings of ICLR*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). *Learning Internal Representations by Error Propagation*, page 318âĂŞ362. MIT Press, Cambridge, MA, USA.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *NeurIPS EMC$\hat{2}$ Workshop*.

Sau, B. B. and Balasubramanian, V. N. (2016). Deep model compression: Distilling knowledge from noisy teachers. *arXiv*, abs/1610.09650.

Shi, X., Padhi, I., and Knight, K. (2016). Does string-based neural MT learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1526–1534, Austin, Texas. Association for Computational Linguistics.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*.

Strobelt, H., Gehrmann, S., Behrisch, M., Perer, A., Pfister, H., and Rush, A. M. (2018). Seq2seq-vis: A visual debugging tool for sequence-to-sequence models.

Sucik, S. (2019). Pruning BERT to accelerate inference. https://blog.rasa.com/pruning-bert-to-accelerate-inference/.

Sun, C., Myers, A., Vondrick, C., Murphy, K., and Schmid, C. (2019). VideoBERT: A joint model for video and language representation learning.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Tang, R., Lu, Y., and Lin, J. (2019a). Natural language generation for effective knowledge distillation. In *Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019)*, pages 202–208, Hong Kong, China. Association for Computational Linguistics.

Tang, R., Lu, Y., Liu, L., Mou, L., Vechtomova, O., and Lin, J. (2019b). Distilling task-specific knowledge from BERT into simple neural networks. *arXiv*, abs/1903.12136.

Tenney, I., Das, D., and Pavlick, E. (2019a). BERT rediscovers the classical NLP pipeline. In *ACL*.

Tenney, I., Xia, P., Chen, B., Wang, A., Poliak, A., McCoy, R. T., Kim, N., Van Durme, B., Bowman, S., Das, D., et al. (2019b). What do you learn from context? probing for sentence structure in contextualized word representations. In *7th International Conference on Learning Representations, ICLR 2019*.

Tsai, H., Riesa, J., Johnson, M., Arivazhagan, N., Li, X., and Archer, A. (2019). Small and practical BERT models for sequence labeling. In *EMNLP/IJCNLP*.

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

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. (2018). GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium. Association for Computational Linguistics.

Warstadt, A., Singh, A., and Bowman, S. R. (2018). Neural network acceptability judgments. *arXiv*, abs/1805.12471.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., and Brew, J. (2019). HuggingFace's Transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771.

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

Yu, S., Kulkarni, N., Lee, H., and Kim, J. (2018). On-device neural language model based word prediction. In *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*, pages 128–131, Santa Fe, New Mexico. Association for Computational Linguistics.

Zeiler, M. D. (2012). ADADELTA: An adaptive learning rate method.

Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks.