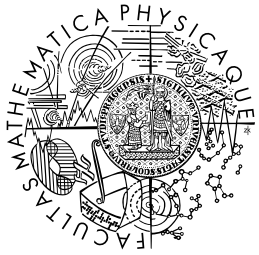

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

David Burian

Document embedding using Transformers

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: Jindřich, Libovický Mgr. Ph.D.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Document embedding using Transformers

Author: David Burian

Institute: Institute of Formal and Applied Linguistics

Supervisor: Jindřich, Libovický Mgr. Ph.D., Institute of Formal and Applied Linguistics

Abstract: Abstract.

Keywords: text embedding document embedding transformers document classification document similarity

Contents

Introduction

- what we are aiming to do
- why is it important/useful — where are embeddings used
- long documents — why?
- transformers — why?

1. Document representation

Representing a piece of text by a dense vector, also known as a text embedding, is an ubiquitous concept in Natural Language Processing (*NLP*). Embedding long continuous pieces of text such as documents is, however, substantially more difficult than embedding words or sentences, as the longer and more complex input still needs to be compressed into similarly sized vector. In this chapter we first explore the use of document embeddings in various scenarios. In this context, we then describe the qualities of document embeddings that we deem as beneficial and which create the motivation behind our training method proposed in Chapter 3.

1.1 On the usefulness of document embeddings

Embeddings that capture the semantics of the document in a low-dimensional vector reduce the noise in the raw input while making the subsequent operations more efficient. These are the main reasons why document embeddings are widely used across different tasks such as classification [Cohan et al., 2020, Neelakantan et al., 2022, Izacard et al., 2021, Ostendorff et al., 2022], ad-hoc search [Singh et al., 2022, Zamani et al., 2018], query answering [Neelakantan et al., 2022], visualization [Cohan et al., 2020, Dai et al., 2015] and regression [Singh et al., 2022].

While some models [Singh et al., 2022] can generate different embeddings for a single input depending on the task, most embedding models output a single vector that is effective across many types of tasks [Neelakantan et al., 2022, Cohan et al., 2020, Ostendorff et al., 2022]. This shows that embedding models can substitute several dedicated models, severely saving on time and resources.

1.2 Desirable qualities of embeddings

As is clear from the previous section, the usefulness of document embeddings comes from 3 properties. An ideal document embedding (1) represents the document's text faithfully (2) with a single vector (3) of low dimension.

In this section we focus on faithful document representation, which we view as a composition of two abstract qualities: *structural* and *contextual*. Document embedding with structural quality (or structural document embedding) faithfully models the relationships between word or word sequences. On the other hand, contextual document embedding accurately composes the meaning of all processed words, capturing their overall theme or topic. We view these qualities as scales, and so a document embedding may have high structural quality, but low contextual quality. Such embedding would capture the relationships between words very well, but would fail to assign correct meaning to these relationships due to their ambiguous meanings caused by lacking context.

Since each document embedding is produced by a model, we may attribute similar qualities to the models themselves. In this sense we speak of *structural or contextual capacity* of the model.

In the following subsections we focus on each quality separately, describing each in more detail. Then at the end of this section we compare the two qualities and setup the basic intuition behind our proposed finetuning method described in Chapter 3.

1.2.1 Structural quality of document embeddings

Structural quality defines how well the embedding captures relationships within the input text. The more complex the relationship is the higher is the structural quality of the embedding that captures it. For instance, for an input "Fabian likes playing the guitar, but Rebecca does not.", an embedding with high structural quality would understand that

1. "Fabian" likes something based on words "Fabian likes" ,
2. a guitar can be played based on words "playing the guitar" ,
3. the two sequences of words separated by comma are in a opposition based on words ", but", and
4. "Fabian" likes to play the guitar based on observations 1, and 2 .

Embedding models that compare input words and sequences of input words can produce document embedding with high structural quality. We say that such models have high structural capacity. A good example of model with high structural capacity is Transformer [Vaswani et al., 2017]. Transformer's self-attention layer allows each word to exchange information with other words. Self-attention thus allows not only comparisons of words, but also aggregation of several words into one. Thanks to Transformer's layered architecture, such aggregations can be compared in the same manner on higher levels. An example of a model with low structural capacity is Paragraph Vector [Le and Mikolov, 2014]. Paragraph Vector compares words only in a single fully connected layer. Such architecture prevents understanding of more complex relationships that build on another relationships such as observation 4 in the example above. Additionally, Paragraph Vector ignores position information, so it cannot know how were the words originally arranged.

1.2.2 Contextual quality of document embeddings

Contextual quality of a document embedding defines how well the embedding captures the overall meaning of larger sequences of text. The higher the contextual quality of an embedding is, the longer is the sequence whose meaning the embedding correctly captures as a whole. For instance, let us consider two documents: 1. a description of typical commercial turbo-jet airplane and 2. a recipe for spicy fried chicken wings. A document embedding with high enough contextual quality would reflect that the meaning of a sentence "Left wing is too hot." dramatically differs between the two documents and would accordingly adjust the sentence's contribution to the resulting document embedding.

Provided the document's text is cohesive and continuous, capturing its overall meaning gets easier as the text's length increases. This is intuitive since, if

we consider smaller pieces of text, meaning of some parts may be ambiguous or misinterpreted. However, as the length of the considered sequence grows, the probability of misinterpreting its meaning shrinks as we have more words, whose common theme we are looking for. Consequently, models that are able to process longer pieces of text have higher contextual capacity. Typical example of a model with good contextual capacity is Paragraph Vector [Le and Mikolov, 2014], which can process, in theory, indefinitely long sequences[1]. Additionally, Paragraph Vector stores a single vector per document which is iteratively compared to all words within that document, which gives the model the opportunity to adjust the contribution of individual words to the overall meaning of the document. On the other hand, Transformer [Vaswani et al., 2017] has much smaller contextual capacity as its memory requirements grow quadratically with the length of the input. In practice this prohibits Transformer from processing longer sequences of text.

1.2.3 Combining structural and contextual qualities

Each quality describes different aspect of faithful representations. Structural quality is focused more on local relationships of words, while contextual quality considers mainly the global picture of the document. From another point of view, structural quality is oriented more towards precision, while contextual quality is oriented more towards recall. In a way the two qualities complement each other. Contextual quality brings in the overall document theme, while the structural quality brings in the detailed meaning of a shorter sequence. These two pieces of information can be then aligned to produce precise and unambiguous document embedding. And so we hypothesize that the combination of these qualities is more beneficial for document embeddings than the same amount of either quality alone.

While we predict that mix of these qualities is important, we are unsure which ratio of the qualities would be the most performant. Arguably, structural quality is more important than contextual, since in extreme cases it can model relationships so complex they span the entire document and thereby substituting the role of contextual quality. On the other hand, the number of total relationships found in a document grows exponentially with the length of the document, while the number of topics covered can grow only linearly. Consequently, we can expect that for a given maximum input length an embedding model with contextual capacity to be much smaller than an embedding model with structural capacity.

To find the optimal mix between structural and contextual quality, in Chapter 3 we propose to combine models of the two extremes. Models such as Transformer with high structural capacity and models such as Paragraph Vector with high contextual capacity.

[1] Provided the vocabulary size stays constant.

2. Related Work

In this chapter we go over the research that we consider relevant to the embedding of long pieces of text using the transformer architecture. First we summarize efforts that have gone into making transformer more efficient so that it can process long inputs. These advancements are crucial to embedding documents which are oftentimes much longer than 512 tokens. The next section is dedicated to typical approaches to training embedding models.

2.1 Efficient transformers

Though Transformer [Vaswani et al., 2017] has proven to be performant architecture (TODO: citations) in the world of NLP it has one inherent disadvantage when it comes to longer sequences of text. The self-attention, which is the principal part of the transformer architecture, consumes quadratic amount of memory in the length of input. This significantly limits Transformer's applicability to variety of tasks that require longer contexts such as document retrieval or summarization.

Thanks to the popularity of the transformer architecture, there is a large amount of research that is focused on making transformers more efficient [Tay et al., 2022]. Most of these efforts fall into one of the following categories:

1. Designing a new memory-efficient attention mechanism,
2. Using a custom attention implementation, or
3. Designing new transformer architecture altogether.

We go over each category separately, though these approaches are often combined. In the section dedicated to custom implementation of self-attention we also mention commonly used implementation strategies that make transformers more efficient in practice.

2.1.1 Efficient self-attention mechanisms

As self-attention is the most resource-hungry component of the transformer architecture, it makes sense to focus on it in order to make the transformer more efficient. The core of the problem is the multiplication of $N \times d$ query matrix and $N \times d$ key matrix, where N is input length and d is a dimensionality of the self-attention. Efficient attention mechanisms approximate this multiplication and thereby avoid computing and storing the $N \times N$ resulting matrix.

Sparse attention

Sparse attention approximates the full attention by ignoring dot products between some query and key vectors. Though it may seem like a crude approximation, there is research that shows that the full attention focuses mainly on few query-key vector combinations. For instance Kovaleva et al. [2019] shown that

full attentions exhibit only few repeated patterns and that by disabling some attention heads we can increase the model's performance. Both of these findings suggest that full attention is over-parametrized and its pruning may be beneficial. Child et al. [2019] shown that repeating attention patterns can be found also when processing images and that by approximating full attention using such sparse patterns we can increase the efficiency of the model without sacrificing performance.

Sparse attentions typically compose several attention patterns. One of these patterns is often full attention that is limited only to a certain neighbourhood of a given token. This corresponds to findings of Clark et al. [2019], who found that full attention gives a lot of focus on previous and next tokens. Another sparse attention pattern is usually dedicated to enable broader exchange of information between tokens. In Sparse Transformer [Child et al., 2019] distant tokens are connected by sever pre-selected tokens uniformly distributed throughout the input. In Longformer [Beltagy et al., 2020] every token can attend to every k-th distant token to increase its field of vision. BigBird [Zaheer et al., 2020] computes dot products between randomly chosen combinations of query and key vectors. These serve as a connecting nodes for other tokens when they exchange information. The last typical sparse attention pattern is a global attention that is computed only on a few tokens. Though such attention pattern is costly it is essential for tasks which require a representation of the whole input [Beltagy et al., 2020]. In Longformer some significant input tokens such as the `[CLS]` token, attend to all other tokens and vice-versa. BigBird additionally computes global attention on few extra tokens that are added to the input.

Sparse attention pattern doesn't have to be fixed, but can also change throughout the training. Sukhbaatar et al. [2019] train a transformer that learns optimal attention span. In their experiments most heads learn to attend only to few neighbouring tokens and thus make the model more efficient. Reformer [Kitaev et al., 2020] computes the full self-attention only between close key, query tokens, while letting the model decide which two tokens are "close" and which are not. To a certain degree this enables the model to learn optimal attention patterns between tokens.

Low-rank approximations and kernel tricks

Besides sparsifying the attention pattern, there are other techniques to make the self-attention more efficient in both memory and time. Wang et al. [2020] show that the attention matrix $A := \text{softmax}(\frac{QK^T}{d})$ is of low-rank and show that it can be approximated in less dimensions. By projecting the $(N \times d)$-dimensional key and value matrices into $(k \times d)$ matrices, where $k << N$ they avoid the expensive $N \times N$ matrix multiplication. The authors show that empirical performance of their model is on par with standard transformer models such as RoBERTa [Liu et al., 2019] or BERT [Devlin et al., 2019].

In another effort, Choromanski et al. [2020] look at the standard softmax self-attention through the lens of kernels. Using clever feature engineering, the authors are able to approximate the elements of the above mentioned attention matrix A as dot products of query and key feature vectors. Self-attention can be then approximated as multiplication of four matrices the projected query and key

matrices, the normalization matrix substituting the division by d and the value matrix. This allows to reorder the matrix multiplications, first multiplying the projected key and the value matrix and only after multiplying by the projected query matrix. Such reordering saves on time and space by a factor of $O(N)$ making the self-attention linear in input length.

2.1.2 Implementation enhancements

Transformer models can be made more efficient by using various implementation tricks. As modern hardware gets faster and has more memory, implementation enhancements can render theoretical advancements such as sparse attentions unnecessary. For example, Xiong et al. [2023] train a 70B model on sequences up to 32K tokens with full self-attention. Nevertheless, the necessary hardware to train such models is still unavailable to many and therefore there is still need to use theoretical advancements together with optimized implementation. For instance Jiang et al. [2023] trained an efficient transformer that uses both sparse attention and its optimized implementation. The resulting model beats competitive models with twice as many parameters in several benchmarks.

Optimized self-attention implementation

Efficient self-attention implementations view the operation as a whole rather than a series of matrix multiplications. This enables optimizations that would not be otherwise possible. The result is a single GPU kernel, that accepts the query, key and value vectors and outputs the result of a standard full-attention. Rabe and Staats [2021] proposed an implementation of full self-attention in the Jax library[1] for TPUs that uses logarithmic amount of memory in the length of the input. Dao et al. [2022] introduced *Flash Attention* that focuses on optimizing IO reads and writes and achieves non-trivial speedups. Flash Attention offers custom CUDA kernels for both block-sparse and full self-attentions. Later, Dao [2023] improved Flash Attention's parallelization and increased its the efficiency even more. Though using optimized kernel is more involved than spelling the operations out, libraries like xFormers[2] and recent versions of PyTorch[3] make it much more straightforward.

Mixed precision, gradient checkpointing and accumulation

Besides the above recent implementation enhancements, there are tricks that have been used for quite some time and not just in conjunction with transformers. We mention them here, mainly for completeness, since they dramatically lower the required memory of a transformer model and thus allow training with longer sequences.

Micikevicius et al. [2017] introduced mixed precision training which almost halves the memory requirements of the model as almost all of the activations and the gradients are computed in half precision. As the authors show, using

[1] `https://github.com/google/jax`

[2] `https://github.com/facebookresearch/xformers`

[3] `https://pytorch.org/docs/2.2/generated/torch.nn.functional.scaled_dot_product_attention.html`

additional techniques such as loss scaling, mixed precision does not worsen the results compared to traditional single precision training. In another effort to lower the amount of memory required to train a model, Chen et al. [2016] introduced gradient checkpointing that can trade speed for memory. With gradient checkpointing activations of some layers are dropped or overwritten in order to save memory, but then need to be recomputed again during backward pass. Another popular technique is gradient accumulation, which may effectively increase batch size while maintaining the same memory footprint. With gradient accumulation, gradients of batches are not applied immediately. Instead they are accumulated for k batches and only then applied to weights. This has a similar effect as multiplying the batch size by k, but not equivalent as operations like Batch Normalization [Ioffe and Szegedy, 2015] or methods like in-batch negatives behave differently. Nevertheless gradient accumulation is a good alternative, especially if the desired batch size cannot fit into the GPU memory.

2.1.3 Combination of model architectures

To circumvent the problem of memory-hungry self-attention layer, some research efforts explored combining the transformer architecture with another architectural concept, namely recursive and hierarchical networks. The common approach of these models is to not modify the self-attention, nor the maximum length of input the transformer is able to process, but to instead use transformers to process smaller pieces of text separately and contextualize them separately. Dai et al. [2019] proposes using recursive architecture of transformer nodes, where each transformer receives the hidden states of the previous one. Since gradients do not travel between the nodes, processing longer sequences requires only constant memory. The resulting model achieves state of the art performance on language modelling tasks with parameter count comparable with the competition. Simpler approach was used by Yang et al. [2020], who proposed a hierarchical model composed of transformers. First, transformers individually process segments of text producing segment-level representations, which are together with their position embeddings fed to another document-level transformer. The authors pretrain with both word-masking and segment masking losses and together with finetuning on the target tasks the model beats scores previously set by recurrent networks.

2.2 Training document embedding models

When we train a document embedding model we aim to improve the performance of the model's embeddings on downstream tasks. There are many types of downstream tasks such as classification, retrieval, clustering or visualizations and an embedding model is generally expected to perform well an all of them. Therefore there is no clear optimization objective, neither there is an objective which is universally agreed to outperform others. This makes the task of training document embedding models particularly interesting and diverse. All training techniques however, have to adapt to currently available training document corpora. The set of available supervised corpora of documents is significantly smaller compared to corpora of shorter sequences such as sentences. This is due to higher annotation costs and complexity. Nevertheless there are corpora of documents linked

via citations or links that offer some additional data for the researches to take advantage of.

In the simplest case, embedding models are trained only through some kind of word or token prediction. Paragraph Vector [Le and Mikolov, 2014] is trained on prediction of masked-out word given the embeddings of the surrounding words and of the whole document. However not all models can learn document embedding through word prediction such as Paragraph Vector. An example of such model is Transformer [Vaswani et al., 2017]. Embedding models based on Transformer's architecture use a combination of pre-training on large unlabelled corpora, where the model gains most of its knowledge, and a finetuning method that increases the quality of the model's embeddings. Cohan et al. [2020] warm start their embedding model focused on scientific literature from SciBERT [Beltagy et al., 2019], which was trained using Masked Language Modelling (or *MLM*) on scientific text. Neelakantan et al. [2022] use large generative model as an initial checkpoint for their text embedding model. Izacard et al. [2021] warm start their embedding model from BERT [Devlin et al., 2019], which is another model trained using MLM. These models however differ in how they then finetune the document embeddings. Cohan et al. [2020] use a triplet loss that takes three documents: a query, positive and a negative document. Triplet loss then minimizes the distance between the query and the positive document while maximizing the distance between the query and the negative document. To obtain a positive and a negative document for given query document the authors leverage the structure of Semantic Scholar corpus [Ammar et al., 2018]. Ostendorff et al. [2022] repeats the experiment, but shows that with a more elaborate method of sampling of negative papers, it is possible to improve the model.

Another popular technique is to train the model by contrasting several inputs' embedding against each other. Again for each input there is at least one similar to it (positive), while the others are usually deemed as dissimilar (negative). The loss would then minimize cross-entropy between the true similarities and the similarities computed from the input's embeddings. As is the case with the triplet loss, the main difference between models is how they obtain the positive and negative documents. Neelakantan et al. [2022] use the input itself as a positive, while all other inputs in given batch are considered as negatives. Using in-batch negatives is very efficient since the model can utilize each input once as a positive and several times as a negative. As the authors point out, the key of this technique is to have large batch sizes – the authors suggest batch sizes of several thousand documents. Izacard et al. [2021] obtain positives by augmenting the original document. The authors also use negatives computed in previous batches. While using these out-of-batch negatives avoids the need for a large batch size, they introduce a new set of problems such as making sure that the trained model does not change too quickly, which would make the stored negatives irrelevant and possibly harmful to the training. The authors solve this issue by using a secondary network, that is updated using the parameters of the primary embedding model.

Though an embedding is usually expected to perform well on any type of task, Singh et al. [2022] train an embedding model whose embeddings are finetuned for a given type of downstream task. The model's task-specific modules are trained in an end-to-end manner on supervised data collected by the authors. Thanks to the use of control codes and a layer connecting all task-specific modules, the

proposed model is able to share knowledge across all types of tasks. While the idea seems reasonable, the authors achieve only small improvement over state-of-the-art models that generate only single embedding.

2.2.1 Comparison to the proposed training method

As other transformer-based embedding models, ours is also warm started from a pre-trained model. However, the following finetuning of our embeddings avoids some of the downsides of the previously mentioned methods. First it does not require any structure in the training data. This is important as there is only limited amount of structured corpora of documents. Typically these would be scientific papers related via citations or Wikipedia articles connected via links. Our training method allows to use any document corpora such as a set of books or news articles. Secondly, it does not require large batch sizes. Despite the advancements mentioned in Section 2.1, using consumer-grade GPU card for embedding long inputs with transformers can still pose a practical challenge. In this context, using a batch size of several thousand is unimaginable. Third, our method does not require to maintain any secondary network while training the main model. Though, our method uses embeddings of other models, these can be generated beforehand and thus not take up resources needed to train the main embedding model. Fourth, our goal is to obtain a model that is applicable to any type of continuous text. We do not limit our embedding model only to specific field such as scientific literature. Finally, our model generates a single embedding which we evaluate on a diverse set of tasks that includes classification of individual documents, classification of document pairs and document retrieval.

3. Distilling structural and contextual qualities of document embeddings

In this chapter we describe our method of finetuning document embeddings. Our method is based on teacher-student training approach, where we distil the knowledge of two teacher embedding models into a single student model. In Section 3.1 we explain our training method in detail and define the general form of used loss function. Then in Section 3.2 we describe the two teacher models we use in the rest of the thesis. Finally, in Section 3.3 we describe the architecture and pre-training of the student model.

3.1 Training methodology

Our training methodology aims to train an embedding model such that its embeddings more faithfully represent the input. As we described in Chapter 1 we distinguish two qualities of faithful representations: structural and contextual. The goal is to instill both of these two qualities into a single embedding model. To do so, we use teacher-student training with two teacher text embedding models, one with high structural capacity, the other with high contextual capacity.

In the following subsections we describe teacher-student training in detail and give high-level overview of the proposed loss function.

3.1.1 Teacher-student training

In teacher-student training we train a single *student* model based on a non-trainable *teacher* model. The goal is to make the student model to imitate the teacher model and thereby digest the teacher's understanding of the input. Teacher-student training is useful in a number of situations. For instance it used to overcome some inherit limitation of the teacher model such as its large size [Sanh et al., 2019]. Also, teacher-student training can be used to enforce some kind of alignment between models' outputs [Reimers and Gurevych, 2020]. The motivation to align the model's outputs often comes from an intuition about language, such as that embeddings of two translations of a given sentence should be the close to each other.

In our setting, we assume two embedding models $\mathcal{T}_S$, $\mathcal{T}_C$ with high structural and contextual capacities respectively. Teacher-student training allows us to instill both of these capacities into a third model $\mathcal{S}$, while also avoiding some of the architectural limitations of both reference models. For convenience we call $\mathcal{T}_S$ *structural teacher*, $\mathcal{T}_C$ *contextual teacher*, and $\mathcal{S}$ *student*.

3.1.2 Abstract loss formulation

Our loss function should align the output of a student model $\mathcal{S}$ with outputs of two teacher models $\mathcal{T}_S$, and $\mathcal{T}_C$. We use two similarity functions $\mathcal{L}_S$, $\mathcal{L}_C$

that compare the student's embedding $y_{\mathcal{S}}$ with structural teacher embedding $y_{\mathcal{T}_S}$ and with contextual teacher embedding $y_{\mathcal{T}_C}$ respectively. As we are unsure which balance of $\mathcal{L}_S$ and $\mathcal{L}_C$ is optimal we introduce weighting parameter λ that balances the effect of the two losses on the final loss. In the most general form, we can assume λ to be dependent on the input text x, since the performance of the teacher models might vary across different inputs. In particular, we can expect λ to be dependent on the length of the input, since for shorter inputs the context is minimal and therefore expendable. The form of the loss as we have described it is defined in Equation 3.1. We explore concrete options for $\mathcal{L}_S$, $\mathcal{L}_C$ and $\lambda(x)$ in Chapter 4.

$$\mathcal{L}(x, y_{\mathcal{S}}, y_{\mathcal{T}_S}, y_{\mathcal{T}_C}, \lambda) = \lambda(x)\mathcal{L}_S(y_{\mathcal{S}}, y_{\mathcal{T}_S}) + \mathcal{L}_C(y_{\mathcal{S}}, y_{\mathcal{T}_S}) \tag{3.1}$$

The two losses could go, in theory, against each other and slow down or halt the training. To avoid this we choose one of the losses to be more strict, while the other to be more forgiving. As we mentioned in Section 1.2.3 we view structural quality as the more important one. Structural quality is necessary to truthfully understand the input text, whereas contextual quality is there just to fill-in the overall theme of the document. Therefore, we choose the structural loss $\mathcal{L}_S$ to be the stricter and exact loss forcing the student to adapt to the structural teacher as much as possible. On the other hand, the contextual loss $\mathcal{L}_C$ should give the student model more freedom in the form of the produced embedding, but still force it to incorporate the information from the contextual embedding.

3.2 Teacher models

In this section we present the teacher models we use during our experiments in Chapter 4. We highlight why we choose the particular models and how we obtain the their embeddings. We chose Sentence-BERT [Reimers and Gurevych, 2020] as the structural teacher model, and Paragraph Vector [Le and Mikolov, 2014] (or PV) as the context teacher model.

3.2.1 SBERT

Sentence-BERT [Reimers and Gurevych, 2019] is a composition of a BERT-like [Devlin et al., 2019] encoder with a mean pooling layer above its final hidden states. We have chosen SBERT as structural teacher for its Transformer architecture, which as we have discussed in Chapter 1, has high structural capacity. Additionally, SBERT is finetuned on NLI datasets to increase its text understanding and to produce embeddings which are semantically meaningful.

There are several versions of SBERT, that differ in the base Transformer encoder, from which is SBERT warm-started and then finetuned. We use SBERT warm-started from MPNet [Song et al., 2020], that achieves high scores across many sentence embedding benchmarks[1], while being reasonably small. To be exact we use its Hugging Face implementation named `sentence-transformers/all-mpnet-base-v2`.

[1] `https://sbert.net/docs/pretrained_models.html`

We generate the embeddings of any training dataset directly without any additional finetuning.

3.2.2 Paragraph Vector

Paragraph Vector [Le and Mikolov, 2014] (also known as Doc2Vec) is a simple text-embedding model that views the input as a Bag of Words (or BoW). Paragraph Vector is composed of two sub-models called Distributed Memory (DM) and Distributed Bag of Words (DBOW). In practice one or both models can be used, where the final embedding is the concatenation of all sub-models' outputs. Both models construct embedding of the whole input with which they predict a randomly masked out input word. DM additionally uses embeddings of neighbouring words.

We choose Paragraph Vector as contextual teacher due to its unique architecture that forces the model to develop a single vector, that summarizes the common theme of the document. Moreover, Paragraph Vector does not have limited maximum input length, and so as a contextual teacher it will always provide some signal to the student regarding the document's context. Also, even though Paragraph Vector cannot match the performance of a substantially more complex models such as Transformers, Dai et al. [2015] show that for larger datasets Paragraph Vector outperforms classical embedding models such as Latent Dirichlet Allocation [Blei et al., 2003] or TF-IDF weighted BoW model [Harris, 1954].

We generate Paragraph Vector's embeddings after training the model on *all* training datasets for which we require a contextual teacher's embedding. There are many hyperparameters that govern how Paragraph Vector is trained. Since there are no universally agreed best-performing values for any of them, we see these as hyperparameters of our teacher-student training method. We explore the effect of some of these parameters on the model's performance in Chapter 4.

TODO: my own graphic here

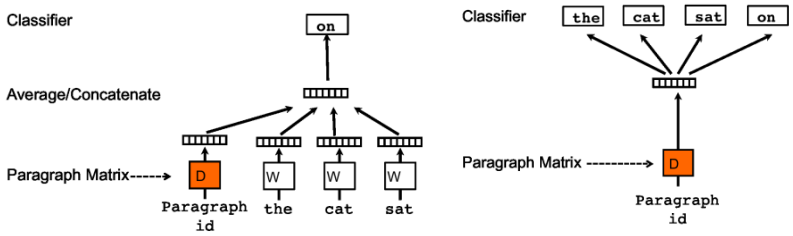

Figure 3.1: PV-DM and PV-DBOW architectures.

3.3 Student model

To test our finetuning method we used a student model already pretrained using Masked Language Modelling on a large corpus. Since our training method relies on the model's basic natural language understanding, we would have to pre-train the model either way. Additionally by using a checkpoint of a pretrained model we dramatically save on resources.

When choosing our model, we valued the following properties. Since we are limited in resources, we needed the student model to be reasonably small and memory efficient. Also, as we are embedding documents, the model needed to have larger maximum input length. We also valued the model's performance, ease of use and simplicity of the model. In the end we have chosen Longformer [Beltagy et al., 2020] as it is reasonably small, memory efficient, performs above averege in comparison to other similar models [Tay et al., 2020] and its self-attention mechanism is straightforward.

We emphasize that our training method can be applied theoretically to any transformer-based model. We, therefore view this decision as a practical one that allows us to test our method more easily, rather then theoretical one that is part of our method.

3.3.1 Longformer

Longformer [Beltagy et al., 2020], is a transformer encoder that is able to process input sequences longer than traditional Transformer using its efficient implementation of self-attention.

Self-attention mechanism

Longformer uses sparse self-attention that does not compute the full $N \times N$ attention matrix. Instead Longformer's self-attention relies on composition of several patterns. We use two patterns: sliding window attention and global attention. With sliding window attention each token attends to $\frac{1}{2}\omega$ tokens on its either side. The neighbourhood size ω can vary across transformer layers. This can be helpful when we try to save on the number of floating point operations per second (or *FLOPS*) while minimizing the negative impact on performance [Sukhbaatar et al., 2019]. With global attention only few tokens selected at runtime attend to all other tokens. Longformer uses two different set of weights for local and global attentions. For easier reproducibility of our experiments we avoid the custom CUDA kernel, and use the author's python block implementation.

Training

Longformer is warm-started from a RoBERTa [Liu et al., 2019] checkpoint with its learned positional embeddings copied 8 times to support inputs up to 4096 tokens long. Copying RoBERTa's positional embeddings showed to be an effective way how to increase the maximum context length of a transformer with learned positional embeddings. Then it was trained further using MLM on long documents for 65k gradient steps.

The specially compiled training corpus has some overlap with RoBERTa's pretraining corpus, but is more focused on longer pieces of text. It includes the following datasets.

- Book corpus [Zhu et al., 2015]
- English Wikipedia

- One third of articles from Realnews dataset [Zellers et al., 2019] with more than 1200 tokens
- One third of the Stories corpus [Trinh and Le, 2018]

4. Experiments

In this chapter we test several settings of the training method we introduced in Chapter 3 and test each variant by evaluating the trained models on downstream tasks. The goal of this chapter is to find the best performing hyperparameters of our training method while trying to outperform both teachers as well as the base student's checkpoint.

This chapter is laid out as follows. First we describe the training data and its format in Section 4.1. Second we define how we compare models in Section 4.2. Third we describe the base configuration of the student model along with the baselines, whose score we are trying to exceed, in Section 4.3. Then we experiment with the structural loss in Section 4.4. Finally with the structural loss already given, we find the best performing contextual loss and the weighting of the two losses in Section 4.5.

4.1 Training data

The training dataset for our validation experiments, which we label as `val_500k`, is created in the same way as our final training dataset `train_1500k` in Chapter 5 except for the number of containing documents. In short, we equally sample two sources that were used to train our base student model: English Wikipedia and RealNews articles [Zellers et al., 2019] with documents longer than or 1200 Longformer tokens. We use our base student's training data, so that any performance gains of the student model over its base checkpoint, can be attributed to our training method, rather than to a higher quality training dataset.

As can be seen in Table 4.1, `val_500k` is a collection of large documents; the mean token count per document is over 1300 and around 65% of the documents could not be processed by a traditional Transformer such as RoBERTa [Liu et al., 2019] or SBERT. The distribution of document length is far from uniform as we show in Figure 4.1.

4.2 Validation tasks

We compare the training variants we compare the trained models' performances on a set of downstream tasks. The set is composed of our evaluation tasks, which

Split	Train	Validation
Documents	500 000	10 000
Tokens	6.85e+08	1.37e+07
Tokens per document	1370.74±1723.38	1371.65±1717.24
SBERT tokens over 384	70.56%	70.07%
SBERT tokens over 512	66.37%	66.14%

Table 4.1: Statistics of `val_500k`. For each split we also show the percentage of documents with the number of SBERT tokens above given threshold.

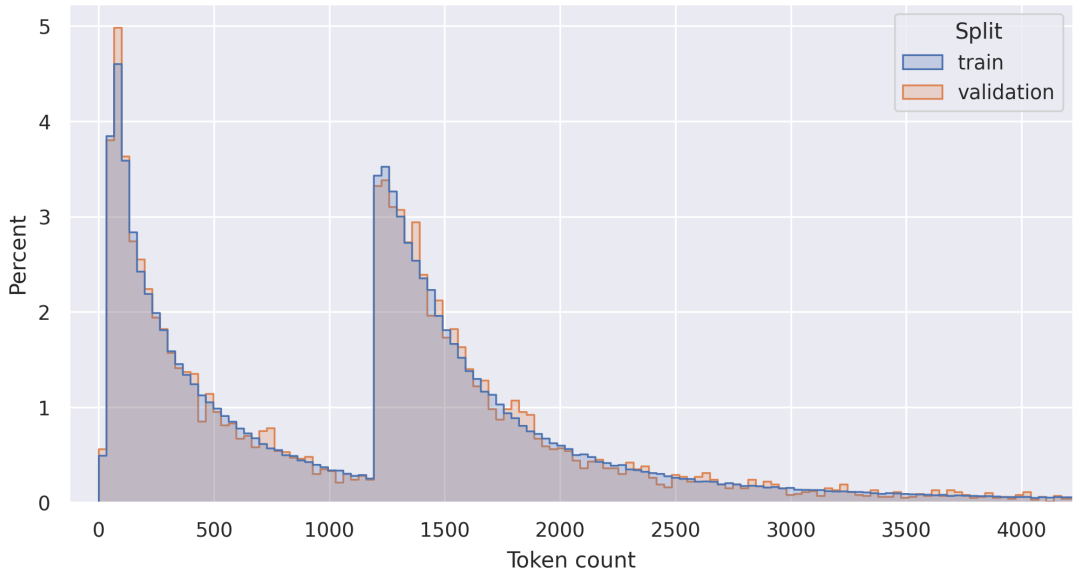

Figure 4.1: 2-peak distribution of number of Longformer tokens per document in `val_500k`.

we describe in Chapter 5, that either have large enough training split suitable for cross-validation or validation split. We tabulate the chosen tasks in Table 4.2. All tasks are classifications and most are scored using a validation split, except for IMDB where we take the mean score of 5-fold cross-validation runs.

To make the validation process quicker we downsample the validation and train splits to 10000 examples. We downsample the datasets according to their label distribution, so that the truncated split has nearly identical label distribution to the original one. We show the number of documents in each task in Table 4.3. In the same table, we also highlight the percentage of documents with above 384 SBERT tokens. These numbers show how many documents must be truncated to fit into the context size of our structural teacher $\mathcal{T}_S$ and therefore also for how many documents we can expect $\mathcal{T}_S$ to receive full information and to give, perhaps, more meaningful embeddings.

We use binary or micro-averaged accuracy as the scoring metric. Often we need to compare performance across several tasks. However not all tasks are equally difficult and so averaging accuracies would lead us to favoring models that performed well on easy tasks, and undervaluing models that performed well on the more difficult tasks. Therefore, we normalize the accuracy by the highest score reached for the given task within the considered models and thereby making the tasks equally difficult. We call this metric *normalized accuracy*.

When validating a trained embedding model on a task we finetune a head that transforms the embeddings into the output format required by the given task. During validation we do not finetune the embedding model itself. Besides speeding up the validation, this gives us also more genuine picture of the performance of the embedding model. Since all our validation tasks are classifications, the heads are just 2-layer neural network classifiers with a cross-entropy loss. We give the complete list of classifier's hyperparameters and training parameters in Table 4.4.

TODO: ask if class distribution graph would add anything or not

Dataset	Evaluation	Classes	Class percentage	
			Train	Validation
ARXIV [He et al., 2019]	validation split	11	9.09±1.24%	9.09±1.01%
IMDB [Maas et al., 2011]	cross-validation	2	50.00±0.00%	-
OC [Xuhui Zhou, 2020]	validation split	2	50.00±0.06%	50.00±0.15%
AAN [Xuhui Zhou, 2020]	validation split	2	50.00±1.50%	50.00±4.57%
S2ORC [Xuhui Zhou, 2020]	validation split	2	50.00±0.09%	50.00±0.32%
PAN [Xuhui Zhou, 2020]	validation split	2	50.00±0.00%	50.00±0.00%

Table 4.2: Validation tasks used in this chapter. Each task is evaluated either using a validation split or cross-validated on training split using 5 folds. The class distributions of all tasks are well balanced except for AAN as can be seen from the mean and standard deviation of class percentages for given task and split.

Dataset	Split	Documents	Over 384 tokens	Over 512 tokens
ARXIV	Train	†10 000	100.00%	100.00%
	Test	2 500	100.00%	100.00%
IMDB	Train	†10 000	24.34%	14.63%
OC	Train	†10 000	12.22%	1.00%
	Test	†10 000	12.38%	1.05%
AAN	Train	†10 000	0.51%	0.03%
	Test	†10 000	0.34%	0.07%
S2ORC	Train	†10 000	33.23%	18.40%
	Test	†10 000	32.94%	18.67%
PAN	Train	†10 000	72.72%	63.37%
	Test	2 908	59.99%	45.34%

Table 4.3: The statistics of validation tasks used in this chapter. We truncated splits marked with † to speed up validation of models. We truncate a split by downsampling it according to its label distribution. For each split we give a percentage of documents with number of SBERT tokens above given threshold.

Parameter	Value
Hidden features	50
Hidden dropout rate	0.5
Hidden activation	ReLU
Epochs	10
Batch size	32
Weight decay	0.1
Label smoothing	0.1
Learning rate	1e-4
Learning rate decay	Cosine
Maximum gradient norm	1.0
Optimizer	AdamW
Mixed-precision training	Yes

Table 4.4: Hyperparameters and training parameters of classification heads used during validation.

Parameter	Value
Batch size	6
Weight decay	0.1
Learning rate	1e-4
Learning rate decay	Cosine
Maximum gradient norm	1.0
Optimizer	AdamW
Gradient accumulation steps	1
Warmup steps	10% of training steps
Gradient checkpointing	Yes
Mixed-precision training	Yes

Table 4.5: Parameters for training student models during validation experiments. We use these values unless specified otherwise.

4.3 Student's model configuration and baselines

As we explained in Section 3.3, we chose Longformer [Beltagy et al., 2020] as our student model. We use Longformer's smaller base version with about 126M parameters implemented by HuggingFace `datasets` library[1].

To obtain the input's embedding we average the final hidden states. We do not use global attention and restricting ourselves to only sliding window attention, with the window sizes ω set to the default 512 tokens. During training of the student model we aim for fast convergence with small memory footprint. We therefore use high learning rate, no gradient accumulation steps, mixed-precision training and gradient checkpointing. The complete list of training parameters is displayed in Table 4.5.

4.3.1 Baselines

As we already mentioned in the beginning of this chapter, the goal of the following experiments is to find out weather we can surpass both teachers as well as the base checkpoint of our student model using our teacher-student training approach. To that end we compare our student model throughout this chapter to three models: SBERT [Reimers and Gurevych, 2019], Longformer [Beltagy et al., 2020] and PV [Le and Mikolov, 2014]. For SBERT we use the same version and checkpoint as for our structural teacher. For Longformer we use the same configuration we use for our student model. For Paragraph Vector we use the same model and checkpoint as for our contextual teacher, which we train in Section 4.5.1.

4.4 Structural loss

To limit the number of hyperparameter combinations, we first experiment with structural loss only and after finding the best performing $\mathcal{L}_S$ we start using both losses at the same time, finetuning both $\mathcal{L}_C$ and λ. This approach is based on our

[1] `https://huggingface.co/allenai/longformer-base-4096`

initial experiments, where the structural teacher proved to be more important to the performance of the student model than the contextual teacher. We therefore finetune the contextual loss to the already given structural loss instead of the other way around.

The structural loss $\mathcal{L}_S$ compares the embeddings of the student $\mathcal{S}$ to those of the structural teacher $\mathcal{T}_S$. As we mentioned in Section 3.1.2 we want $\mathcal{L}_S$ to minimize the distance bewteen the two embeddings as much as possible.

We considered two basic losses: Mean Squared Error (or MSE) and cosine distance.

While MSE forces the two embeddings to be identical, cosine distance only forces the embeddings to have the same direction.

The structural loss $\mathcal{L}_S$ is one of the two losses our teacher-student training will use. For inputs that fit into the maximum context length of the structural teacher, the structural loss should minimize the dissimilarity between the structural teacher's and the student's embeddings as much as possible. This follows our assumption that for inputs that the structural model can process whole, the structural model produces an embedding with the deepest understanding of the input that is available to us. It is thus important that the similarity $\mathcal{L}_D$ should take into account is the absolute similarity rather than some approximate measure of it.

(TODO: What else to say about this? What dissimilarity we use will be in Experiments chapter as well as deciding "when an input is applicable".)

4.5 Improving both qualities at the same time

After finding the best performing variant of $\mathcal{L}_S$, our goal is to find the best performing $\mathcal{L}_C$. Instead of experimenting with just the contextual loss, we train models with both losses used. So in essence, we look for a variant of $\mathcal{L}_C$ that will complement the already chosen $\mathcal{L}_S$ the best, rather than looking for the overall best performing $\mathcal{L}_C$.

First we train the contextual teacher $\mathcal{T}_C$ in the next section. In the following sections we experiment with $\mathcal{L}_C$ and with the weighting of the structural and contextual losses.

4.5.1 Generating contextual embeddings

As we have explained in Section 3.2.2, we have chosen Paragraph Vector [Le and Mikolov, 2014] as our contextual teacher. Since there is no concept of pretraining, as in the case of transformers, we train PV from scratch. We use the implementation provided by Gensim library[2].

Grid searching best performing hyperparameters

There is a great number of hyperparameters governing the training of Paragraph Vector. We focus only on four hyperparameters, while following the recommendations of related literature or leaving the parameter to its default value, sensibly

[2] `https://radimrehurek.com/gensim`

set by the Gensim library authors. Both the adopted and the grid-searched hyperparameters are displayed in Table 4.6, where the mentioned hyperparameters have the following meaning:

- `dm` – PV architecture; true for Distributed Memory (DM), false for Distributed Bag of Words (DBOW)
- `vector_size` – dimensionality of the generated embedding
- `min_count` – words with document frequency below this limit will be ignored
- `text_pre_process` – applied word processing done before the model's training; for stemming we used PorterStemmer implemented by `nltk` library[3]
- `negative` – number of noise words used for negative sampling during training
- `window` – the maximum distance between known and predicated word
- `sample` – percentile threshold configuring which words will be downsampled; 0 for no downsampling
- `dbow_words` – whether to train word embeddings using Word2Vec [Mikolov et al., 2013] Skip-gram architecture simultaneously with DBOW document embeddings
- `epochs` – number of iterations over the corpus

We train both DM and DBOW as recommended by the authors of Paragraph Vector [Le and Mikolov, 2014]. `vector_size`, `min_count` and `text_pre_process` all control the regularization of the model. Settings such as higher dimensional embedding, small min count and no text pre-processing, regularize the model the least. They give the model the most information on its inputs, while also providing it with large embedding through which the model can express in more detail. On the other hand, using lower dimensional embedding, large minimal count and heavy text pre-processing, forces the model to be more general and less precise. The model has less detailed information on its input and must squeeze all of it to a small vector. We did not see any value in trying higher dimensions of embeddings than 1024 as in later experiments we need to distill the contextual embedding to a 768-dimensional embedding of our student model. Intuitively, the larger the contextual embedding is going to be, the smaller the percentage of information the student model is going to be able to digest. Also there is no value in considering `min_count` to be lower than 2, since we would only add words that are unique to a single document. Embeddings of such words would be poorly trained and would not add meaningful information to the document's embedding. Last hyperparameter that deserves mention is `dbow_words`. DBOW on its own does not train word embeddings, which are randomly generated. Setting `dbow_words` to true causes DBOW to also train word embeddings using Word2Vec Skip-gram model in each epoch. Lau and Baldwin [2016] showed that random word embeddings hurt the model significantly, and so in our opinion the overall longer training, with `dbow_words` set to true, is worth the performance gain.

[3] `https://www.nltk.org/api/nltk.stem.porter.html`

Hyperparameter	Value(s)	Recommended by
dm	true, false	-
vector_size	100, 768, 1024	-
min_count	2, 10% of train corpus	-
text_pre_process	stem, lowercase, none	-
window	5	default
negative	5	default, Lau and Baldwin [2016]
sample	0	default
dbow_words	true	Lau and Baldwin [2016]
epochs	10	default, Dai et al. [2015]

Table 4.6: Used hyperparameters for training Paragraph Vector. We grid-searched four hyperparameters: PV architecture, vector size, min. word count and pre-processing of words. The rest of the hyperparameters we adopted either the default values set by the authors of the Gensim library[2] or recommended by the mentioned literature.

We train all variants on the entire `val_500k` corpus. We report the normalized classification for each variant in Figure 4.2. Large embedding dimensions and low minimum count were favored. There is no clear preference among the types of pre-processing. DBOWs vary more in performance, occupying the best and the worst positions, whereas DMs are a bit more consistent.

Will we actually do the concatenation? There is an opportunity to make up some time, which I don't have enough of. Also I thing the argument "we need to cramp this huge (2048d) contextual embedding down into 768d student embedding" is pretty sound.

Training the contextual teacher

We take the best performing hyperparameter values from the previous section and train the model on a large corpora. As we later train our student model with the contextual teacher's outputs, each piece of data we train our contextual model with, is effectively also part of the student's training data. Therefore we construct the PV training dataset similarly as `val_500k`: we evenly sample English Wikipedia articles and RealNews [Zellers et al., 2019] articles with more than 1200 Longformer tokens. We do so for the same reasons that were mentioned in Section 4.1.

Since PV is a small model and is able to process data very quickly during training, we can afford to use all of the RealNews articles that are long enough and an equal amount of Wikipedia articles. This results in a corpus that has 8.3M documents. We train the model for 10 epochs, but restrict the maximum vocabulary size to 1.2×10^7 words to limit the memory footprint to roughly 140 GB.

4.5.2 Structural loss selection

The contextual loss $\mathcal{L}_B$ minimizes the disagreements between the embeddings of the contextual teacher model and that of the student model. There are two major differences between the contextual loss and the structural loss. First the contextual loss is always applicable. It might not be the loss that reflects our

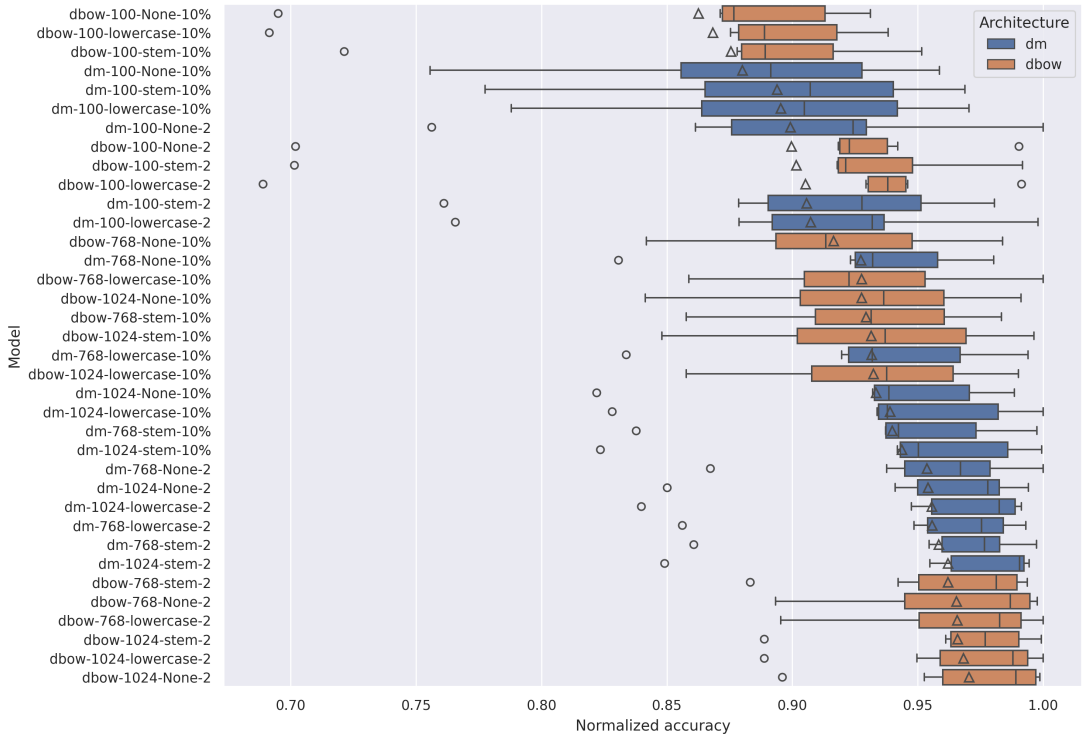

Figure 4.2: Boxplot of normalized accuracies of all PV models on validation tasks. Model is identified by architecture, embedding dimension, text pre-processing and mininum count, in this order, delimited by a hyphen. The models are sorted according to the mean normalized accuracy marked with a triangle.

ultimate goal the most for the given input. Nevertheless we still assume that for *every* input the contextual teacher model offers some information which the structural teacher model does not have. Second the contextual loss is not as exact as the structural loss. Instead of forcing the teacher model to adhere to two possibly very distinct ways how to encode information into a dense vector representation, we give the model a little bit of freedom. We do so by letting the model decide how exactly it should encode the information contained in the contextual teacher's embedding. We give the model more leeway on the breadth side rather than the structural side, because we expect that the precision of the embedding is more important than capturing every piece of the input.

With the above taken into an account we chose to use a variant of *Canonical Correlation Analysis* Hotelling [1992] (or *CCA*) as a base for our breath loss $\mathcal{L}_B$. A variant of CCA fits our needs very nicely as it computes a correlation of outputs after some projection. While the projection gives the model the freedom to restructure its embeddings, the correlation ensures that the two embeddings agree with each other. Before going further let us briefly describe CCA and its variants we considered.

Canonical Correlation Analysis

Canonical Correlation Analysis (CCA) computes linear projections of two vectors such that their projections are maximally correlated. For formal definition reference Definition 1.

Definition 1 (Canonical Correlation Analysis)**.** *For two matrices $X_1 \in \mathbb{R}^{n_1 \times m_1}$ and $X_2 \in \mathbb{R}^{n_2 \times m_1}$, Canonical Correlation Analysis finds $p \in \mathbb{R}_1^m$ and $q \in \mathbb{R}_2^m$ that maximize*

$$\begin{aligned} corr(X_1 p, X_2 q) &= \frac{p^T X_1^T X_2 q}{||Xp||||Yq||} \\ s.t. \quad ||X_1 p|| &= ||X_2 q|| = 1 \end{aligned} \tag{4.1}$$

Definition 1 suggests that CCA gives only a single number as the measure of correlation of two matrices. When the dimensions of the input vectors are large enough, however, often there exists at least one combination of features that results in correlation of 1. As this would make CCA relatively useless in the context of high-dimensional spaces we assume multiple correlations for several mutually orthogonal projections. We name such Canonical Correlation analysis as CCA for more dimensions and define it formally in Definition 2

Definition 2 (Canonical Correlation Analysis for more dimensions)**.** *For two matrices $X_1 \in \mathbb{R}^{n_1 \times m_1}$ and $X_2 \in \mathbb{R}^{n_2 \times m_1}$, Canonical Correlation Analysis for k dimensions finds $P \in \mathbb{R}^{m_1 \times k}$ and $Q \in \mathbb{R}^{m_2 \times k}$ that maximize*

$$\begin{aligned} &\sum_{i=1}^{k} corr(X_1 P_{*i}, X_2 Q_{*i}) \\ s.t. \quad &P^T X_1^T X_1 P = I_k = Q^T X_2^T X_2 Q \end{aligned} \tag{4.2}$$

If we consider the conditions in Definition 2, the resulting value can be easily reformulated:

$$\sum_{i=1}^{k} corr(X_1 P_{*i}, X_2 Q_{*i}) = trace(P^T X_1^T X_2 Q) = trace(P^T \Sigma_{X_1 X_2} Q) \tag{4.3}$$

Where $\Sigma_{X_1 X_2}$ is the covariance matrix of X_1 and X_2.

TODO: analytical solution to pure CCA

As we noted above we would like to use CCA as a base for our contextual loss $\mathcal{L}_B$ in order to establish correspondence between the contextual teacher's embedding and the student's. The problem using CCA as it was defined in Definition 2 as a loss is that it is defined in the context of the whole dataset rather than just a minibatch. It is therefore unclear how should be CCA computed using just a pair of minibatches.

Someone (TODO: citation) found that using large enough batch size is sufficient for the training to converge.

Deep CCA

Deep CCA (or DCCA) is an extension of CCA that computes projections using neural networks. As such it is more powerful than plain CCA as it allows for non-linear projections. To compute DCCA the network has to be trained on the pairs of vectors with CCA as its loss.

TODO: graphic of architecture

If CCA is weaker condition than just correlation, DCCA is even weaker since there is no limit to how the projections should look like.

Soft CCA

Soft CCA reformulates CCA and thus allows its straightforward use in the context of minibatches. With constraints from Definition 2 taken into account, CCA can be formulated using Forbenious matrix norm:

$$
\begin{aligned}
P^*, Q^* &= \underset{P,Q}{\operatorname{argmin}} \, ||X_1 P - X_2 Q||_F^2 && (4.4)\\
&= \underset{P,Q}{\operatorname{argmin}} \, trace\Big((X_1 P - X_2 Q)^T (X_1 P - X_2 Q)\Big) && (4.5)\\
&= \underset{P,Q}{\operatorname{argmin}} \, -2trace(P^T X_1^T X_2 Q) && (4.6)\\
&= \underset{P,Q}{\operatorname{argmax}} \, trace(P^T X_1^T X_2 Q) && (4.7)\\
&= \underset{P,Q}{\operatorname{argmax}} \sum_{i=1}^{k} corr(X_1 P_{*i}, X_2 Q_{*i}) && (4.8)
\end{aligned}
$$

So, in essence minimizing CCA is the same as minimizing the difference between projections, whose features are decorrelated. This is the formulation Soft CCA builds on. Soft CCA decomposes CCA into to parts:

- minimization of the difference between projections

$$||X_1 P - X_2 Q||_F^2 \tag{4.9}$$

- decorrelation of each projection P

$$\sum_{i \neq j} (P^T X_{mini}^T X_{mini} P)_{ij} = \sum_{i \neq j} \Sigma_{X_{mini} P}, \tag{4.10}$$

  where X_{mini} is batch-normalized minibatch.

To bring correlation matrix of the current minibatch $\Sigma_{X_{mini} P}$ closer to the true covariance, the decorrelation part is in fact computed from a covariance matrix that is incrementally learned.

In this way Soft CCA incrementally decreases CCA through incrementally learned approximation of the projections' covariances.

Loss selection

- mention which of the CCA variants we chose
- reiterate the formulation in the context of teacher-student training

Mean Squared Error

Vanilla CCA

SoftCCA

4.5.3 Balancing structural and contextual

4.6 Ablation studies

- here we can ablate some parts of our method (e.g. training with just the contextual teacher)

5. Evaluation

In this chapter we evaluate our student model on a number of downstream tasks. We train a student model with the best performing hyperparameters' values from Chapter 4 on a large text corpora. We then evaluate this final model on a set of 8 document downstream tasks. We simulate a scenario where there are no data to train a dedicated model on or finetune a pre-trained embedding model. In our opinion this is where a generic pre-trained embedding model is most valued and we therefore focus on evaluating embedding models for this exact use case. We also use the same checkpoint for all types of tasks to highlight the usability of our model in different use cases.

5.1 Our model

TODO: think of a name, so we are not constantly labelling it "our"

5.1.1 Training data

We train our student model on a corpus of long documents. We train our student model on the same data that were used to train Longformer. By using the data our student model was already trained on, we eliminate the possibility that the training data were the key ingredient and the cause of the performance gain or loss of our student model. Instead, our training method is to be blamed or given credit. Following Longformer's approach, we use the English Wikipedia[1] and documents from Realnews dataset [Zellers et al., 2019] that have above 1200 Longformer's tokens. We mix both sources equally, by randomly sampling from both datasets. Because Wikipedia does not have a validation split, we sample the validation documents again from its train split, avoiding already chosen training documents. We label the resulting dataset as `train_1500k` to clearly identify it. As can be seen in the Table 5.1, `train_1500k` contains 1.5 million long documents out of which only about 30% can be processed by our structural teacher whole. Though `train_1500k`'s length distribution, which is depicted in Figure 5.1, is far from uniform.

[1] `https://huggingface.co/datasets/wikipedia/viewer/20220301.en`

Split	Train	Validation
Documents	1 500 000	30 000
Tokens	2.06e+09	4.15e+07
Avg. tokens per document	1374.18±1736.64	1382.12±1697.09
SBERT tokens over 384	70.54%	70.73%
SBERT tokens over 512	66.36%	66.84%

Table 5.1: Statistics of `train_1500k`. For each split we also show the percentage of documents with the number of SBERT tokens above given threshold.

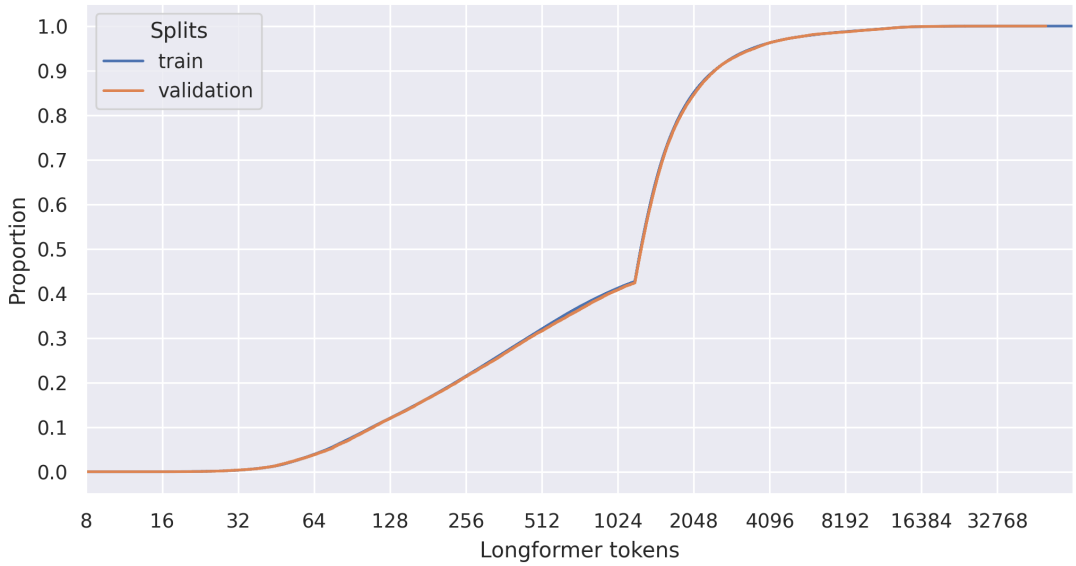

Figure 5.1: Estimated cumulative distribution function of number of Longformer tokens per document in `train_1500k`.

5.2 Classification tasks

Classification is a typical type of evaluation task for a textual embedding. In our evaluation set we cover several use cases including classification based on sentiment, on content, as well as citation and plagiarism detection. Our evaluation set also contains large span of document lengths from the very short texts having just over 100 tokens to extremely long with about 12 thousand tokens. An overview of our classification tasks is displayed in Table 5.2 and their splits' statistics in Table 5.3. We also plot the length distribution of all tasks in Figure 5.2 to show how well the tasks cover the range of document lengths.

We use a heavily regularized neural network as a classifier for each task. The classifier gets an embedding or a concatenation of two embeddings, in case of document pair classification, and is trained with a cross entropy loss to classify its input. We provide all the relevant hyperparameters in Table 5.4. For all task we use accuracy as the evaluation metric. For tasks with more classes we use micro averaging to give each input the same weight.

We do not finetune the embedding model for each task. While it could increase the performance, finetuning the embedding model for each task makes it more difficult to estimate the benefits of our training method.

5.2.1 Tasks' descriptions

IMDB movie reviews

The IMDB dataset [Maas et al., 2011] (denoted as IMDB) is a binary classification dataset that is quite frequently included in evaluation of long-context NLP models [Zaheer et al., 2020, Beltagy et al., 2020, Le and Mikolov, 2014]. The datasets consists of movie reviews, each with associated rating on a 10 point scale. The reviews that rated the movie with 7 points or higher are classified as positive, and reviews with less than 4 points are classified as negative. For each movie

Dataset	Inputs	Classes	Class percentage	
			Train	Test
ARXIV	documents	11	9.09±1.25%	9.09±1.30%
IMDB	documents	2	50.00±0.00%	50.00±0.00%
OC	pairs of documents	2	50.00±0.07%	50.00±0.34%
AAN	pairs of documents	2	50.00±1.50%	50.00±0.77%
S2ORC	pairs of documents	2	50.00±0.09%	50.00±0.33%
PAN	pairs of documents	2	50.00±0.00%	50.00±0.00%

Table 5.2: Overview of our classification tasks. Each task classifies either documents or pairs of documents into several classes. We also show the mean and the standard deviation of percentage of class label within a given split.

Dataset	Split	Documents	Tokens over 384	Tokens over 512
ARXIV	Train	28 388	100.00%	100.00%
	Test	2 500	100.00%	100.00%
IMDB	Train	25 000	24.56%	14.68%
	Test	25 000	23.54%	13.95%
OC	Train	240 000	12.31%	1.07%
	Test	30 000	12.24%	1.02%
AAN	Train	106 592	0.37%	0.06%
	Test	13 324	0.45%	0.08%
S2ORC	Train	152 000	33.24%	18.67%
	Test	19 000	32.96%	18.20%
PAN	Train	17 968	69.93%	59.45%
	Test	2 906	60.82%	47.37%

Table 5.3: Statistics of splits of our classification evaluation tasks. For each task and split we include the percentage of documents with SBERT tokens above a given threshold. This illustrates how many documents can our structural teacher process as a whole and how many documents need to be truncated.

Parameter	Value
Hidden features	50
Hidden dropout rate	0.5
Hidden activation	ReLU
Epochs	30
Batch size	32
Weight decay	0.1
Label smoothing	0.1
Learning rate	1e-4
Learning rate decay	Cosine
Maximum gradient norm	1.0
Optimizer	AdamW
Mixed-precision training	Yes

Table 5.4: Parameters for training classification heads during evaluation.

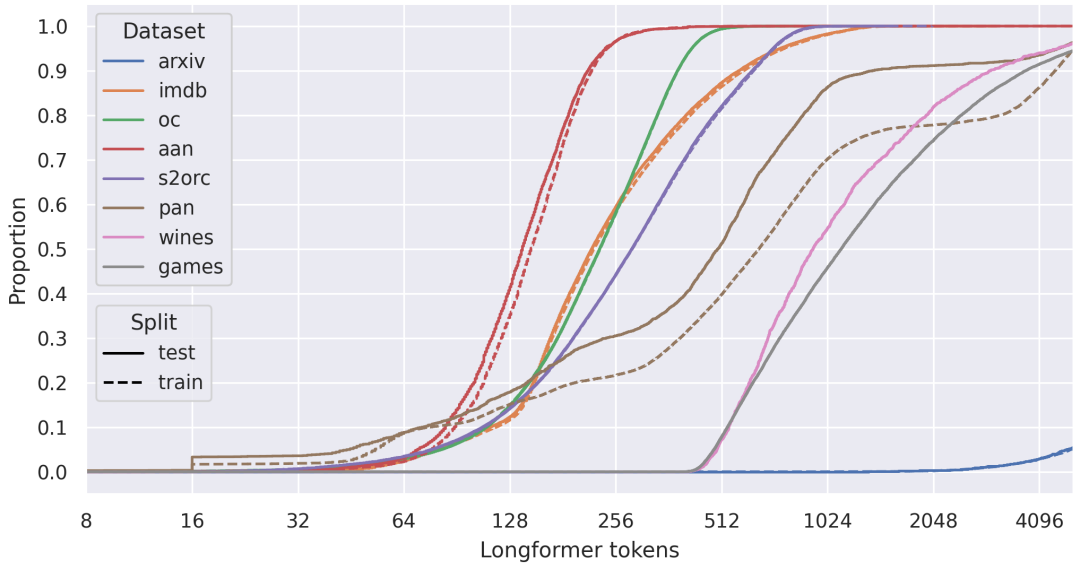

Figure 5.2: Estimated cumulative length distribution of the number of Longformer tokens in a document.

there can be up to 30 reviews and the test set contains disjoint set of movies. Along with the train and test splits the dataset contains also an unsupervised split without any rating or class labels. While the majority of the reviews are shorter; around 25% of documents are longer than the maximal context of our structural teacher and around 14% being longer than 512 tokens. Consequently, we use IMDB dataset to gain an estimate of the performance of our model on short to medium documents.

Arxiv papers

Arxiv papers [He et al., 2019] (denoted as ARXIV) is a collection of papers from ArXiv[2] – online archive of scholary papers. Each paper contains its full text spanning at least 1000 words. The longest papers are truncated to 10000 words. The papers are classified to 11 groups, that correspond to the scientific field of the paper. Since each paper can be associated with several scientific fields a small portion of the documents ($\approx 3.1\%$) appears more than once, but with different label each time. The scientific fields cover mainly fields of Computer science such as Artificial inteligence or Data structures but also fields connected with Mathematics such as Group theory or Statistics theory. The classes are not quite balanced with the most numerous class having around 12% of documents, while the least numerous class only having around 7%. ARXIV represents the extreme case of very long documents. As 97% of the dataset's documents are longer than the maximum context length of our student model.

ACL Anthology Network Corpus citations

The ACL Anthology Network Corpus citations dataset [Xuhui Zhou, 2020] (denoted as AAN) is a citation prediciton dataset. Each example in the dataset

[2] `arxiv.org`

contains a pair of paper's abstracts and is classified as positive if the first document cites the second one, or negative if it does not. The dataset is compiled from the ACL Anothology Network Corpus [Radev et al., 2013], where each source paper creates positive pairs with all cited papers and negative pair with one other randomly sampled non-cited paper. The focus of this dataset is on very short documents; just around 4% of documents are longer than 256 tokens. AAN has the shortest documents of all datasets in our evaluation set.

Semantic Scholar Open Corpus citations

The Semantic Scholar Open Corpus citations dataset [Xuhui Zhou, 2020] (denoted as OC) is also a citation prediction dataset in the same format as AAN. As the dataset name suggests it was compiled from the Semantic Scholar Open Corpus [Bhagavatula et al., 2018]. In this dataset, only a single positive pair is generated for each source paper, which results in much higher count of unique papers in the dataset compared to AAN. OC contains relatively shorter sequences with only 12% of documents larger than the maximum context of our structural teacher.

Semantic Scholar Open Research Corpus citations

The Semantic Scholar Open Research Corpus citations dataset [Xuhui Zhou, 2020] (denoted as S2ORC) is our third and final citation prediciton dataset. The source of this dataset is the Semantic Scholar Open Research Corpus [Lo et al., 2019] where each paper is divided into sections that are connected via links to papers cited within the given sections. This structure is used to generate positive and negative pairs in the citation dataset S2ORC. A section is paired with an abstract of cited paper to create a positive pair and with an abstract of non-cited paper to create a negative pair.

The documents in S2ORC are comparatively longer than those of the previously mentioned tasks, except for ARXIV, with over 50% of document being longer than 256 tokens.

PAN plagiarism detection

The final classification dataset is the PAN plagiarism detection dataset [Xuhui Zhou, 2020] (denoted as PAN). It was constructed from PAN plagiarism alignment task [Potthast et al., 2013], which is a collection of pairs of web documents, where the sections relevant to plagiarism are humanly annotated both in the source as well as in the suspicious document. PAN is a binary classification task where each document pair is classified as positive or negative. Positive inputs pair the source's plagiarised section with a part of the suspicious document containing the corresponding plagiarised section. Negative pairs are constructed from the positives one by replacing the source's segment by different part of the same document that is not annotated as being plagiarised. PAN contains longer documents; around 60% are longer than the maximum context size of our structural teacher and just under 50% of document are longer than 512 tokens. PAN therefore creates an intermediate step between documents of medium length in tasks such as IMDB or S2ORC and extremely long documents in ARXIV.

Dataset	Documents	Sources	Targets per source	Tokens over 384	Tokens over 512
WINES	1 662	89	8.92±1.23	100.00%	90.43%
GAMES	21 228	88	8.74±2.35	100.00%	91.78%

Table 5.5: Statistics of our similarity-based evaluation tasks. Each dataset is composed of several source documents each with a set of similar target documents. Additionally we show the percentage of documents with SBERT tokens above a given threshold to highlight how many documents need to be truncated in order to be processed by our structural teacher.

5.2.2 Results

5.3 Similarity-based tasks

Besides from classification tasks we also evaluate our model on two tasks that are based on similarity metrics between two embeddings. Our evaluation set focuses on long Wikipedia articles from two different fields of interest. The statistics of similarity-based tasks are displayed in Table 5.5. The distribution of documents' lengths is plotted in Figure 5.2.

We use cosine as a measure of embeddings' closeness. Same as with the classification tasks, here we also do not finetune the embeddings to a given task. Consequently, we do not need training or validation splits and we can use the whole datasets for evaluation. Our main scoring metric is Mean Average Precision (*MAP*), but we also present Mean Reciprocal Rank (*MRR*). While MAP scores the whole predicted ordering, MRR is more interpretable and can be more important in scenarios where we care only about the first positive result.

5.3.1 Tasks' description

Wines and Video games Wikipedia articles

Both of our similarity-based tasks are datasets consisting of Wikipedia articles from two fields of interests: wines (denoted as WINES) and video games (denoted as GAMES) [Ginzburg et al., 2021]. Each dataset contains around 90 source articles each associated with around 10 similar articles. We find the two datasets very unique as they combine two aspects which are rarely seen together. First the similarities are based on human annotations and not on proxy measures such as common citations or outgoing links. Second the documents are relatively long with around 90% of documents being longer than 512 tokens.

5.3.2 Results

Conclusion

- what i have done — what are the model's results
- other findings

Bibliography

Waleed Ammar, Dirk Groeneveld, Chandra Bhagavatula, Iz Beltagy, Miles Crawford, Doug Downey, Jason Dunkelberger, Ahmed Elgohary, Sergey Feldman, Vu Ha, et al. Construction of the literature graph in semantic scholar. *arXiv preprint arXiv:1805.02262*, 2018.

Iz Beltagy, Kyle Lo, and Arman Cohan. Scibert: A pretrained language model for scientific text. *arXiv preprint arXiv:1903.10676*, 2019.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Chandra Bhagavatula, Sergey Feldman, Russell Power, and Waleed Ammar. Content-based citation recommendation. *arXiv preprint arXiv:1802.08301*, 2018.

David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341*, 2019.

Arman Cohan, Sergey Feldman, Iz Beltagy, Doug Downey, and Daniel S Weld. Specter: Document-level representation learning using citation-informed transformers. *arXiv preprint arXiv:2004.07180*, 2020.

Andrew M Dai, Christopher Olah, and Quoc V Le. Document embedding with paragraph vectors. *arXiv preprint arXiv:1507.07998*, 2015.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

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

Dvir Ginzburg, Itzik Malkiel, Oren Barkan, Avi Caciularu, and Noam Koenigstein. Self-supervised document similarity ranking via contextualized language models and hierarchical inference. *arXiv preprint arXiv:2106.01186*, 2021.

Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

Jun He, Liqun Wang, Liu Liu, Jiao Feng, and Hao Wu. Long document classification from local word glimpses via recurrent attention learning. *IEEE Access*, 7:40707–40718, 2019. doi: 10.1109/ACCESS.2019.2907992.

Harold Hotelling. Relations between two sets of variates. In *Breakthroughs in statistics: methodology and distribution*, pages 162–190. Springer, 1992.

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

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*, 2021.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. Revealing the dark secrets of bert. *arXiv preprint arXiv:1908.08593*, 2019.

Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*, 2016.

Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

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

Kyle Lo, Lucy Lu Wang, Mark Neumann, Rodney Kinney, and Dan S Weld. S2orc: The semantic scholar open research corpus. *arXiv preprint arXiv:1911.02782*, 2019.

Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150, 2011.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

Malte Ostendorff, Nils Rethmeier, Isabelle Augenstein, Bela Gipp, and Georg Rehm. Neighborhood contrastive learning for scientific document representations with citation embeddings. *arXiv preprint arXiv:2202.06671*, 2022.

Martin Potthast, Matthias Hagen, Tim Gollub, Martin Tippmann, Johannes Kiesel, Paolo Rosso, Efstathios Stamatatos, and Benno Stein. Overview of the 5th international competition on plagiarism detection. In *CLEF Conference on Multilingual and Multimodal Information Access Evaluation*, pages 301–331. CELCT, 2013.

Markus N Rabe and Charles Staats. Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021.

Dragomir R Radev, Pradeep Muthukrishnan, Vahed Qazvinian, and Amjad Abu-Jbara. The acl anthology network corpus. *Language Resources and Evaluation*, 47:919–944, 2013.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

Nils Reimers and Iryna Gurevych. Making monolingual sentence embeddings multilingual using knowledge distillation. *arXiv preprint arXiv:2004.09813*, 2020.

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

Amanpreet Singh, Mike D'Arcy, Arman Cohan, Doug Downey, and Sergey Feldman. Scirepeval: A multi-format benchmark for scientific document representations. *arXiv preprint arXiv:2211.13308*, 2022.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems*, 33:16857–16867, 2020.

Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers, 2019.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6), dec 2022. ISSN 0360-0300. doi: 10.1145/3530811. URL `https://doi.org/10.1145/3530811`.

Trieu H Trinh and Quoc V Le. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*, 2018.

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

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

Noah A. Smith Xuhui Zhou, Nikolaos Pappas. Multilevel text alignment with cross-document attention. In *EMNLP*, 2020.

Liu Yang, Mingyang Zhang, Cheng Li, Michael Bendersky, and Marc Najork. Beyond 512 tokens: Siamese multi-depth transformer-based hierarchical encoder for long-form document matching. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 1725–1734, 2020.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

Hamed Zamani, Mostafa Dehghani, W Bruce Croft, Erik Learned-Miller, and Jaap Kamps. From neural re-ranking to neural ranking: Learning a sparse representation for inverted indexing. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pages 497–506, 2018.

Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. Defending against neural fake news. *Advances in neural information processing systems*, 32, 2019.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

List of Figures

List of Tables

A. Attachments