Institut für Informatik

Reproducibility of SciBERT

Jan-Niklas Weder

Modul : Data Mining

Contents

Introduction

We all interact with countless different computer systems every day. These interactions are diverse, but all require that the human understands what the computer or software system is trying to communicate, and the same must also be true for the other direction. This problem exists in many different forms and is not exclusive to a human-machine interaction. For example, two people will have problems communicating if they do not know how to communicate with one another. But going back to computer systems and humans, these interactions can take place in different ways. For example, a relatively common way is to use something like a keyboard and a mouse or touchscreen to interact with the system and receive feedback from it through a display. With such communication channels the foundation has already been laid, but there is still the problem that the computer and the human being can only communicate by means of a common language. Thus a computer cannot evaluate an input, which is not known to it. This applies of course in both directions. If we take this idea a step further and move from direct interaction with a system to human-generated texts, the problem is similar to the one described above, but on a larger scale. Now the question arises why a machine should be able to understand texts that were previously created by a human, but here we can take a look at search engines as a frequently encountered example from our everyday life. If a search engine could understand the texts that it knows just as well as a human being, it could use this understanding to derive its results from a query and would not need to take the detour over other statistically motivated procedures. If an understanding of a given document were present, even more complex tasks could be solved by the computer. For a more intuitive understanding of the difference, imagine two people receiving the same text, but only one of them has knowledge of the language in which the text was written, and now the difference in the possible questions these people are capable to answer about that text is relatively obvious.

There are already models that can interpret human language and translate it into their own representation, for example BERT [5], which we will look at in more detail later. However as a brief overview, the idea behind BERT is to interpret a word depending on its context in a text. This seems relatively trivial at first, but creates some relatively difficult challenges and problems.

If we now think back to the example just above and now two people are given the same task for an identical text, we would expect two different answers and exactly here lies the problem, if we now turn this procedure around and these two people would each produce texts based on the same intention and even the same knowledge, we would again obtain two different texts. But ideally these would be interpreted identically by a human as well as by a machine. This is already rather unlikely if not impossible for humans and poses a similar problem for computer based solutions and different languages have not even been considered yet. One of the problems is the different vocabulary that different people might resort to. Exactly at this problem SciBERT [2] comes into play. SciBERT uses the ideas that BERT is based on and tries to apply them to a new domain. We will also go into more detail about SciBERT, but it can be

mentioned here already that the vocabulary used by SciBERT differs from that of BERT by about 58%. [2]

In this paper we will be using the SciBERT model to solve a specific task based on a data set. The task will be a relation classifiaction or REL in short and the underlying dataset will be Chemprot.[12] Both the data set and the task will be discussed in more detail at a later stage. To solve this task, the basic model of SciBERT will be reused and the adaptations for the specific task will be rebuilt based on the original paper. The REL task was chosen because it seemed to be relatively well described in the programming language used here, which is Julia, and was therefore suitable to be rebuilt. This particular dataset was chosen because it can be used for REL tasks and because it is publicly available in its original version. This means that no registration or anything like that is required to obtain it, and it was supplied by the original authors in a preprocessed form intended to be used with the SciBERT model, which means that relatively little customization was required to use it.

Another aspect was that this part should run relatively well on regular hardware and in comparison to the actual core of the orginal paper, meaning SciBERT itself, it should also be computable in a reasonable time range with hardware that could be expected in a normal desktop computer. SciBERT itself took about a week to be fully trained resulting in a base model that could be further trained to be used for more specific tasks. Notably, this week involved hardware beyond what is normally found in a desktop computer.

The focus here will be to examine individual parameters that were not investigated further in the original paper but were mostly chosen on the basis of previous work or knowledge, or that were investigated further but their effects were not reported in the paper. We will investigate that in order to be able to assess the choice of these parameters more accurately. Two parameters are thereby in the foreground: the epoch and the learning rate. Both parameters were investigated but the best performing parameters for each task were used later in the original paper. However, there is no information provided on which parameters these were and how they performed in comparison to the other tested ones. Exactly these missing information in the original paper are examined here together with their influence on the result. Furthermore, the parameter ranges from the original paper will be expanded to give a better overall picture of their impact on the overall performance. Another aspect of this paper will be the implementation of the described project in the programming language Julia and the consideration which hardware is necessary and consequently what time periods are necessary for different hardware platforms.

State of the art

In the following, we will first look at BERT in order to have the basis for all further approaches that will be considered here. BERT is followed by BioBERT, the first variation of BERT that we will look at. These two approaches are two approaches underlying SciBERT, in the sense that they were developed before SciBERT. After BioBERT we will briefly look at S2ORC-BERT which targets a specific domain similar to BioBERT. Last but not least comes AOG-BERT which tries, with a more extensive modification to BERT, to convey further information which otherwise are not explicitly considered in the BERT model. After these approaches we will turn to SciBERT.

2.1 BERT

BERT is a modern approach to create a model that can take into account the context of a word in both directions. Thus, the model would be better able to interpret words through their context. Based on this architecture, a model is then pretrained on a corpus using masklm and nextsentence tasks. This base model can then be used for new tasks with relatively little effort. Tokens denote a representation for individual words that the model knows. These tokens can already be seen in the figure 2.1.

The masklm task consists of masking random words in a sentence and having the model guess what the masked word actually was. This task is commonly referred to as masked ML task or MLM for short. For training the original BERT model, about 15% of all input tokens are masked and by predicting the missing words, BERT is supposed to learn an understanding of natural language. In the figure 2.1 this would lead to the situation that some tokens of the lowest row are masked. [5]

The Next Sentence Prediction task consists of inferring relationships between two sentences. Because this task is not covered by MLM. The Next Sentence prediction, or NSP, is designed to enable the model to acquire an understanding of complete sentences. This is accomplished by a classification task that requires the model to predict whether two given sentences follow each other. Therefore, the CLS tokens are used for this task. Especially QA and NLI are supposed to benefit from this training. [5]

These two tasks were then used to train BERT on the BooksCorpus and on English Wikipedia texts. At this point, a pretrained model is obtained, which can then be used in various ways depending on the particular task. Further training on a a new task is referred to as finetuning. [5]

Since finetuning tasks are quite specific, the details depend on the selected task. However, in general, the procedure can be classified into one of two groups. First, there are classification tasks based on complete sentences. These generally use the representation of the CLS token with a classifier layer. The other group of tasks operates at the token level and therefore uses the representation of the tokens using one or more additional layers. Both types of tokens can be seen in the figure 2.1. Once the architecture is set, the finetuning can take place. This is done end to end, so the whole transformer, i.e. BERT and the task specific part are tuned together. If BERT is not supposed to be changed, there is generally the possibility to freeze

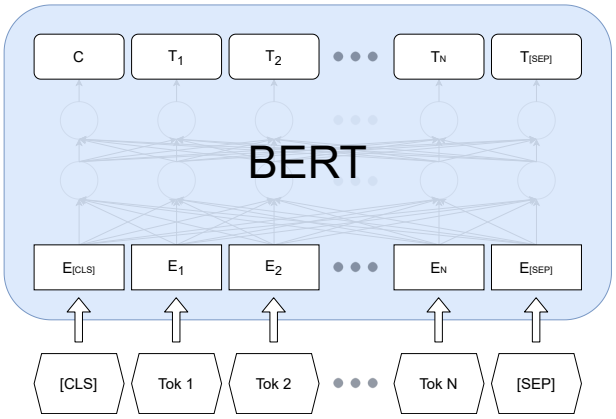

Figure 2.1: Schematic illustration of BERTs architecture.
Here the inputs and outputs that bert expects and delivers can be seen. Below the blue highlighted area we see the already tokenized words enclosed by the CLS and SEP token. These are now processed by BERT and one receives for each token the representation created by BERT visible here in the uppermost line. In between are representations that BERT uses as a kind of intermediate result.
Source: Adapted from [5]

specific layers and thus prevent the change of weights in these. This leads to the fact that for example only the task specific part of the transformer is trained.

2.2 BioBERT

BioBERT can be viewed as an extension of BERT that emerged because BERT alone was not yielding the desired results in the biomedical domain. This observation has often been associated with the differences in word distributions between a general domain such as Wikipedia, on which BERT had been trained, and the highly specialized words that are frequently used or used only in the corresponding domain.[5, 9, 10] This difference in the underlying corpora not only implies that adaptations in the architecture of the model itself might be necessary, but moreover implies a possible need for adaptations in the vocabulary and tokenizer being used. [9]
Based on the knowledge that the vocabulary may differ significantly, it was hypothesized that a model that takes these features into account should perform significantly better on tasks within that specific domain than models that are more general and may not even have seen that domain previously. Based on this hypothesis, BioBERT was created. A model that is supposed to be better adapted to the biomedical domain than BERT. [9]
Nevertheless, BioBERT itself is just a further trained version of BERT. This means that the original BERT model was used as a base and further trained on either PubMed abstracts, PubMed Central full-text articles, or both. Therefore, the authors chose to retain the original BERT vocabulary in order to use the pre-trained version of BERT as a foundation. This had the advantage that the original model only needed to be further trained on the new corpus and the required training time could be reduced significant. For the tokenizer WordPiece was used to handle the problem with unknown words. It was also considered to create a new vocabulary, but this idea was discarded in order to preserve the previously mentioned advantage of being able to utilize the pre-trained BERT model to save resources. As a result, it was observed that BioBERT performed better than BERT in version 1.0 that was based on PubMed abstracts and the full text articles of PubMed Central biomedical landscape with very few exceptions. Although the actual measured improvements sometimes vary quite significantly, it can be safely concluded that even continued training of BERT on a biomedical corpus can lead to significant improvements on tasks based in this domain. [9]

2.3 S2ORC-BERT

S2ORC, known as the Semantic Scholar Open Research Corpus, is a corpus that contains a large number of papers as well as metadata and the references associated with the papers. According to the authors, the full text portion of the corpus alone is the largest structured academic text corpus available in April 2020.[11] This corpus is based on semantic scholar

papers and thus comes from several different origins. Although the main part of the paper is the acquisition and processing of papers and the resulting construction of the S2ORC corpus, the authors also use that corpus to continue training BERT. This is relatively close to the basic idea and approach behind BioBERT as well, but in this case is even closer to the approach used for SciBERT. [2, 11]

We will take a closer look at SciBERT at a later point in time. S2ORC-BERT, as the model trained here is called, unlike BioBERT, is trained from scratch and is therefore not based on the pretrained BERT model. Some important features are that the loss is calculated by cross entropy and the optimizer is Adam. The learning rate as well as the number of epochs per task are derived from two predefined sets. Here the combination is chosen that gives the best result for the respective task according to the development set. Since the S2ORC-BERT is rather used for validation of the corpus and clearly less focused on a new architecture for BERT, many parameters are identical to those of other papers, in particular the values reported for SciBERT were used.[11] It should be mentioned that the difference between this model, BERT and SciBERT is almost exclusively due to the underlying corpus and the associated vocabulary.

2.4 OAG-BERT

Another approach that not only seeks to better match the corpus on which BERT is trained to the corresponding domain, but also simultaneously attempts to teach the model a more extensive understanding of the texts. With this further goal, however, the training strategy underlying BERT also needs to be adapted, because as we saw earlier, BERT's pretraining is based only on the MLM task and a NSP task. Although these two tasks together ensure that we obtain a reliable initial model, the question remains open whether specific tasks can be solved better by more directed training in the pretraining phase or by embedding additional information. [5, 10]

Based on this, one could argue that a model with domain entity knowledge might perform better than a model that does not utilize such information. For example, Liu et al. argue that specific institutes may have a focus on certain scientific areas and that this knowledge has the potential to support the model in assigning a paper to a research area.[10] While one might generally assume that the actual text should also provide enough information to assign a paper to research areas, this knowledge could still turn out to improve model performance.

Compared to existing approaches, OAG-BERT now tries to incorporate so-called non-homogeneous knowledge from the Open Academic Graph into the model. Here, non-homogeneous knowledge refers to information about authors, fields of study, venues and affiliation. To integrate this information into the model 3 crucial changes are made compared to BERT. [10]

The first adaptation is the heterogeneous entity type embedding. Here the token type embeddings are replaced by entity type embeddings, so that the different information types receive unique labels, through which it is possible to identify which input belongs to which group. [10]

The next adaptation is the Entity-aware 2D-positional encoding. Just like BERT, OAG-BERT needs positional embeddings to encode the sequence order. However, BERT does not have the ability to distinguish between two adjacent entities and would interpret them as a single one. To solve this problem, the positional embeddings in OAG-BERT are two-dimensional and encode the so-called inter-entity sequence in the first dimension and the intra-entity sequence in the second dimension. [5, 10]

The last change is the span-aware entity masking. Although the masking strategy is not changed for the abstract or the actual text, a special masking is proposed for the new entity types. This is supposed to help OAG-BERT to learn complex entities especially if they consist of many tokens. So an entity consisting of four or less tokens will be completely masked, but as soon as it is longer only a part of the entity will be masked. The length of the mask is thereby drawn from a geometrical distribution. The training can be grouped in two parts, the first part is only trained with normal texts. That means only abstract, main text and the title is used. The model obtained after this training is referred to by the authors as

vanilla OAG-BERT. This is in contrast to the full OAG-BERT model. To obtain this, the vanilla model was now further trained using the heterogeneous entity information. [10]

In comparison with, for example, SciBERT, which was chosen by the authors for comparison, OAG-BERT performs very well in tasks involving the entities that are included under the term heterogeneous entities. However, it also shows that the gap between the two models can be greatly reduced by finetuning the entire SciBERT model. Thus, after finetuning, OAG-BERT can only significantly outperform in Venue and Affiliation and performs about as well as SciBERT in the field of study category.[10]

SciBert

SciBERT uses the original BERT architecture and thus incorporates adaptability and broad applicability.[2, 5] SciBERT is a version of BERT trained on a different corpus. This is intended to make SciBERT better at understanding natural language that originates from the domain it is intended for. More specifically, in this case, the biomedical and computer science domain. In the original paper, the authors presented four different versions of SciBERT, two that use the standard vocabulary of BERT and two that use a specialized vocabulary to better represent the vocabulary of the corpus. It is worth noting that the two different vocabularies overlap by only about 42%.This shows how different the two underlying corpora are, and therefore suggests that the SciBERT versions using this adapted vocabulary may perform significantly differently than the standard BERT model simply because of this more adapted vocabulary for the given domain already, and this might also be the case when compared to the SciBERT model using the vocabulary from BERT. [2]

While the architecture used in the SciBERT model is described in great detail, as well as the changes made for specific tasks, there are some missing pieces of information that underlie certain decisions, and some decisions were made based on previous models and are therefore not necessarily transferable to this new model without some restrictions.

The first thing to mention here is that the corpus on which the model is trained is not specified precisely enough or that the actual data used cannot be traced. Only the source of the data was described, but not the selected data [2], and this information does not seem to be reproducible at all.

Another example would be the architecture for the tasks themselves. This is determined without considering other possibilities that could potentially produce similar or even better results. Likewise, the optimizer, dropout, and loss function are referenced from another paper without considering the possibility that an alternative might be more appropriate. A similar situation can be observed for the batch size and the choice concerning the number of epochs and the learning rates being considered. While several options are mentioned for epochs and learning rates, this is followed by settling for the best epoch and learn rate in each case. Furthermore, the decisions regarding learn rate and epoch are not discussed further and thus the information on the exact influence of the learning rate and the number of epochs is missing. Another aspect would be that of tasks and grades. Here, some frequently used tasks were used without further explanation. In addition, the selected scores are based solely on other work and are not further justified or extended by other possible scores. Also, only one score is described here, which is supposed to be an average of several attempts, and it would have been more useful to include information about how much the scores vary.

In this paper, we will focus on the influence of the number of epochs on the score as well as on the change of the loss over time. In particular, we will look into how the model changes from epoch to epoch. At the same time we explore whether the range of two to four epochs considered in the original paper is sufficient or whether more epochs might be reasonable. To be more precise, we will look at the range of one to ten epochs and consider the changes in the loss and

the score. This will then be extended using different learning rates to show how these affect the speed of convergence of the model.

Experiments

In this section we will look at replication in more detail and discuss the approach and possible deviations from the original paper. So first comes the basic structure, followed by going through some examples that show us what the functions do and if our implementations work as expected. After that we will briefly talk about the hardware that was used. This is followed by a more detailed discussion of our implementation. Then we will look at some of the more serious problems and finally we will look at the obtained results.

Since the basic architecture for the underlying BERT model remains the same and does not change, we will only look at the task specific part. Furthermore, we restrict ourselves to the classification tasks and, to be more precise, to the REL task. The implementation consists of four main parts. They are the data acquisition and preprocessing, the transformer itself and last but not least the evaluation of the model with the help of a score. Here we use the F1 score to follow the original paper in this area as well.

Simple tests and examples

To ensure that these individual areas function correctly and that errors have not already sneaked in. There are several outputs that show the functionality in a small scale and can ideally catch some errors. These outputs can be found in the Jupyter Notebook under Tests. In this section, the first record and the corresponding label of the training dataset are displayed first. The same is repeated for the test data set. After these examples, a dry run of the tokenizer follows. This is to assure us that the tokenize function, which puts the data into a form that the transformer can read, is working correctly. This is shown with a sentence that leads to the same result both by hand and by the function. Next, the tokenization of special tokens is examined. This is to guarantee that the special tokens needed for the NER task are correctly recognized and translated. This is followed by some examples that test the functionality of the loss function and the embedding of the previously translated sentence. At this point, however, the verification of correctness becomes difficult, since the interpretation of the representation of the CLS token is no longer directly evaluable for humans. Although it can be shown that this process functions and presumably also operates correctly, this example no longer guarantees this. This is followed by the tests that refer to the evaluation, in other words to the F1 score. Here we first consider whether the formats have been converted correctly so that the Metric package which is used for the calculation can read them and then we briefly compare whether the example delivers the expected F1 score.

Hardware requirements

In this section, we will take a brief look at the usability of different hardware platforms for creating transformer models and training or testing them. More specifically, we will compare the google-colab environment with an Nvidia GPU and an AMD GPU. Due to the randomness of the hardware assignment on the google-colab page, the used GPU can not be defined more precisely. The utilized Nvidia GPU was a GeForce 940MX with about 2 GB of VRAM, and

the AMD GPU on the other side was an RX580 with about 8 GB of VRAM.

At this point the extent to which AMD's ROCM stack [1] is usable will be briefly described, because surprisingly we were able to define the model and make predictions with it in a newly created state. Unfortunately, due to instabilities in the ROCM stack after an update that must have broken some internal dependencies that the kernel and ROCM driver must have relied on, the Linux kernel could no longer use the GPU, and so the video output of the computer was unusable.

Even though this shows that an AMD GPU is actually capable of running the Transformers.jl [4] package and loading at least a newly defined model. Even though I cannot reveal whether the model could be trained or otherwise used further. This fact in itself is surprising, since AMD itself describes the support status of the RX580 as only potentially possible [1], and Julia describes AMD GPU support as experimental. [3]

However, everyone should be mindful not to use the ROCM-Stack on a productive system due to its instability, but rather only in a virtualized environment or on systems where the unusability of graphics cards will not impede the use of the system.

This brings us to the GeForce 940MX. This graphics card unfortunately reaches its limits due to its memory size. 2GB of VRAM, of which even a little bit less is usable, does not suffice for the used BERT model nor for SciBERT and thus results in an out of memory error. Therefore, this hardware will not be considered further.

Thus only the google-colab environment remains. This platform provides about 10 to 15 GB of VRAM. After the model has been moved to the graphics card, Julia shows an occupancy of just over 2 GB memory with the help of the memory indication of the CUDA driver. Here we can see why the 940MX was not able to load the model. The calculation of an epoch usually takes between 500 and 1000 seconds. These fluctuations could have several causes, but are most likely due to the google-colab environment and in particular the fact that several instances share the same graphics card. This is most likely the reason for the observed strong fluctuations from epoch to epoch. This assumption is particularly reasonable, since sometimes almost every epoch lasted the same time, apart from relatively small fluctuations of less than 50 seconds. Interestingly, the entire video memory is used for caching, even though the model consumes just over 2 GB of memory and the entire original chemprot dataset is about 5MB in size when unpacked. Still, the runtime required in the google-cab environment is about as expected and significantly faster than training on a CPU. The CPU, which was a Xeon E3-1230 v3 with four cores, needed about as long for a single step as the graphics card in the google-colab eviremont needed for an epoch. A step here consisted of only one sentence and the corresponding label.

Data preprocessing

Due to the availability of the datasets used by the original authors, we will use their prepared datasets, which are already prepared in such a way that they can be more easily used for training and still differ only slightly from the original datasets. The datasets we will use will be retrieved directly from the SciBERT GitHub page and will be made available through the DataDeps package [13], which provides an easy way to retrieve data that may or may not be available locally. If not already stored locally, it is cached in the local Julia path and within Julia, the DataDeps package provides the appropriate paths to the data and retrieves it from the defined source as needed. In addition, a hash can also be defined to ensure that the provided data is identical to the expected data.[13]

In the following section, we will take a closer look at the original data and the individual changes that were made to use these data sets for the training process.

Chemprot

The Chemprot data is provided in a JSON line file format. More precisely, each line consists of a text and the corresponding label. A field for metadata is also provided, but is usually not used. In its original format, the chemprot corpus consists of a develop, test, and train set, where the develop, test, and train folders correspond to the files of the same

name within the chemprot folder provided on SciBERT's GitHub page. The difference arises from the database-like structure in which the chemprot corpus is originally provided, those subdivided sets of information are, for example, the text itself that is in one file and the positions and annotations are in another. These subdivided information sets have been merged by Beltagy, Lo, and Cohan and are provided in a single file in the format mentioned earlier. [2, 12]

Since the data is already very close to the format that is needed for the transformer, only small changes have to be made. These consist mainly of not changing the special tokens that are already included in the data by the tokenizer or wordpiece. Afterwards these preprocessed sentences should only be changed by using the vocabulary function. This takes place in the tokenize function, which first locates the special tokens and finds out which pair comes first. This is followed by splitting the sentence at the locations of the special tokens. Now these subsets are processed by the tokenizer and by wordpiece and then reassembled together with the corresponding special tokens to form a whole sentence. After that the numeric representation is generated by the vocabulary and the segment representation is generated together with the corresponding mask. The segments always get ones, because we only process single sentences with this function and the mask is created with the mask function from the Transformers.jl [4] package. Now only the correct labels are missing. These are put into a 1 out of k scheme using the onehot function from Flux. [6, 7] For this function we need the correct label and an array with all existing labels. This array has already been defined based on all the labels found in the training data. With all this data we now have all the prerequisites to continue with the transformer.

Implementation of the replication

The first step, which belongs to the actual replication, is to rebuild the original transformer. For this we first need the basic model. We use Transformers.jl[4] for this task and some others that are still to come. This package includes some very helpful functions for handling transformers. These range from loss functions to wrappers that make it much easier to create new transformers. We therefore use Transformers.jl to load the base model and use set Classifier to replace the layers previously required for NSP and MLM with the ones we need. For the REL task, this means that the actual BERT architecture, as shown in the figure 2.1, should get an additional classifiaction layer. Here we use a dropout of 0.1 as described in the original paper. At this point we deviate somewhat from the original paper, which describes that the CLS layer is passed directly to the classifier layer.[2] We realize this by using the loss function, which only passes forward the representation of the CLS token to the classification layer and not the representations of all individual tokens. We will discuss the loss function in more detail in the following. Another remark would be that here the classification layer is realized by a Dense layer using the Flux library. This layer can be seen as a fully connected one. Now the optimizer is missing, which the same as in the orginal one namely Adam. The batch size will be set to 1 due to the implementation here. This differs from the original paper which used 32.[2] The main reason for this decision was that it created some problems using mini batches and so far not all of them could be solved. For this reason all following results will use a batch size of 1.

With this we move on to the loss function. This not only takes over the task of an actual loss function, in other words to calculate an error between the prediction of the model and the real label, but also deals with how the data flows through the model. This can be seen by the fact that both the embed and transform steps are done explicitly here. This allows us to simply call our loss function with the corresponding label and dataset and get the corresponding value for the loss. For the actual loss calculation, we use the logitcrossentropy as opposed to the crossentropy that was used in the original paper.[2] This decision was made because the crossentropy functions here expect values between 1 and 0 that sum up to 1, and this would be achieved, for example, by using a softmax layer. However, performing these two steps by hand is numerically more unstable than using the logitcrossentropy function that incorporates these two functions. [6, 7]

Apart from the loss, we need one more function, which is the function that computes a score for the task. In the case of the REL tasks this will be the F1 micro score. We calculate this score using the Metrics.jl [8] library. For this we need to collect all the predictions for the test data, the predictions themselves are analogous to the process described for the loss function. Furthermore we need the real label and this combination of label and prediction must now be converted into a certain format, from this the F1 micro score can be calculated directly using the 'f_beta_score' function from the Metrics package. [8] Now only the actual training is missing. For this two loops are used, one corresponding to the epochs and the other to the index in the training dataset. At this point different statistics are calculated and saved for later analysis. These include memory usage, runtime and F1 score per epoch as well as loss for each training step.

This training process is then performed for several models and for different learning rates. This leads us then to our results.

Complications

In the following, we briefly review some of the problems we have encountered during this work. The order follows that of the information flow in the model, starting with the input data and ending with the output.

After this sequence the first problem is to transform the data, which contains already the two text sections that are marked with the special tokens "[<<]", "[>>]", "[[[ ]" and "[ ]] ", these sentences must be processed by the tokenizer and by wordpiece, but without changing the mentioned tokens. If the individual sentences were processed without further precautions, the special tokens would be modified as well. However, due to this alteration, these tokens would no longer be interpreted as special tokens by the vocabulary, but would be seen as "normal" words. To avoid this, the tokenize function was written that splits the text into partitions and processes them individually using the tokenizer and wordpiece. This was necessary primarily because partitions identified by a pair of corresponding special tokens might not have the same length after processing. Due to this potential variation in segment length, it is not directly possible to find the new position of the tokens based on the old one and simply add them manually. Instead, the individual sections separated by the special tokens are processed one by one and then the special tokens are placed in between the corresponding segments. This procedure is based on the addition of the "[CLS]" and "[SEP]" tokens in the CLS, but extends it.

The next problem follows directly the one described before. The individual records of the data sets sometimes consist of characters that do not occupy only one code unit and so problems can occur when the end of a string is expected to be at the position of the length of the string. A code unit here describes a fixed size in memory that is generally required for encoding a single character. Fortunately, an error is generated when attempting to create a view of a string that ends or begins in the middle of a character that spans multiple code units. Although the error is relatively vague, a closer look at the string quickly reveals that the error occurs near special characters. With this knowledge, because of the different number of code units sometimes required to encode special characters, one can quickly find the explanation and underlying problem in the appropriate documentation. Here the functions 'firstindex' and 'lastindex' or the function 'split' can help. All these functions take into account the differences between position and code unit.

Another, more general problem was that some packages provide very few examples of how they should be used, and at the same time are not sufficiently detailed in the specifications. This leaves unanswered questions that initially prevent the packages from being used. For example, the MLBase package provides many different metrics, but requires an object of type ROCNums for computation. This type can be generated by the MLBase package, but the documentation only says that a ground truth and the predictions of the model are needed as input. However, the documentation for the actual ROCNum type says nothing about the format in which these two pieces of information must be provided, and there is nothing at all about the ROCNum type in

the examples. With a little luck, however, one comes across the documentation of the Confusion matrix, which contains an example of how to create a ROCNum object. Using this example, one can then derive the correct formats. However, missing information of this type can be found multiple times in diffrent packages and requires either some luck to find a working example, or trial and error to eventually hopefully figure out the correct format. Another example of such a problem, was the gradient calculation. In the official documentation of the Transformers.jl package there are still outdated examples that no longer work, and so you are quickly faced with the problem of not knowing why a certain part doesn't work. To be exact, it was relatively easy to find out which part of the code is not working as it should, but a very big problem was the assumption that the not working part should work, because it was only a function call and it corresponded one to one to the official documentation. Only later by a forum contribution of the developer himself it became clear that this function does not supply the necessary information any more and thus in our case the update step did nothing. A warning would have been appropriate here that the function call does not work anymore. In this case, a note that some information in the documentations are still missing or a hint that the documentation or the examples do not work anymore because they are outdated would have been very helpful. Especially since one example seems to work, it would have been important for it to stand out in some way from the ones that are no longer working, and to not give the impression that they all are still up to date.

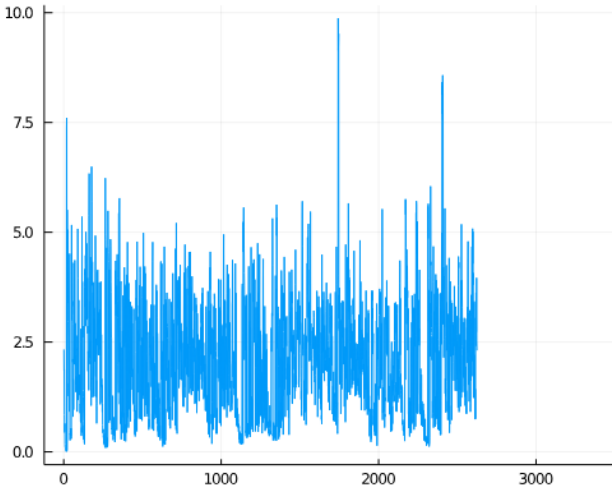

Figure 4.1: The calculated loss for the REL experiment using SciBERT with the scivocab vocabulary. Each x-axis step represents one training step. It can be seen that the loss does not decrease and is no longer displayed from a little more than 2500 steps. This is due to the fact that the calculated loss results in NaNs from this point on.

Observed results

Now we briefly come to the results we observed. Here it is noticeable that after a few thousand steps only NaNs are obtained as loss. As soon as NaNs appear, they should result in the fact that no further training is possible, since the training is based on the difference between the calculated result and the expected result. If this difference cannot be determined, no further update step will be possible. The NaNs can also be seen in figure 4.1 .There they are represented by the suddenly stopping curve, which should show a few thousand additional steps, but the plot function does not visualize NaNs. The NaNs that are generated here seem to be created internally within the CUDA environment. Especially since all inputs were considered exemplary for the model and they seem to be correct so far, it is reasonable to assume that there might be an incompatibility between the software components causing the NaNs here. Since some parts of the software packages used here are still under very active development, this might have resulted in bugs sneaking in. On the other hand it could be a problem caused by outdated software, for example since we have no direct control over the software used in the Google-Colab environment this is another possible source of errors. Another possibility would be numerical instabilities resulting in NaNs. However, this seems relatively unlikely since explicit approaches were used here to fix exactly these numerical weaknesses. Last but not least, there could also be an error in the implementation. This possibility is

opposed by the individual intermediate steps, which exemplify the correct functionality. In particular the fact that the input, which the model receives, can be directly observed and no obvious errors are found there.

Another important observation here is that the NaNs are already present in the model, i.e. do not arise from the loss function, but are already present in the last layer of our model and are only transported further from here. This again speaks for the first or last assumption.

Discussion

As already seen in the state of the art section, there are already improvements since the publication of SciBERT, but even the more recent models are definitely not yet performing perfectly, which is probably not a realistic expectation, but still they show potential for improvement in terms of their scores in the various NLP tasks. If one briefly deviates from the evaluation by means of score at this point and takes a look at the efficiency and necessary resources, an even greater deficit becomes apparent than the actual performance in the NLP tasks. In particular, these high demands on the available hardware used generally makes it difficult to conduct research in this area. Here, this was solved using google-colab environment, which in turn has its own weaknesses, especially due to restrictions that limit longer-lasting training times.

But now we are getting back to the actual scores and here especially S2ORC-BERT showed the impact that a larger corpus can have. In particular, the comparison between OAG-BERT and S2ORC shows that a more complex model is not always necessary to perform better on NLP tasks. However, OAG-BERT allows us to incorporate more information into the underlying BERT model, which turns out to be useful for tasks that can benefit from such knowledge. Here it can be summarized that there is not a perfect model but one has to decide depending on the area of application. Nevertheless, it can be said that there are advancements to models that could replace the previous one. For example S2ORC-BERT, which can generally be used instead of SciBERT. However, it must be noted that S2ORC-BERT is almost always better than SciBERT, even though S2ORC-BERT covers a much wider range of domains and is not as precisely tuned as SciBERT to the domains that are task relevant. While the difference in F1 score is usually small, one might expect S2ORC-BERT to perform similarly well in other domains, whereas SciBERT might need more data and longer finetuning to come close to S2ORC-BERT performance.

For the replication it can be summarized that especially the documentation and the examples for many Julia libraries used here are insufficient. Missing information may still be acceptable, but wrong or nonfunctioning examples will quickly lead users astray. Especially if currently working and non working examples are not separated and are even in the same folder. Although tests often offer a working example, in order to explicitly consider only these, it must be known that the normal documentation is not always reliable. Furthermore, the problem with the available hardware became apparent. Depending on the individual equipment, it is extremely difficult to investigate models of this size effectively.

As a result of the replication related to the goals set at the beginning, it can be summarized that this was not successful and ended in the output of NaNs. Therefore, no statement can be made here about the parameters addressed at the beginning.

Further development

For future research two different goals can be concluded. On the one hand, for highly specific tasks, model extensions like those pursued with OAG-BERT may be advisable. On the other hand, models that are specialized in a certain domain seem to per-

form better than models that have not been trained in this domain. This became very clear by the differences between BioBERT, SciBERT and BERT. On the other hand, a model that does not specialize in a few domains, but instead has the later underlying domains as part of its training data set, may actually perform better than more specific ones. This would lead to the conclusion that in the future models should be trained on as general corpora as possible and then perform comparatively well with specialized models in a wide variety of environments. However, it should be noted that these would have the benefit of being able to perform comparably across various domains.

Appendices

A Data

The data used is based on the already processed data used in SciBERT [2] and is loaded and processed directly in the code. The underlying data can therefore be found here.

B Code

The complete code used for this project can be found here. The code makes use of the already mentioned Julia packages.

Bibliography

[1] Advanced Micro Devices. *ROCm Version 4.0.1*. GitHub. Jan. 2021. URL: https://github.com/RadeonOpenCompute/ROCm/tree/rocm-4.0.1.

[2] Iz Beltagy, Kyle Lo, and Arman Cohan. "SciBERT: A Pretrained Language Model for Scientific Text". In: *EMNLP 2019* (Mar. 26, 2019). arXiv: 1903.10676 [cs.CL].

[3] Tim Besard et al. *AMDGPU.jl Version 0.2.4*. GitHub. Mar. 2021. URL: https://github.com/JuliaGPU/AMDGPU.jl.

[4] Peter Cheng. *Transformers.jl Version 0.1.7*. GitHub. Oct. 2020. URL: https://github.com/chengchingwen/Transformers.jl.

[5] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[6] Michael Innes et al. "Fashionable Modelling with Flux". In: *CoRR* abs/1811.01457 (2018). arXiv: 1811.01457. URL: https://arxiv.org/abs/1811.01457.

[7] Mike Innes. "Flux: Elegant Machine Learning with Julia". In: *Journal of Open Source Software* (2018). DOI: 10.21105/joss.00602.

[8] Adarsh Kumar. *Metrics.jl Version 0.1.0*. GitHub. May 2020. URL: https://github.com/AdarshKumar712/Metrics.jl.

[9] Jinhyuk Lee et al. "BioBERT: a pre-trained biomedical language representation model for biomedical text mining". In: *Bioinformatics* (Sept. 2019). Ed. by Jonathan Wren. DOI: 10.1093/bioinformatics/btz682.

[10] Xiao Liu et al. *OAG-BERT: Pre-train Heterogeneous Entity-augmented Academic Language Models*. 2021. arXiv: 2103.02410 [cs.CL].

[11] Kyle Lo et al. "S2ORC: The Semantic Scholar Open Research Corpus". In: (Nov. 2019). arXiv: 1911.02782 [cs.CL].

[12] Qinghua Wang et al. "Overview of the interactive task in BioCreative V". In: *Database* 2016 (2016), baw119. DOI: 10.1093/database/baw119.

[13] Lyndon White et al. "DataDeps.jl: Repeatable Data Setup for Reproducible Data Science". In: *Journal of Open Research Software* 7 (2019). DOI: 10.5334/jors.244.