CSE 481N: NLP Capstone: Blog Post 7

Team name: bitmaps - **bi**narized **t**ransformers for **m**aking f**a**st **p**rediction**s**
List of members: Tobias Rohde, Anirudh Canumalla
GitHub URL: nlp-capstone

A New Dataset: BooksCorpus + Wikipedia

In past blog posts we made use of the PennTreeBank-3 dataset for training and evaluating various forms of BERT. However, the dataset is was difficult to work with since it was tokenized from the PTB Tokenizer, so we had to sanitize it first before tokenizing it again with the Bert tokenizer. This week, our main focus was preparing a better dataset. We decided to use a subset of the original pre-training data. This seemed like a good choice of dataset, since we are essentially performing transfer learning from the original weights. Furthermore, we found that there were pre-existing tools for preprocessing the data. Using a subset of a large dataset will also allow us to easily use more data in the future without having to acquire a larger dataset. The BERT pre-training data consists of BooksCorpus and English Wikipedia, which makes for about 19GB of pure textual data.

Getting and processing the data

We used script from NVIDIA for processing the data. NVIDIA provides a script that (1) downloads BooksCorpus and Wikipedia, (2) extracts the raw text (3) segments the data into lines (4) merges the two datasets (5) distributes the data into train and test shards (80% and 20%). Instead of using their code to download BooksCorpus, we used the copy provided by UW to speed up the progress. Processing this data took more than a day in compute time and was extremely memory and performance intensive.

Dataloader

After running the NVIDIA code, we had 256 shards of data for the training and test split. We wrote a custom data loader that reads in the shards and creates sequences of a given maximum length by concatenating sentences from the shards. The dataloader also tokenizes the data and converts tokens into indices we can feed into BERT. This is an extremely time consuming process that takes hours. Thus we implemented a simple cache that writes the resulting tensors of indices to disk and reads them back in if they exist. This speeds up the data loading process to just a matter of seconds. The dataloader is also responsible for randomly masking out tokens. This happens on the fly, as data is requested from the loader. Batching, shuffling etc. is taken care off for us by PyTorch.

Deciding on a subset

Initially we attempted to evaluate BERT on the entire testing data, but we quickly found this is infeasible. It would take about 6 hours to evaluate BERT on the training data and 32 hours for the training data. Gradient calculations and the backwards pass would increase this furthermore, making it infeasible to train for even a single epoch. From the very start we expected to use less data anyways since we are using the trained full-precision weights of BERT for initialization. Thus we decided to choose a subset of this data. We wanted to choose a subset size such that we can iterate through new ideas in a reasonable amount of time, while still having enough data to have the model learn. We decided that about 15 minutes per epoch (on a

GPU) is a good amount of time to quickly run experiments. We ran inference and training with different sizes of data and measured the training time. We found that using 65536 sequences of length 128 for training gave us the best balance of speed and amount of data. In terms of performance and speed, using for instance 32768 sequences of length 256 would be equivalent, but we decided to specifically use a sequence length of 128 since this was used in the BERT paper (in addition to sequences of length 512). For the test data, we chose a size of 8192 and the same sequences length. Running inference on a dataset of this size takes only about 30 seconds and in FP16 mode only 10 seconds (see later section). We did not include the results from these timing experiments here since they just served the purpose of selecting a dataset size and did not yield any new or interesting results.

Evaluating BERT on new data and analysis of performance

We evaluated the original BERT model on the new test data subset. In the first row of the table we included the performance of BERT on the old PennTreeBank-3 dataset for reference. The second row contains the results of BERT evaluated on the new data. Note that the cross entropy is significantly lower and accuracy and mean reciprocal ranking are significantly higher for the new data as compared to the old data. We believe that this has two main reasons. Mainly, BERT was pre-traind on this data and thus expected to perform well. Secondly, as mentioned earlier, the PTB data is not very clean and we had to manually de-tokenize it. Additionally, in our original implementation, many very short sequences were fed through the model due to the way we built batches from the data. With very short sequences, BERT lacks context and it is more difficult to predict masked tokens. For the current dataset, we carefully pack sentences into long sequences to minimize the amount of padding added for sequences shorter than 128.

Model	Cross Entropy	Accuracy	MRR	Time (ms)	Memory (MB)
BERT (Original, Old Data)	4.3050	0.4017	0.4750	1.4471	417
BERT (Original)	2.2025	0.6210	0.6968	2.9821	417
BERT (16-bit)	2.2024	0.6210	0.6968	1.2903	208

Table 1: Model Precision on the BooksCorpus + Wikipedia Dataset

16 bit (half-precision) floating point format

In our earlier blog posts we discovered that the Embedding matrix is a significant contributor to the overall memory usage, making it an important target for binarization. However, we confirmed in the baseline approaches that binarizing the Embeddings is extremely detrimental to performance [See Blog Post 5]. In order to still reduce memory usage, we attempted to represent the embedding parameters with 16-bit precision instead of the traditional 32-bit floating points numbers. This was difficult, since simply converting those parameters to 16-bit broke the inference since in PyTorch performing operations involving both 16-bit and 32-bit tensors does not work. Thus we attempted to turn the entire network to 16-bit precision, expecting an increase in cross entropy (and a decrease in accuracy). This result is shown in Table 1 above.

Performance of 16-Bit BERT

To our surprise, the 16-bit model performed virtually the same, in fact marginally better (which is likely due to chance). Cross entropy, accuracy and mean reciprocal ranking are almost unchanged. This was very surprising to us. We expected it to work reasonably well, but this had no effect on the performance of the

model. However, the memory usage was obviously reduced in half. It is still our goal to binarize this network, but now we know that we can change some weights to half-precision and thus reduce the memory usage in half without any additional work. If we are not able to perform binarization for the embedding layers, this is a great alternative.

Inference time of 16-bit BERT

The inference time after converting weights to 16-bit was reduced by more than a factor of 2. We believe that this is primarily because operations with smaller numbers are simply faster, but this also allows the GPU we are using to make use of its "Tensor cores", which are specifically optimized for multiply-accumulate operations of 16-bit floating point matrices. From now on, we will use 16-bit floating point numbers for all of our inference, since it is clearly much faster and has no impact on performance of the model.

Training with 16-bit floating point weights

After discovering that 16-bit weights work so well, we considered also using 16-bit weights during training, which would hopefully gives us similar speed improvements. However, we found online that for training the increase precision is necessary, since gradient updates can often times be small. Thus people keep 32-bit precision weights as "master weights" and update those during gradient descent instead of the 16-bit weights. This approach is very similar to our binary training approach, described in Blog Post 6. We decided that it is not worth to attempt to get mixed precision training to work, since then we would work with 1-bit, 16-bit and 32-bit weights at the same time and the time spent on implementing this might be longer than the time we save from the improved performance. However, we can make use of the 16-bit weights during training when performing knowledge transfer. Recall that the idea behind this was to use the original BERT to make predictions for the current batch of data and then use the output of each layer to penalize the output of each layer of the binarized BERT we are training. We can simply run the original BERT in 16-bit mode and then convert the output of each layer to 32-bit, which will speed up training significantly.

Initial training results on new data

Our initial training results seem promising. We trained our binarized BERT model with full-precision embeddings and no knowledge transfer (i.e $\lambda = 0$) for 7 epochs, during which the cross entropy loss went from an initial x to 5.4950. This is significantly better than what we had last week. Using the old data, the cross entropy went from an initial 9.1692 to 6.3053 after 16 epochs. We need to compare these numbers with care since they are from different datasets, but the overall lower cross entropy indicates that the higher quality of data we have now improves learning. We did not include the other metrics since we first want to train for more epochs and our main focus right now is to monitor and the reduce the cross entropy loss.

Next Steps and Conclusion

During this week, our efforts were primarily focused on data processing, which took much more time than anticipated. While our initial goal was to run more experiments, we decided that it is more important to finalize the data we want to use for training and testing. Originally, we only used the PennTreeBank-3 data since it was easy to access and it gave us a quick way of running experiments and testing some of our initial ideas. Now that we have settled on approaches we want to explore in depth, we had to go back and focus more on the data aspect. Getting the BooksCorpus and Wikipedia dataset was difficult. We used code by

NVIDIA for acquiring and processing the data, which was difficult to get to work since it required nvidia-docker. Running this code was extremely time consuming and we had to rerun it multiple times before getting it right. We also only manage to process this large dataset on one machine so far, so we haven't been able to run experiments on multiple machines yet. Our next goal is to get the dataset on multiple machines so that we can run experiments on our home computers and on the nlpg servers. Now that the data work is done, we focus entirely on training and tuning our main hyperparameters: learning rate (scheduling), L2 regularization and the λ's for knowledge transfer (see Blog Post 6). All the other hyperparameters of BERT remain fixed. We will also consider other binarization schemes, especially for approximating the derivative of the sign function. We are setting up a queue-like document where we write down sets of hyper parameters and modifications that we want to try and then each of us will run the experiments from this queue whenever possible.