CSE 481N: NLP Capstone: Blog Post 6

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

Scaled binary weights

Part of the reason why our naive binarization baseline approach did not work well was that the binary weights were poor approximations of the full-prevision weights. In this section we discuss a slightly more advanced approach that we adapted from the XNOR-Net paper.

Assume $W \in \mathbb{R}^n$ is a weight vector that we want to approximate with a binary vector $B \in \mathbb{B}^n$. $\mathbb{B}$ is just a set of two values. Additionally, we introduce a scaling factor $\alpha \in \mathbb{R}^+$. Ideally we want $\alpha B \approx W$. First we need to decide on two values to restrict our vector B to. We choose $\{-1, 1\}$, since with $\{0, 1\}$ we can only approximate vectors consisting of either positive or negative values well, since we have a single scaling factor α that is either positive or negative. Thus we assume $B \in \{-1, 1\}^n$. We will use the L_2-norm squared for minimization, so we wish to minimize:

$$\mathcal{L}(\alpha, B) = ||W - \alpha B||_2^2 = (W - \alpha B)^T (W - \alpha B) = W^T W - 2\alpha W^T B + \alpha^2 B^T B$$

Note that $B^T B = \sum_{i=1}^{n} B_i^2 = \sum_{i=1}^{n} 1 = n$ and that $W^T W$ is a constant and thus can be ignored in optimization. Thus in order to minimize $\mathcal{L}(\alpha, B)$, we need to minimize

$$-2\alpha W^T B + \alpha^2 n = -2\alpha W^T B + \alpha^2 n.$$

First we find the optimal choice of B. We can ignore $\alpha^2 n$ and since $\alpha > 0$, we have that $-2\alpha < 0$, so we need to chose B to maximize $W^T B$. Note that $W^T B = \sum_{i=1}^{n} W_i B_i$. We can choose each B_i independently to be -1 or 1. The optimal B_i is given by $\text{sign}(W_i)$ (where we arbitrarily chose $\text{sign}(0) = 1$), since then the product $W_i B_i$ is positive. Thus we have $B^* = \text{sign}(W)$. For finding the optimal α, we take the derivative:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = -2W^T B + 2\alpha n$$

We set $B = \text{sign}(W)$ as found above and note that $W^T \text{sign}(W) = \sum_{i=1}^{n} W_i \, \text{sign}(W_i) = \sum_{i=1}^{n} |W_i| = ||W||_1$. Using this and setting the derivative to 0, we get

$$-2||W||_1 + 2\alpha^* n = 0$$

and solving for α yields $\alpha^* = \frac{1}{n}||W||_1$. In summary, this optimization suggest a new binarization scheme. Namely, given a vector $W \in \mathbb{R}^n$, we set $B = \text{sign}(W)$ and $\alpha = \frac{1}{n}||W||_1$ and use $\alpha B = \frac{1}{n}||W||_1 \, \text{sign}(W)$ for the approximate weights. Note that this introduces an additional full-precision scalar per weight vector. However, this additional amount of memory is negligible when considering the overall memory usage of BERT.

Revisiting the previous baseline approaches

In this section, we repeat the experiments from the previous blogpost using the new scaled binary weights and briefly compare them to our previous unscaled approach. Particularly, we use the new binarization scheme to binarize the same subsets of BERT components as in last weeks experiments. We evaluate the different binarization approaches on the the Penn Tree Bank validation data. Tokens were masked out as in the BERT paper, as described in detail in Blog Post 5. The results are presented in the table below. For a description of the columns and the rows please refer to Blog Post 5, section "Baseline approaches and results".

Binarized Components	Cross Entropy	Accuracy	Mean Reciprocal Rank 32	Inference Time (ms)	Memory Usage (MB)
None (Orig. BERT)	4.3488	0.3986	0.4714	2.0188	420
All	11.3236	0.0079	0.0213	1.9760	13
Layers	9.1692	0.0274	0.0451	1.9865	103
Layers Except Self Attention	9.1608	0.0153	0.0522	1.9982	208
Layers Except FFNN	8.1667	0.0652	0.0911	2.0120	313

Inference time

It is important to note that inference time is currently not particularly useful. Instead of bits, we are using full-precision `FloatTensors` which are limited to taking on values in $\{-1, 1\}$. Thus we don't expect the inference times to decrease due to binarization. We continue to keep track of inference time for consistency with the experiments in the last blog post. However, note that the inference times here are all higher than previously, since these experiments were run on a GTX 1080 Ti as opposed to a TITAN RTX. This highlights an issue we have not addressed yet, which is to also factor in the speed of the GPU/CPU used for inference.

Key Takeaways

Overall the model still performed poorly. However, in some cases they are improvements over last week's baselines. Using the scaled binary weights, we saw improvements in accuracy and mean reciprocal rank for all approaches except for the the approach where we did not binarize self attention. We again inspected the predicted logits for the masked tokens. We think that the accuracy achieved by the scaled binary weights approach is less prone to accidental success than from the last blog post. In the previous baselines, the model would only predict a single token. In addition, these tokens were chosen with high-confidence (incorrectly) and were often punctuation marks (periods, commas, etc). This time, the model predicted many different tokens, and achieved markedly higher accuracies than last week. The version of BERT we are using has a vocabulary size of 30522, so a model that randomly predicts tokens would have an expected accuracy of $\frac{1}{30522} \approx 0.00003276$. For comparison, the approach where we binarized all layers except for the feed-forward neural networks (FFNN's) achieved an accuracy of 0.0652, which is significantly higher. We believe that this approach worked better since with the additional scaling factor α, the approximated weights were much closer to the original weights. The goal of our previous approach was not to on approximate the original weights closely, but binarize the weights in the simplest way.

Training a binary neural network

For the binary neural network, we are attempting the same approach as in XNOR-Net. It does not make sense to directly train binary weights, since gradients tend to be small and thus would often not change the value of the binary weights. Particularly, if the gradient is less than 1 in absolute value, the value of the binary weight won't change the sign if we naively re-binarize the weight after a parameter update. Instead, we use a different approach. We keep the full precision weights during training and perform gradient descent on them. However, we perform binarization (as described earlier) in selected components of the model. Importantly, this adds new "binarization" nodes to the computation graph. Thus in the backward pass, gradients of the binarization process are calculated and thus backpropagated, thus affecting the update of the full-precision weights. Finally, after the network is trained, the binarization code can be removed from the network and the weights of the binarized modules can be replaced with their binarized versions. One important issue is that during the binarization process, we used the sign function. Thus the gradient is 0 everywhere, except for at 0 where it's not defined. This causes gradients to become 0 and learning halts. To counteract this, the XNOR-Net authors have modified the gradient of the sign function to be:

$$\frac{\partial\,\mathrm{sign}(w)}{\partial w} = \begin{cases} w & \text{if } |w| \le 1 \\ 0 & \text{otherwise} \end{cases}$$

We are planning on testing different ways of defining this gradient in the future, but for now we are using what has worked for XNOR-Net.

Training binary BERT

First, we implemented a custom sign function module with the modified derivative as described above. We created a custom linear module that performs the binarization by making use of the sign function module. We then replaced all linear layers in the BERT layers with our binary linear layers , except for the embedding layer and the language modeling head. This corresponds to the "Layers" approach from the table above. We started by trying to train this binarized version of BERT on a small subset (8192 sequences of length 64) from the PennTreeBank-3 training data. We immediately noticed how performance intensive this training is, despite our small training data and short sequence length. Furthermore, we noticed that training BERT is very sensitive to learning parameters. Our initial attempts of training binarized BERT were unsuccessful. Our hope was that we could overfit BERT to this small dataset, but the loss was barely changing, despite us trying different hyperparameters for learning. We found parameters that seemed to work decently, and we trained binarized BERT for 16 epochs on this small dataset. We then evaluated it on the validation data we used in the previous experiments. Below are the results:

Binarized Components	Cross Entropy	Accuracy	Mean Reciprocal Rank 32	Inference Time (ms)	Memory Usage (MB)
Layers (trained)	6.4053	0.1021	0.1645	2.2765	103
Layers	9.1692	0.0274	0.0451	1.9865	103

We can see that only training for a small number of epochs with little data, already gives significant improvements for cross entropy, accuracy and MRR. We are excited to try to train with more data and for more epochs.

Failure modes of training

As a sanity check, we trained original non-binarized BERT on this small subset to make sure we can overfit it to reach close to 100% accuracy. We used the same Adam parameters as in the paper, but found that we had to decrease the learning rate to 10^{-6} to make BERT fit the learn properly. We also noticed that the loss did not start decreasing immediately, but instead would vary slightly and start decreasing later on during training. This indicates to us that it is essential to train for much more epochs, so we suspect that us only training for 16 epochs was one of the reasons we did not see a large decrease in loss. We also think that we need to train with much more data. Compared to the original dataset BERT was pre-trained with, our dataset of 2^{13} sequences is tiny. Since we are not training from scratch, we think it is okay to have a smaller dataset, but we will certainly use more data in the future than we are using now. We only chose to use such little data to be able to quickly run experiments to adjust learning parameters. Another suspicion we have for why training is not working very well is that the values flowing through the network might become too large, which causes the gradients of the sign function to become 0 and essentially halt training.

Next steps

Above we identified several issues with training. Firstly, we want to train for more epochs to see if the loss of binary BERT decreases once trained for longer. Secondly, we want to use more data for training. So far we have used the PennTreeBank-3 data, but have decided that we want to use a subset of the data that BERT was originally trained on, namely BooksCorpus and Wikipedia. We already got access to the BooksCorpus and are considering using it for training. Also, so far we have used torchtext for loading the PennTreeBank data, but we want to try using allennlp dataset readers, since they support efficient batching that groups sequences of similar length to avoid excess amounts of padding, which is an issue with out current implementation. Thirdly, we want to further tune learning parameters. In the BERT paper, the authors had a specific learning rate scheduling mechanism that we want to try too. Fourthly, we want to investigate the potential issue of vanishing gradients by manually inspecting the gradients later during training. Finally, we want to attempt a similar method as presented in the paper Training a Binary Weight Object Detector by Knowledge Transfer for Autonomous Driving. We already implemented a similar idea and ran initial experiments. This is discussed briefly in the next section.

Knowledge Transfer

In knowledge transfer, the idea is that we train the student network (binary BERT) to produce similar outputs as the teacher network (original BERT) at each layer by adding a penalizing term to the loss. Particularly, our initial approach is as follows. BERT (base) has 12 transformer layers, each of which produces a tensor of hidden states of shape `(batch_size, max_sequence_length, hidden_size)`. Let $H_i^{(t)}$ be the hidden states tensor of the ith layer of original BERT after a forward pass and similarly, $H_i^{(s)}$ the output of the ith hidden layer of binary BERT. Also let $\lambda \in \mathbb{R}^{12}$ be a vector of non-negative scalars, which we chose as hyperparameters. Then we define the transfer loss as

$$\mathcal{L}(H^{(t)}, H^{(s)}) = \sum_{i=1}^{12} \lambda_i ||H_i^{(t)} - H_i^{(s)}||_2^2$$

Essentially, we are just trying to get binary BERT to produce approximately the same outputs at each layer. This loss is simply added to the cross entropy loss during training.

We implemented this and ran some initial experiments with a few values of λ, but the performance of the model was not better than before after training for the same number of epochs. We suspect that we carefully need to tune λ and again have to train for many more epochs. We also implemented the above without looking much into papers, we only got the broad idea from the aforementioned computer vision paper. Thus we plan on reading more papers about approaches similar to this one. We believe that knowledge transfer has a lot of potential and could help significantly during training.