Learn how to make BERT smaller and faster

blog.rasa.com/compressing-bert-for-faster-prediction-2/

August 8, 2019

In this blog post, we discuss ways to make huge models like BERT smaller and faster. You will learn:

- Why compressing today's best-performing models is very important (jump to the section)
- What ways to compress models there are and why accelerating models is more difficult than making them smaller (jump to the section)
- What we found out while trying to compress BERT with the*quantization* method, using TensorFlow Lite (jump to the section)

Motivation

Models are (too) big

Today's best-performing systems in language processing or computer vision use huge neural architectures. Take language models as an example: the state-of-the-art are transformer-based architectures like BERT [1], XLNet [2] or ERNIE 2.0 [3], available as pre-trained models for anyone to use for any language task. These models often have hundreds of millions of parameters.

Big models are slow, we need to accelerate them

Despite their superb accuracy, the huge models are difficult to use in practice. Pre-trained models typically need to be fine-tuned before use, which is very resource-hungry due to the large number of parameters. Things get even worse when serving the fine-tuned model. It requires a lot of memory and time to process a single message. What is a state-of-the-art model good for, if the resulting chatbot will only be able to handle one message per second? To make them ready for scale, we seek to accelerate today's well-performing language models, in this case, by compressing them.

Overview of model compression

If you just want to see our empirical results, jump to this section.

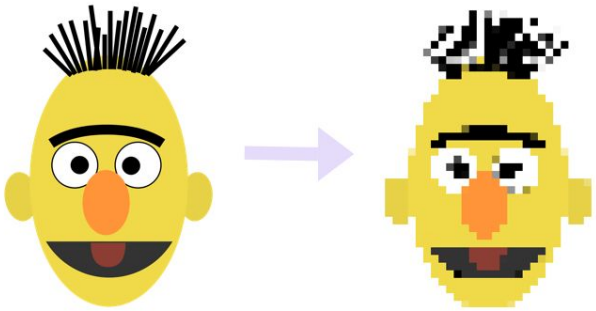
What is it?

Compressing models can serve different purposes: make the model physically smaller to save disk space, make it require less memory during computation, or make it faster. Intuitively, these aims are intertwined: if you make a model smaller, it will likely be faster too. In fact, many of the recently proposed compression methods aim to make models both smaller and faster. We will briefly summarize the research in this area and point out the key concepts.

Compression during training vs after training

Compressing a model means reducing the number of parameters (weights) or their precision. The simplest way is compressing a model *post training* (before serving it). However, the network doesn't get a chance to recover after overly aggressive pruning and may significantly underperform. It's better to apply compression in small steps during training, giving the model a chance to recover by further learning from the data. Importantly, many compression methods can be applied both post-training and during training. It is up to you to choose: post-training can be faster and often doesn't require training data, while during-training compression can preserve more accuracy and lead to higher compression rates.

Quantization



The theoretical view

Quantization means decreasing the numerical precision of a model's weights. One promising method [15,4] is*k-means quantization*: given a model's weight matrix W of floating point numbers, we group all of them into N clusters. Then, we turn W into a matrix of integer values from [1..N], each being a pointer to one of the N clusters' centres. This way, we've compressed each element of the matrix from 32-bit floats to only log(N)-bit integers. Computer architectures will typically only allow you to go down to 8 or 1 bits. Note that the latter is a rare case because *binarizing* a weight matrix - making it contain only 2 distinct values - will likely hurt the model too much.

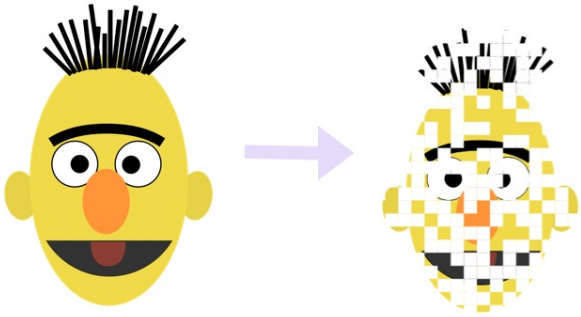
The practice

It is important to note that methods like k-means quantization don't improve memory requirements or speed. Why? Because, at inference time, before being used, each weight matrix has to be reconstructed, i.e. filled again with 32-bit floats (the cluster centres). Hence, we call this **pseudo quantization** in contrast to **real quantization**, when each weight is *permanently* encoded using fewer bits. One simple method is implemented in the TensorFlow Lite toolkit. It turns a matrix of 32-bit floats into 8-bit integers by applying a simple "centre-and-scale" transform to it: $W_8=W_{32}/scale+shift$ (*scale* and *shift* are determined individually for each weight matrix). This way, the 8-bit W is used in matrix multiplication and only the result is then corrected by applying the "centre-and-scale" operation in reverse. A variation that should accelerate computation even more is *full quantization* (also part of TensorFlow Lite), where not just the weight matrices, but all model math is quantized (in particular, the activations). The memory requirements in both cases are 4x lower. However, any acceleration depends on how much faster your computer architecture can multiply the integer matrices. This is where machine learning research hits the reality of computing infrastructure.

Quantizing models during training

Beyond the post-training quantization discussed so far, you can do even better with *quantization-aware training* [5] (also available in TensorFlow Lite). The modified training flow then looks like this: for each training step, 1. quantize the weights, 2. compute the loss in such quantized network, 3. compute gradients of the loss with respect to the unquantized weights, 4. update the unquantized weights. After the training, you finally quantize the model's weights and use the quantized model for prediction. The network thus learns to optimize its quantized versions's performance.

Pruning



Removing weight connections

Pruning removes parts of a model to make it smaller and faster. A very popular technique is*weight pruning* [6, 7], which removes individual connection weights. This technique is sometimes compared to the early development of the human brain, when certain connections are strengthened while others die away. Simple *weight magnitude pruning* removes the weights closest to 0, using magnitude as a measure of connection importance. Connections are removed by setting the corresponding elements of a weight matrix to 0. Note that it does not make the weight matrix smaller or speed up computation. To actually save storage or memory space, you should use the weight matrices in a sparse matrix format. Any acceleration then depends on the particular implementation of sparse matrix multiplication. At the time of writing this post, sparse multiplication in TensorFlow is not much faster than normal (dense) multiplication (see these benchmarking results), but there are promising workarounds such asthis one by OpenAI for faster sparse multiplication on GPUs. If you are curious, TensorFlow's Model Optimization toolkit now provides tools for applying weight pruning to Keras models.

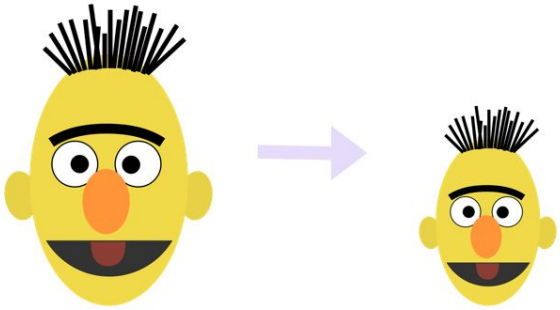
Removing neurons

Compared to weight pruning, *neuron pruning* removes entire neurons. This translates to removing columns (or rows) in weight matrices. As a measure of neuron importance, a combination of neuron activations *and* error gradients (computed on a representative dataset) works well [8]. Unlike with weight pruning, here, the weight matrices are made physically smaller and any computation with them faster. We are currently exploring neuron pruning applied to a pre-trained BERT model during the fine-tuning phase. By alternating between pruning and fine-tuning steps, the remaining neurons have a chance to compensate for the removed ones.

Removing weight matrices

Recently, pruning has been taken to the next level: in [9], the authors remove entire attentional heads from big transformer-based models with minimal accuracy losses. Naturally, such a simple approach is attractive. However, removing entire weight matrices ignores the importance of individual neurons. We hypothesize that pruning at a more fine-grained level can ultimately allow us to shrink models more. On the other hand, higher compression rate doesn't guarantee better acceleration: neuron pruning can result in differently-sized weight matrices, which makes matrix multiplications harder to efficiently parallelize.

Knowledge distillation



Learning smaller models from big ones

Knowledge distillation is not a model compression technique, but it has the same goals and effects. After training a big and slow model (the teacher), a smaller model (the student) is trained to mimic the teacher's behaviour - whether its outputs or its internal data representations. This leads to very straightforward improvements in both speed and size across different types of networks, from CNNs [10] to LSTMs [11]. Very interesting is the effect of teacher-student architectural differences: while [10] recommend students deeper and thinner than teachers, [12] and [13] present tricks for better knowledge transfer between very different architectures. In [14], the authors even successfully learn a small Bi-LSTM student from BERT.

Connecting knowledge distillation with other techniques

We think that better understanding of big networks can help with knowledge distillation: if you know how different parts of BERT behave, you can design student networks more cleverly (e.g. with fewer heads) or let them learn the teacher's most important internal representations. In this sense, quantization and pruning are tools to help you analyse your models by seeing which parts and how aggressively can be simplified or removed. Of course, it may well be that you will be satisfied with the performance of a very simple student: a uniformly down-scaled teacher.

Hands-on: compressing BERT with quantization

Let's speed up BERT

Now let's share our own findings from compressing transformers using quantization. A later blog post focused on pruning will follow. The aim is to speed up the inference of BERT so that we can use the model for better intent classification and named entity recognition in the NLU pipeline. Right now, our BERT-based intent classifier takes

~120ms on a CPU to process a single message, while our other classifiers are often ~100x faster.

Details of the setup

To start with, we compressed a lightweight intent classifier consisting of a 2-layer bi-directional transformer encoder preceded and followed by single embedding layers. Later, we moved on to the BERT classifier (based on the pre-trained uncased "base" variant with 110 millions of parameters, see here). With both classifiers, we used a bigger version of the NLU data from Rasa's demo bot Sara. Model accuracy was measured as the macro-average F1 score. All our code uses TensorFlow and you can even dig through our code in this branch of the Rasa repo.

Pseudo quantization

Getting to know the models without speeding them up

To get a taste of how much transformer-based models can be quantized, we applied the k-means post-training pseudo quantization to the small classifier. Before compression F1 on a test portion of the data was 0.80. We found that:

- Grouping the values of each weight matrix into as few as 4 clusters works well (F1 stays at 0.79, a 1.6% relative drop). This means 16x compression (from 32 to 2 bits per weight). Going for only 1 bit, i.e. binarising each weight matrix, degraded F1 drastically.
- The embedding layers are more sensitive to quantization than the transformer encoder. Binarising the encoder's weight matrices while keeping the embeddings layers 8-bit quantized (i.e. with 256 clusters) yields the same F1 values as the uncompressed model.

Clearly, the small classifier can be quantized aggressively and real 8-bit post-training quantization is unlikely to hurt the accuracy.

Real quantization with TensorFlow Lite

What is TFLite

To actually accelerate the classifiers, we turned to TensorFlow Lite (or TFLite for short) - a toolkit for deploying TensorFlow models on small devices. It features a converter which turns TensorFlow models into 8-bit post-training quantized TFLite models and, optionally, applies further optimizations. To speed up inference, the converter effectively replaces the TensorFlow operators in a computational graph with their TFLite 8-bit counterparts.

Working with a young toolkit

At the time of writing this blog post, TFLite is an exciting project under active development. Many TensorFlow operators still don't have their 8-bit equivalents implemented (see this list). For example, the converter will fail on a model which uses `tf.cum_prod` or `tf.scatter_nd`. As a temporary work-around, TFLite lets you port such unsupported operators from TensorFlow (at the cost of not having them optimized). Sadly, the porting feature (marked as experimental) was making the converter fail while applying further model optimizations. We even ran into this bizarre error occuring on Linux but not on MacOS. In the smaller classifier, we had to rewrite the code to remove all unsupported operators, while converting BERT went smoothly. Beyond quantizing only the weights, we wanted to try the full quantization implemented in TFLite. Unfortunately, we encountered converter errors which were too difficult to debug, and we decided to not go any further.

Can TFLite improve inference on big devices?

At the moment, TFLite optimizes models for mobile and IoT devices. On a desktop CPU, the BERT classifier's inference time increased from ~120ms to ~600ms per message (without further TFLite optimizations). Applying any one of the 2 optimizations - `OPTIMIZE_FOR_SIZE` or `OPTIMIZE_FOR_LATENCY` - increased the inference time to ~2s/message. This shows how the optimizations do not necessarily generalize to desktop or server architectures. However, the TFLite 2019 roadmap looks promising and we hope that the toolkit will overlap more with our use cases. Right now, it is unlikely to help us accelerate our models.

Quantizing with TFLite: the results

- The real 8-bit post-training quantization didn't hurt the models' accuracy. The simple transformer encoder's F1 decreased only by 0.2% relative. The BERT classifier's F1 didn't change after quantization and went down by only 0.3% relative after applying any of the further TFLite optimizations.
- In terms of model size, the quantization indeed made both models 4x smaller (from 32 to 8 bits per weight), which can be an important result if you have big models and limited disk space or memory.
- If you want to speed up your models' inference on desktop or server CPUs, TensorFlow Lite will probably not help you. In our case, the quantized models were ~5x slower.
- Seeing the runtime increase, we concluded that TFLite is not fit for our use cases right now. However, we encourage you to explore TFLite's full quantization (quantizing all model math), as well as quantization-aware training.

Conclusion

We gave an overview of the main concepts and approaches to model compression. As discussed, compressing models for disk size or memory usage at negligible accuracy losses is realistic. Compressing for speed, on the other hand, is tricky in practice. It can depend on faster integer multiplication or sparse matrix multiplication.

In our hands-on exploration, we showed how a small transformer and BERT encoders can be quantized drastically without significant accuracy loss, although speeding up the inference is hard. Post-training 8-bit quantization using TensorFlow Lite slowed down BERT's inference by ~5x.

As a next step, we are exploring weight and neuron pruning applied to BERT. The preliminary results are promising, so stay tuned for a future blog post. In the longer term, we can try knowledge distillation, designing better student networks by using the insights from other compression experiments.

Do you have any favourite model compression tricks? We would love to discuss them with you in ourRasa Community forum. We also encourage you to use Rasa for exploring new techniques: it is very easy to build your own component - such as an intent classifier - and make it part of an entire NLU pipeline. At Rasa, we love open source and the framework used in this blog post is publicly available.

Sources

[1] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Retrieved from https://arxiv.org/abs/1810.04805

[2] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., & Le, Q. (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding. Retrieved from https://arxiv.org/abs/1906.08237

[3] Sun, Y., Wang, S., Li, Y., Feng, S., Tian, H., Wu, H., & Wang, H. (2019). ERNIE 2.0: A Continual Pre-training Framework for Language Understanding. Retrieved from https://arxiv.org/abs/1907.12412

[4] Cheong, R., & Daniel, R. (2019). transformers.zip: Compressing Transformers with Pruning and Quantization. Retrieved from http://web.stanford.edu/class/cs224n/reports/custom/15763707.pdf

[5] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2017). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Retrieved from https://arxiv.org/abs/1712.05877

[6] Gale, T., Elsen, E., & Hooker, S. (2019). The State of Sparsity in Deep Neural Networks. Retrieved from https://arxiv.org/abs/1902.09574

[7] Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both Weights and Connections for Efficient Neural Networks. Retrieved from https://arxiv.org/abs/1506.02626

[8] Molchanov, P., Tyree, S., Karras, T., Aila, T., & Kautz, J. (2016). Pruning Convolutional Neural Networks for Resource Efficient Inference. Retrieved from https://arxiv.org/abs/1611.06440

[9] Michel, P., Levy, O., & Neubig, G. (2019). Are Sixteen Heads Really Better than One? Retrieved from https://arxiv.org/abs/1905.10650

[10] Romero, A., Ballas, N., Kahou, S., Chassang, A., Gatta, C., & Bengio, Y. (2014). FitNets: Hints for Thin Deep Nets. Retrieved from https://arxiv.org/abs/1412.6550

[11] Kim, Y., & Rush, A. (2016). Sequence-Level Knowledge Distillation. Retrieved from https://arxiv.org/abs/1606.07947

[12] Luo, P., Zhu, Z., Liu, Z., Wang, X., & Tang, X. (2016). Face Model Compression by Distilling Knowledge from Neurons. Retrieved from https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/11977/12130

[13] Mirzadeh, S., Farajtabar, M., Li, A., & Ghasemzadeh, H. (2019). Improved Knowledge Distillation via Teacher Assistant: Bridging the Gap Between Student and Teacher. Retrieved from https://arxiv.org/abs/1902.03393

[14] Tang, R., Lu, Y., Liu, L., Mou, L., Vechtomova, O., & Lin, J. (2019). Distilling Task-Specific Knowledge from BERT into Simple Neural Networks. Retrieved from https://arxiv.org/abs/1903.12136v1

[15] Han, S., Mao, H., & Dally, W. (2015). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. Retrieved from https://arxiv.org/abs/1510.00149