Pruning CodeBERT for Improved Code-to-Text Efficiency

Alex Gu
MIT
gua@mit.edu

Ria Sonecha
MIT
rsonecha@mit.edu

Saaketh Vedantam
MIT
saakethv@mit.edu

Bharat Runwal
Independent Researcher
bharatrunwal@gmail.com

Diganta Misra
Mila - Quebec AI Institute, Landskape AI
diganta.misra@mila.quebec

Abstract

The size and prevalence of large language models (LLMs) make them an apt target for model compression. Most LLMs consist of a Transformer encoder and decoder, which each have 6 to 12 layers of multiheaded self-attention blocks, along with fully connected layers. This results in a large number of parameters, making them quite expensive to train and query. Our work focuses on finding techniques to prune CodeBERT, a specific LLM trained to work multimodally between text and code. We explore the effects of structured and unstructured magnitude pruning on the encoder layers of CodeBERT, evaluating on the task of generating natural language comments from a piece of Ruby code.

1 Introduction

1.1 Large Language Models for Code

Large language models are powerful tools for understanding, summarizing, translating, and generating text. In recent years, researchers have started developing large language models to accelerate the software development process by training them to perform tasks such as natural language code search, code documentation, and code generation. One such example is CodeBERT, a pre-trained, bimodal, transformer-based neural network that learns representations to support NL-PL tasks [1]. CodeBERT uses the same model architecture as RoBERTa (Figure 1), which has 125M trainable parameters [2]. finetuned versions of CodeBERT achieve state-of-the-art performance on code search and code documentation generation.

In addition to new models for NL-PL tasks, the creation of large NL-PL datasets has been crucial for the development of large language models for code tasks. Microsoft's CodeXGLUE (General Language Understanding Evaluation benchmark for Code) is a benchmark for code intelligence which contains 14 datasets for 10 code intelligence tasks [4].

1.2 Pruning Neural Networks

Previous works have shown that it is possible to prune the majority of model parameters without affecting the overall model accuracy [5, 6]. Other studies have also shown that pruned models perform well in terms of metrics besides accuracy such as generalization, prediction uncertainty, robustness, and interpretability [7]. The evaluation for these works was done on CNN models trained on image datasets. For our project we were interested in exploring the effect of pruning large language models, specifically language models for code intelligence.

Preprint. Under review.

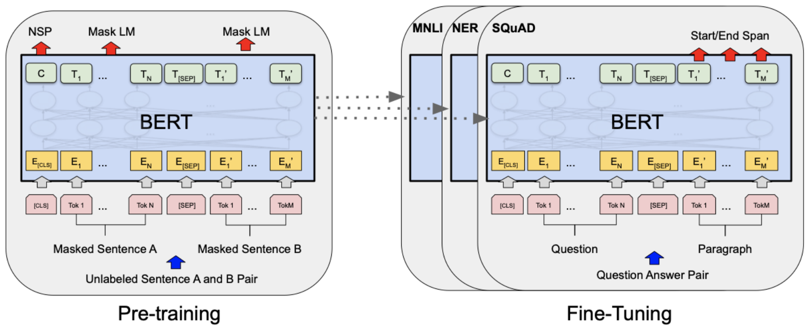

Figure 1: CodeBERT model architecture. Image taken from [3].

BLEU-4 Score	Quantization (Torch Dynamic)	Original	Half Precision (model.half())
Valid	9.38	13.23	13.26
Test	8.83	12.25	12.24

Figure 2: Performance of quantization methods

2 Experiments and Results

For our project we focused on large language models optimized for the code-to-text task (natural language comment generation). For all of our evaluation we used a subset of the full code-to-text dataset which only contained examples in Ruby. The Ruby dataset contained the fewest data points, allowing us to train and evaluate our pruned model in a reasonable amount of time. In our experiments we were primarily focused on 4 types of pruning: random pruning, magnitude-based pruning, layer-wise random pruning, and layer-wise magnitued-based pruning.

Quantization: We tried two naive quantization methods: first, using fp16 inference (via `model.half()` in PyTorch), and second, using INT8 quantization with `torch.quantization.quantize_dynamic()`. As shown in Figure 2, using fp16 does not adversely affect performance, while using INT8 *in this specific manner* does. We suspect that using other approaches like LLM.int8 could improve the performance, but did not pursue this route further.

Encoder Pruning: Specifically, we prune the weights of the attention queries, keys, and values, as well as the dense layer. This means that we keep the biases, embedding layers, LayerNorm parameters, and the entirety of the decoder.

Experiment 1: LTH-style pruning, rewinding to CodeBERT: In the first experiment, we start with a pre-trained CodeBERT model. Inspired by the Lottery Ticket Hypothesis [7], at each finetuning epoch, we run the default 3000 iterations of finetuning, prune 10% of the encoder weights (pruning method differs by experiment), and rewind the weights to that of the pre-trained CodeBERT model. In the initial experiment, we compared global unstructured random and L1 pruning. In the L1 case, because we used an unstructured approach, we compared all prunable parameters across the entire encoder. The results, using a batch size of 16, as shown in Figure 3, demonstrating that L1 pruning performs better than random pruning. In addition, throughout these experiments, as shown in Figure 4, we discovered that using a batch size of 8 is much better than using a batch size of 16. Therefore, we use batch size 8 for the rest of the experiments.

Experiment 2: LTH-style pruning, rewinding to finetuned CodeBERT: We found that in the previous experiment, at low sparsities, we started with a model that had not been trained to convergence, so we do not get good accuracy at those sparsities. Therefore, in this experiment, we start with a CodeBERT model finetuned on translating Ruby to natural language. We keep the same setup as Experiment 1, but rewind to the weights of the finetuned model at each step. As shown in Fig. 5, this approach generally outperforms the previous at low sparsities, but not higher sparsities.

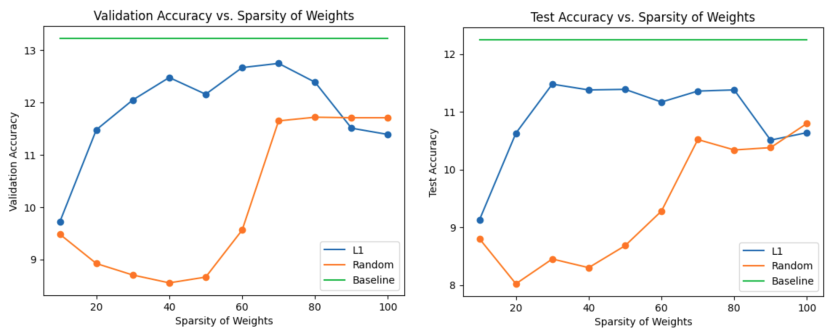

Figure 3: Performance of LTH style pruning, rewinding to pre-trained CodeBERT

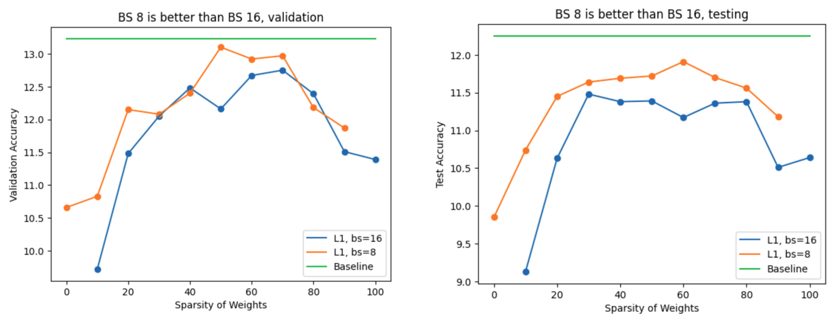

Figure 4: Performance of LTH style pruning, showing that batch size has a considerable effect on output

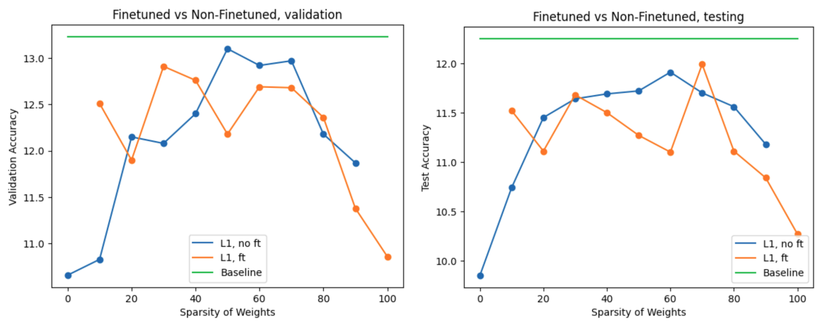

Figure 5: Comparing best results of rewinding to CodeBERT vs. finetuned CodeBERT

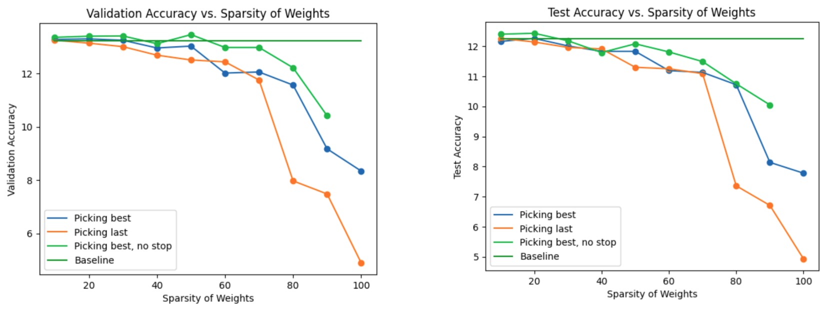

Figure 6: Comparing structured pruning with validation-based pruning results

Experiment 3: Structured pruning with validation-based pruning: We next investigate structured pruning: instead of doing L1 pruning globally on the entire encoder, we do L1 pruning on each set of parameters separately. This means that we prune 10% of each attention query weight set, 10% of each dense layer weight set, etc. The rationale behind this is that the weights are likely not normalized across the whole network, and if the weights of one of the layers is much smaller than the rest of the network, the full layer would likely be removed. We found that the performance was still poor for the following reason: for low sparsities, when we start with a finetuned model, the performance is already good. Therefore, doing further finetuning would lead to an overfitted model. This experiment shows that even though we are rewinding, if we prune based on the overfitted model weights, we get poor performance. Therefore, in the next set of experiments, we try three methods, described below. All results shown use structured L1 pruning.

- *Picking last*: instead of finetuning the full 3000 iterations per epoch, we measure the BLEU-4 score every 100 iterations, and stop if the current score is worse than the previous two measurements. We then prune and rewind from this last model that is obtained.
- *Picking best*: we use the same stopping criteria as the previous approach, but prune based on the model with the *highest BLEU-4 score*.
- *Picking best, no stop*: In this approach, we *do not stop early*, finetune the full 3000 iterations at each epoch, and prune based on the best model.

The results are shown in Figure 6. We can see that the highest-performing approach, as expected, is to pick the best model without stopping. We see that we are able to recover the original accuracy only at 20% sparsity, though we are able to get close with 50% sparsity.

3 Discussion and Future Work

Throughout the above set of experiments, we observed a few very interesting results that deserve further investigation:

- Batch size can have a relatively large impact on performance
- At extremely high sparsities (above 80%), the initial experiments of rewinding to the weights of a pre-trained CodeBERT performed much better than rewinding to a finetuned CodeBERT.
- With LTH type pruning, if we prune and rewind from an overfitted model, the performance is much worse than if we do so from a correctly tuned model. This is a bit surprising, and seems to suggest that the weight patterns differ between overfitted models and models in the sweet spot.

In addition, we identify a few interesting directions and questions for future exploration:

- In our work, we've only done pruning on the encoder and investigated very basic pruning methods. Is it possible to use more sophisticated pruning techniques (like pruning the decoder, pruning attention weights that are rarely used) to achieve better performance?
- In this work, we did hand-waved comparisons of rewinding to CodeBERT and rewinding to a finetuned CodeBERT. Do the standard results in the LTH literature still hold for finetuning?

- Data pruning is another common method to increase the efficiency of LLM's. It has been shown that in language models, it is possible to feed in a small amount of the original sentence for tasks like sentiment prediction by analyzing attention weights. To what extent is it possible to exploit data pruning in the code modality?
- Understanding how pruning on code LLM's is different from language LLM's: code is a modality that is much structured than language. Hence, LLM's trained on code should be different than LLM's trained on natural language. Is it possible to investigate the difference from the lens of pruning?

References

[1] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

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

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[4] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[5] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

[6] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

[7] Tianlong Chen, Zhenyu Zhang, Jun Wu, Randy Huang, Sijia Liu, Shiyu Chang, and Zhangyang Wang. Can you win everything with a lottery ticket? *Transactions of Machine Learning Research*, 2022.