Adaptive Computation with Iterative Model Instantiation

On-Device Machine Learning Project Proposal

Patrick Fernandes, Jared Fernandez, Hao Zhu, Haoming Zhang

1 Motivation

Modern state-of-the-art systems often have millions and even billions of parameters, sometimes requiring large workstations with many GPUs to be able to do inference with them. However, most computational devices in the world are *embedded* devices, making the use of such large models not only slow (due to small computational power) but impossible (due to memory constraints).

Approaches to circumvent this typically rely on *model compression* techniques that aim at reducing the original model size or representing them with less precision (and therefore using less memory). However most of the time this comes at the cost: typically these models lose predictive accuracy, and some of these methods do so non-uniformly over the data distribution [5], potentially exacerbating algorithmic biases.

Typically the main constraint in memory comes from volatile memory (such as RAM and VRAM) that, while much faster, tends to be more expensive than non-volatile memory (such as flash memory and hard disks). Since modern deep-learning frameworks assume that the full model is instantiated into memory, this makes volatile memory the main constraint of these models. But what if during the computation of the model, we could iteratively load the model's parameters by taking advantage of the layered nature of current SotA systems?

While this would increase certainly the latency of a forward pass due to the slow read speeds of non-volatile memory, what if for most inputs we could *end* the computation early, and only performing the full computation for harder examples? Recent work has shown that it is possible to dynamically determine the number of layers necessary to perform inference on an input, and this can even potentially increase the accuracy of the model. Could we combine this line of research with the idea described previously to allow us to run large models on small devices while preserving accuracy and expected latency?

2 Hypothesis

Based on this we formulated the following hypothesis:

- *It is possible to the run a model's inference computation graph that use more memory than a device has by iteratively instantiating the model's parameters*: A confirmation of this hypothesis would allow running large, un-compressed models on small embedded devices.
- *It is possible to minimise the overhead time of iterative instantiation based on the model's architecture and device memory constraints such that it makes its use practical in real world applications*: While the first hypothesis is concerned with existence, a positive answer to this one would mean that such algorithm would not be prohibitively expensive and could be applied to embedded devices without affecting the users experience significantly.
- *It is possible to optimize the runtime-accuracy tradeoff on every single device by including the corresponding hardware features in the current early exiting schema*: Hardware features includes corresponding hardware specifications and even the model instantiation cost from the two hypotheses above.

3 Methodology

3.1 Iterative Model Instantiation

Given a model with n parameters $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$, which can be decomposed into

$$f_\Theta(x) = f_1 \circ f_2 \circ \cdots \circ f_n(x), \tag{1}$$

execution time function $E : \mathcal{F} \to \mathbb{R}$, and memory constraints M, we are interested in the following optimization problem:

$$\begin{aligned} &\min_{t_0=0,t_1,t_2,\ldots,t_m=n} \sum_{i=1}^{m} E(f_{t_i} \circ f_{t_i+1} \circ \ldots f_{t_{i+1}-1}) \\ &\text{s.t.} \sum_{j=t_i}^{t_{i+1}-1} \texttt{sizeof}(\theta_i) \leq M, i = 0, 1, \ldots, m-1 \end{aligned} \tag{2}$$

Simplification and Greedy Solution If the execution time function is a linear function of parameter size, i.e.

$$E(f_a \circ f_{a+1} \circ \ldots f_b) = \sum_{i=a}^{b} E(f_i) = \sum_{i=a}^{b} k\texttt{sizeof}(\theta_i) + C, \tag{3}$$

where $k, C \in \mathbb{R}^+$ are constants. C represents the *cost* associated with instantiating the parameters. In this case, the optimization problem is reduced to

$$k \sum_{i=1}^{n} \texttt{sizeof}(\theta_i) + Cm, \tag{4}$$

in other words, the number of instantiation operations m. Obviously, the greedy solution, $t_i \leftarrow \max_l \text{ s.t.} \sum_{j=t_{i-1}}^{l-1} \texttt{sizeof}(\theta_i) \leq M$, is the optimal one.

3.2 Hardware Aware Early Exiting

Recent work examining transformer neural networks has shown significant redundancy across model layers [7]. In practice, execution of all layers is not necessary to produce high accuracy model predictions. Existing approaches use an MLP "off-ramp" after each encoder block, which use the encoded hidden states of the intermediate layers to make a final prediction without executing all of the models layers [1, 17, 19].

Previous approaches to dynamic model inference use estimated model confidence or patience as criteria for early-exit. However, such approaches are not sensitive to the settings in which models are being deployed or the real-world constraints of the hardware. We propose further conditioning early exit on estimated model latency.

Rather than having the early exit module depend solely on model hidden states, we propose the addition of features for the model architecture and hardware setting. Models can be evaluated by examining the downstream accuracy as a function of realworld runtime.

Early Exiting with Runtime Thresholding To ensure that model latency falls within an acceptable range, the criteria can be enforced with a hard constraint on runtime after each layer.

Given a execution time constraint t, we seek to find the model with the highest performance meeting the desired latency requirement. Define a latency predictor $LP : \mathbb{R}^{n+m} \to \mathbb{R}$ and an early-exit classifier $EE : \mathbb{R}^{n+m} \to \mathbb{R}$ with hidden state inputs n and m features corresponding to the hardware and model architectures.

After each layer execution, if the model confidence is at least the threshold value c return the output of the early-exit classifier. If the model confidence is below the confidence threshold, predict the latency of computing the output of the next layer and continue with execution if the the total estimated execution time is below t.

Should the projected time execution exceed model threshold, return the output of the early-exit classifier with the highest confidence from the previously executed layers.

Adaptive Model Execution Rather than using hard constraints for model execution time, the decision to early exit can be informed by hardware constraints by adding L1 or L2 penalties based on expected latency computations or hardware usage. These modifications can use the same classifier and trained parameters as those proposed with strict thresholds and instead use the values as inputs to the loss function.

4 Experiments

For our experiments, we will consider BERT [3] as the core model. This model has found widespread usage in natural language processing and early experiments show that the *base* model just barely fits into the memory of the a Jetson Nano 2GB (making it a suitable test bed for early exiting), while the *large* is doesn't fit into this device (making it a suitable for the iterative model instantiation experiments).

In particular we will evaluate our model in the GLUE benchmark[16], a collection of nine natural language understanding tasks, including single-sentence tasks CoLA and SST-2, similarity and paraphrasing tasks MRPC, STS-B and QQP, and natural language inference tasks MNLI, QNLI, RTE and WNLI. We will consider as the main metrics both **accuracy** and **latency**. As baselines we will compare the performance with the original BERT-*base* and a quantitized version of the BERT-*large*.

5 Resources

We will conduct our on-device experiments on Jetson Nano 2GB Development Kit, a small computer integrated with a 128-core Nvidia Maxwell GPU, a quad-core ARM A57 CPU and 2GB LPDDR4 shared memory. Also, we will try to deploy our models on other devices, including a Jetson Nano 4GB and Raspberry Pi 4 8GB, to compare the performance with different RAM constraints and computing capabilities.

We will perform off-device training and evaluation for our adaptive computing models. Therefore, cloud computing resource credits are required. With every $100 AWS credit, we can get around 80 hours usage of one p2.xlarge instance (1 Nvidia 12-GB K80 GPU, 61 GB RAM).

6 Extensions

6.1 Optimizing Non-Linear Cost Functions

We have shown that the simplification to the instantiation problem in Section 3.1 has a trivial solution. The linearity between execution time and parameter sizes doesn't always hold. The optimization becomes much trickier when the function has quadratic or higher order terms. Another problem is that execution time is also influenced by the neural architecture, which is harder to measure in an analytical way.

As an extension, we could consider this as bandit problem, where for each $i = 1, 2, \ldots, n$ one can choose to instantiate the layer with the previous ones or not. In this project, we will attempt to use a standard upper confidence bound to solve this problem, and compare the results with the greedy baseline.

6.2 Other I/O

To further prove the practical utility of the proposed method on real, small device use cases, we experiment with peripherals, including cameras and microphones. By having these I/O devices in the pipeline, we can demo how our method works in terms of perceived user quality (latency, accuracy) when compared to compressed baselines.

6.3 Additional Benchmarking

Following works in [3, 9], we could utilize benchmark datasets, including SQuAD [12, 13] and RACE [8], to further evaluate the representational capabilities of our models on downstream tasks.

7 Related Work

7.1 Tensor Computation with Heterogeneous Memory

Many works [2, 15, 11, 4, 6, 14, 10] in the literature have tackled the problem of performing computations with large models with a memory footprint larger than the accelerator's memory. However, classically, this is achieved with *model parallelism* [2, 15, 11], where the model is split across different devices, disregarding the case where only a single device is available.

Nevertheless some works have tackled this by using a slower memory in alternative to the accelerator's main one [4, 6, 14, 10]. However there are some crucial differences between these approaches and ours: (1) most of these approaches are deal with CNNs, where the main memory footprint comes from the activations rather than model parameters; (2) they are mostly concerned about a training regime, where extra the computation graph either needs to be keep or reinstantiated for the backpropagations; and (3) they mostly consider the case of offloading GPU memory to CPU memory, which simplifies the problem since they can manipulate memory in process.

In contrast, our approach considers the inference scenario in small embedded devices, where model is too large to fit into RAM and therefor parameter manipulations need to be done *a apriori*.

7.2 Adaptive Computing

Recently, many works [17, 19, 18] have explored adopting early exiting, in order to accelerate inference time of large-scale pre-trained language models such as BERT [3]. In particular, these methods use an MLP "off-ramp" after each encoder block, which use the encoded hidden states of the intermediate layers to make a final prediction without executing all of the models layers.

[17] adopted a threshold for the entropy of each off-ramp's distribution to dynamically exit after intermediate layers. [19] introduced Patience-Based Early Exit, in order to prevent errors from one single classifier. [18] proposed a new fine-tuning strategy and a learning-to-exit module that extends early exiting to tasks other than classification. All of these works show comparable or even better accuracy on benchmarks and significantly lower inference time.

References

[1] Arian Bakhtiarnia, Qi Zhang, and Alexandros Iosifidis. Multi-exit vision transformer for dynamic inference. *CoRR*, abs/2106.15183, 2021. URL https://arxiv.org/abs/2106.15183.

[2] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

[4] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 875–890, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378465. URL https://doi.org/10.1145/3373376.3378465.

[5] Sara Hooker, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. Characterising bias in compressed models. *CoRR*, abs/2010.03058, 2020. URL https://arxiv.org/abs/2010.03058.

[6] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378530. URL https://doi.org/10.1145/3373376.3378530.

[7] Yigitcan Kaya and Tudor Dumitras. How to stop off-the-shelf deep neural networks from overthinking. *CoRR*, abs/1810.07052, 2018. URL http://arxiv.org/abs/1810.07052.

[8] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations, 2017.

[9] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2020.

[10] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *CoRR*, abs/2002.05645, 2020. URL https://arxiv.org/abs/2002.05645.

[11] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019. URL http://arxiv.org/abs/1910.02054.

[12] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

[13] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018.

[14] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *CoRR*, abs/2101.06840, 2021. URL https://arxiv.org/abs/2101.06840.

[15] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

[16] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[17] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating BERT inference. *CoRR*, abs/2004.12993, 2020. URL https://arxiv.org/abs/2004.12993.

[18] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. BERxiT: Early exiting for BERT with better fine-tuning and extension to regression. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 91–104, Online, April 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.eacl-main.8.

[19] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit, 2020.