COBRA: Enhancing DNN Latency Prediction with Language Models trained on Source Code

Anonymous authors
Paper under double-blind review

Abstract

With the recent developments of Deep Learning, having an accurate and device specific latency prediction for Deep Neural Networks (DNNs) has become important for both the manual and automatic design of efficient DNNs. Directly predicting the latency of DNNs from their source code yields significant practical benefits. It opens a way towards profilers that can instantly feedback the latency of a given piece of deep learning code to the developer. In this paper, we conduct a preliminary study for source code based latency prediction of DNNs. We introduce **Co**de **B**ased **R**untime **A**pproximation (COBRA), that leverages a transformer encoder to learn representations of short code snippets. These representations are then aggregated by a Graph Convolutional Network (GCN) that captures the algorithmic dependencies and that estimates the latency of the implemented DNN. Our experiments with COBRA show promising results and indicate that latency prediction from code can be competitive with traditional latency prediction methods for DNNs.

1 Introduction

Deep learning methods have had tremendous success for a variety of applications, e.g., image and automatic speech recognition (ASR), natural language processing (NLP), data restoration or generation. A major challenge is the development of efficient DNN architectures that can also be deployed on mobile devices or on embedded platforms with restricted memory or computational power (Cai et al., 2019; Wu et al., 2019). When tailoring DNN architectures to a specific target device, deep learning engineers typically require knowledge of the DNN latency. Simply deploying a DNN and measuring its latency requires a lot of manual effort. In particular, the deployment often relies on many steps such as graph optimization, operator fusion, weight and activation quantization or DNN compilation.

Many methods for DNN latency prediction have been proposed. Among them, graph convolutional network (GCN) based predictors like proposed for BRP-NAS (Dudziak et al., 2020) are today considered to be the state-of-the-art. They use a GCN to predict the latency of a DNN from its network graph representation. Graph vertices are individual network layers and edges correspond to the data flow between the layers. Each vertex is characterized by a feature vector, the layer representation. A major problem is how to design good layer representations, such that they carry enough information to predict the latency of real-world DNN architectures. BRP-NAS for example uses one-hot encodings of the network layer type. This very simple approach works well with the special set of DNN architectures provided in the NAS-Bench-201 dataset (Dong & Yang, 2020). However, in general, the latency of single network layers, and hence of whole DNNs, does not only depend on the layer types but also on the parameters of the layers, such as the shape of the input and output tensors. Hence, this layer representation does not generalize well to more realistic DNN architectures.

We propose a novel and flexible method to predict the latency of DNNs using layer representations extracted from source code. We call this method **Co**de **B**ased **R**untime **A**pproximation (COBRA). First, COBRA interprets the source code to extract the corresponding network graph and layer implementations. The layer implementations consists of small code snippets with function calls to a deep learning framework. Second, it uses a transformer encoder to embed each layer implementation into a representation that is well suited for latency estimation. We propose a special training trick for the transformer encoder that enables us to learn source code embeddings that are especially convenient for DNN latency prediction. Finally, these extracted layer representations are aggregated by a GCN that captures the data dependencies between the function calls and estimates the latency of the DNN.

To our knowledge, COBRA is the first method that directly uses code representations to infer the latency of DNNs. In our experiments, we compare COBRA with BRP-NAS, a GCN based predictor that uses hand-crafted layer representations that consist of the layer type, all the layer parameters, as well as of the measured layer latency. Interestingly, COBRA consistently outperforms GCN based predictors that use such hand-crafted representations.

This paper is structured as follows: Section 2 gives an overview of important related work. The details of the COBRA model and first experiments are given in Sections 3 and 4. Finally, Section 5 gives an outlook of our future directions of work.

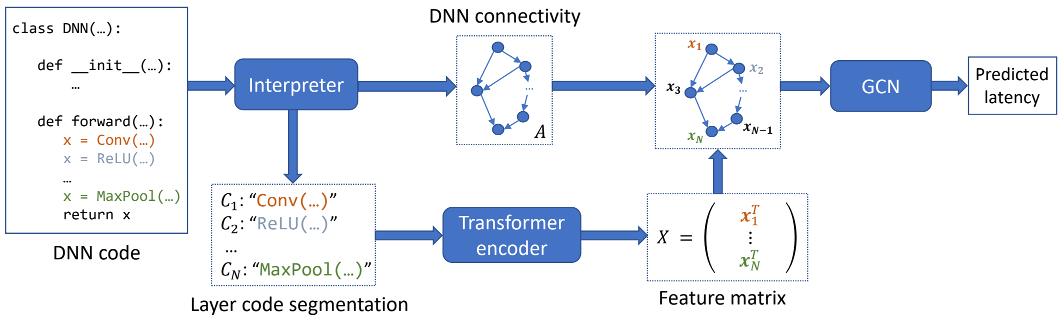
2 Related work

Language models on code: The recent works on CodeBERT (Feng et al., 2020) and Codex (Chen et al., 2021) have shown that we can train transformers like BERT (Devlin et al., 2019) and GPT-3 (Brown et al., 2020) as language models (LM) for source code. These models work well for tasks involving both programming and natural language, such as code or docstring generation. Further, Allamanis et al. (2021) used self-supervised learning to detect and repair code, given very simple defects. However, to our knowledge it has not been tried yet to predict intrinsic properties of source code, such as the corresponding latency.

Latency predictors. Different methods have been developed to predict the latency of DNN models for inference. The simplest one consists of using the number of FLOPs as a proxy for the latency estimation. However, the latency of a DNN has been shown not to be strongly correlated with FLOPs and therefore the predictions are often inaccurate (Ma et al., 2018; Tan et al., 2019; Wu et al., 2019). As a consequence, two distinct types of predictors have been devised to improve the quality of latency inference.

The first approach is to sum up the individual latencies of all the different elements composing a given neural network. Thus, many works (Cai et al., 2019; 2020; Wu et al., 2019) simply use lookup tables to obtain the latency of each layer in the neural network and then derive the end-to-end latency by summing the layer latencies. However, the efficiency of this layer-wise predictor is limited. Deep learning frameworks often fuse multiple layers into a single operation to accelerate the inference on real hardware devices. This fusion is not taken into consideration by simple layer-level predictors. Besides, building a finite lookup table restricts us to select only a subset of all possible layer configurations. To overcome such limitations, nn-Meter (Zhang et al., 2021) uses a different granularity by considering kernels instead of layers. The authors define a kernel as being an independent execution unit on a device, i.e., the fusion of different layers, performed by the framework. They develop an automatic method to detect the kernels of a neural network on a specific device and then build latency predictors in a supervised way for each of these kernels. However, building many kernel latency predictors is costly.

The other class of latency predictors are end-to-end predictors, which directly estimate the latency of the DNN. In contrast to sum-based predictors, end-to-end predictors consider the connectivity of the network. BRP-NAS (Dudziak et al., 2020) uses a graph convolutional network (GCN). Therefore, it requires the adjacency matrix of the given neural network to integrate the connectivity directly into the predictor. HELP (Lee et al., 2021) uses a GCN or a multi-layer perceptron (MLP) to predict the latency of DNNs. They test their approach on cell-based DNNs, like in the NAS-Bench-201 (Dong & Yang, 2020) dataset, and also on network architectures based on the FBNet (Wu et al., 2019) and on the MobileNetV3 (Howard et al., 2019) backbones. HELP and BRP-NAS yield highly accurate latency prediction for DNNs. However, the DNN architectures that they consider for the training and testing are all very similar, in the sense that layers of the same type always share the same set of parameters and therefore have equal latency. Of course, this is not true for general DNN architectures.



```
class DNN(…):

  def __init__(…):
    …

  def forward(…):
    x = Conv(…)
    x = ReLU(…)
    …
    x = MaxPool(…)
    return x
```

Figure 1: **COBRA** estimates the latency of a DNN from its source code representation. The model consists of three parts: 1) a code interpreter, 2) a transformer encoder to extract layer representations, 3) a graph convolutional neural network (GCN) that aggregates these representations and estimates the latency of the DNN.

3 METHOD

As shown in Figure 1 COBRA performs three steps: 1) It interprets the code and extracts a set of layer implementations, as well as their call graph. 2) A transformer encoder that consumes layer implementations one at a time and embeds them into feature vectors. 3) A GCN that uses both the call sequence and the layer embeddings to compute an estimate of the latency of the defined DNN.

The interpreter is a rule-based module. It identifies and extracts all layer implementations, that are single function calls to a deep learning toolbox. It provides two outputs: 1) The set of layer implementations $\mathcal{C} = \{C_1, C_2, ..., C_N\}$. To simplify the problem we apply normalization, meaning that we parse all the function calls to the deep learning toolbox and re-order their parameters, such that they are always passed by name and in exactly the same order. Further, all white spaces are removed. 2) An adjacency matrix $\boldsymbol{A}$ that encodes the dependencies between all C_n. More specifically, for a DNN that is defined by N layer implementations, $\boldsymbol{A}$ is an $N \times N$ upper triangular matrix.

The transformer encoder processes each C_n individually. First, C_n is tokenized. We use the uncased English BERT tokenizer from the PyTorch transformer implementation that uses a dictionary of $D = 30,522$ tokens. The transformer encoder itself has an architecture similar to BERT (Devlin et al., 2019) and also uses the same positional encoding. More specifically, it consists of 4 multi-head attention blocks with 16 attention heads each. Both the hidden and embedding dimension of the attention blocks are $C = 512$. Layer normalization is always performed before attention is applied. The maximum sequence length is $K = 600$ tokens. Consequently, the output of the encoder for the input C_n is $\boldsymbol{H}_n \in \mathbb{R}^{K \times C}$.

Figure 2 shows our pre-training setup for the transformer encoder. It uses a combination of self-supervised and supervised training to learn embeddings that are suited for latency prediction. For the self-supervised part the encoder is pre-trained to predict masked input tokens. We denote these predictions with $\hat{\boldsymbol{E}}$. This way, the transformer encoder learns to capture the properties of dependence in single layer implementations and learns to act as a language model for source code. Input tokens are predicted using a single linear layer of size $C \times D$ that is applied to the output $\boldsymbol{H}_n$ of the encoder.

We noticed that it is problematic that layer implementations rely on many parameters like input and output shapes or kernel sizes. Although these parameters heavily influence the layer latencies, the transformer will not be able to capture the properties of dependence between them if we do not provide additional information. In particular, these parameters are all independent and it is impossible to predict them once they are masked. To solve this problem, we append the tokenized measured latency of the layer implementations. With this additional information, all parameters of a layer that have an influence on the latency become dependent. These properties of dependence can then be learned by the encoder. Note that the tokenized latency is only appended during pre-training of the transformer, but not for fine tuning or for inference.

For the supervised part, the transformer encoder is also trained to explicitly predict the layer latency. Therefore, $\boldsymbol{H}_n$ is processed with a latency prediction head. Its input is the average of the token

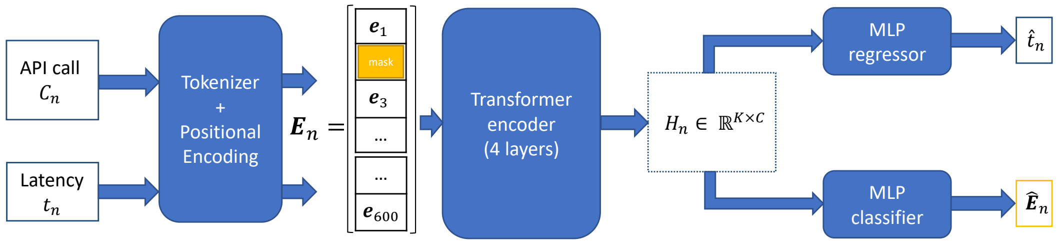


Figure 2: The transformer encoder with the masked token and the latency prediction heads, as used during pre-training.

embeddings $\boldsymbol{x}_n = \boldsymbol{H}_n^T \mathbf{1}/K \in \mathbb{R}^C$. More specifically, use a MLP with one hidden layer of size 256 and an output layer of size 1 that estimates the log-latency of the layer. Consequently, the transformer encoder is trained with two losses, i.e., the categorical cross-entropy loss for token reconstruction and the mean squared error (MSE) for the log-latency estimation. By combining these two losses for pre-training, we make sure that the transformer encoder learns to extract token representations that capture the syntax of the layer implementation while also carrying information about their latency.

The GCN combines the output of the transformer encoder for all $C_n \in \mathcal{C}$ and calculates a latency estimate for the full DNN. In general, a GCN (Kipf & Welling, 2017) can process graph signals. Let $\mathcal{G} = (V, E)$ be a graph, where V is a set of N vertices and E is a set of edges connecting them. A graph signal assigns a vector $\boldsymbol{x}_i = \boldsymbol{x}(v_i) \in \mathbb{R}^C$ to each vertex. GCNs exploit the properties of dependence between all $\boldsymbol{x}_i$ in order to make predictions on the vertex or on the graph level, while taking the graph structure into account.

For a graph with N vertices, the graph signal can be summarized in a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N]^T \in \mathbb{R}^{N \times C}$. The layer-wise propagation rule of the GCN is then defined as

$$\boldsymbol{X}^{(l)} = f(\boldsymbol{X}^{(l-1)}, \boldsymbol{A}) = \sigma(\boldsymbol{A}\boldsymbol{X}^{(l-1)}\boldsymbol{W}^{(l)}), \tag{1}$$

where $l = 1, 2, ..., L$, $\boldsymbol{X}^{(l)}$ and $\boldsymbol{W}^{(l)}$ are the layer index, the layer activation and the weight matrix, respectively. Further, $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ is the adjacency matrix of the graph.

In our case, v_i is a DNN layer implementation consisting of function calls to a deep learning toolbox and $\boldsymbol{x}_i$ is the corresponding layer representation that is extracted by the transformer encoder. In particular, the input of the GCN is $\boldsymbol{X}^{(0)} = \boldsymbol{X}$. The adjacency matrix is a binary upper triangular matrix with $A_{i,j} = 1$ if layer i is connected to layer j and $A_{i,j} = 0$ otherwise. σ is the ReLU activation function.

The latency estimates are computed from $\boldsymbol{X}^{(L)}$ the activation of the last GCN layer. More specifically, we first perform a global average pooling over the vertices dimension $\boldsymbol{X}^{(L)^T}\mathbf{1}/N$ in order to get a vector that encodes the properties of the whole graph and then apply a linear layer to obtain the output.

In our experiments, the GCN is trained in a supervised setup using the mean absolute percentage error (MAPE) $\mathbb{E}[|\hat{y}_n - y_n|/y_n]$ as a loss. Moreover, we train it independently of the transformer encoder, meaning that the parameters of the encoder are kept fixed after pre-training and are not optimized together with the GCN parameters.

4 Experiments

Datasets: We collect two automatically generated datasets, one to pre-train the transformer encoder and the other to train the COBRA model. The first dataset consists of $291,658$ randomly generated layer implementation C_n and their measured latency t_n. The set is split into $289,198$ training and $2,460$ test samples. We consider the layers `Add`, `Convolution`, `Linear`, `ReLU`, `BatchNormalization`, `AveragePooling`, `MaxPooling` and `GlobalAveragePooling`, where all the input arguments are chosen randomly. For `Linear`, we can for example choose random input and output tensor shapes. More

Table 1: Latency prediction errors for DNNs, using COBRA. We compare to layer-wise sum and GCN based predictors that use hand-crafted layer representations.

method	MAPE	RMSE (ms)	error bound (%)			
			±1%	±5%	±10%	±25%
Layer-wise sum	0.0807	**3.54**	15.2	51.8	78.0	95.0
BRP-NAS[a]	0.0358±7e-3	13.3±3.4	26.1±5.8	81.9±7.6	93.4±3.1	99.6±0.3
BRP-NAS++[b]	0.0225±3e-3	6.50±1.3	37.2±5.3	92.7±3.3	98.3±0.4	99.5±0.1
COBRA (Ours)	**0.0165±1e-3**	6.89±1.9	**45.3±2.7**	**96.2±1.6**	**99.0±0.4**	**99.8±0.2**

[a]A GCN based latency predictor, using layer representations consisting of the layer type and its parameters.

[b]A GCN based latency predictor, using layer representations consisting of the layer type, its parameters and the measured layer latency.

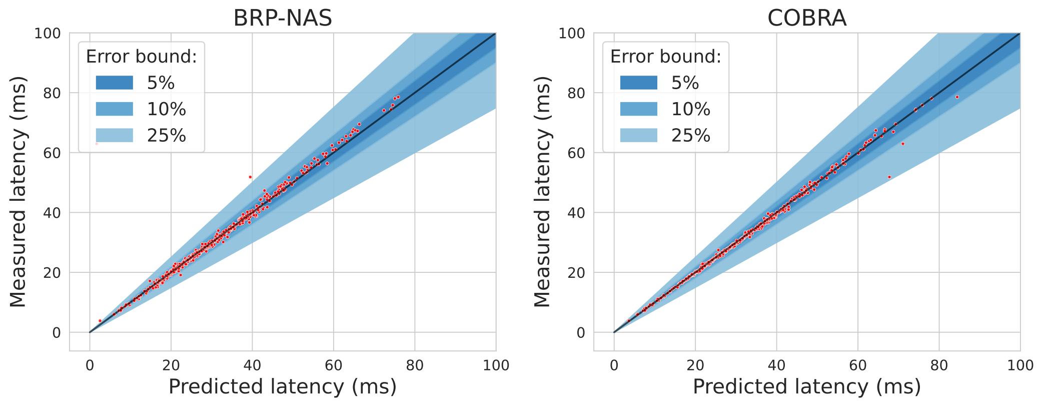


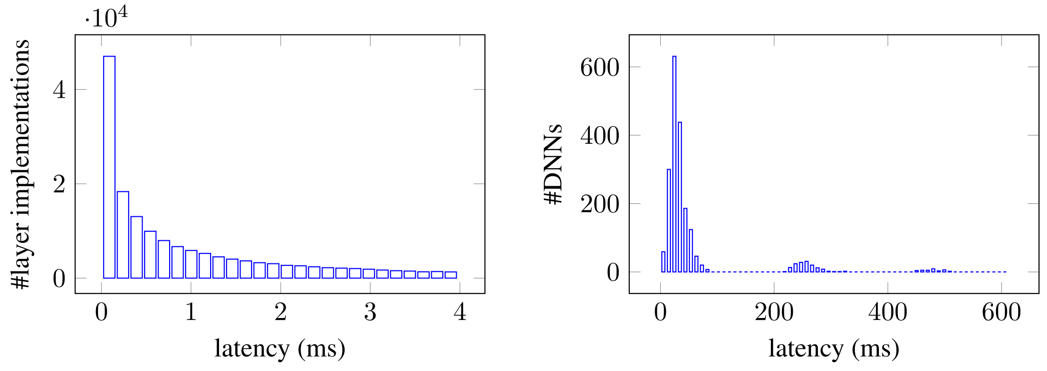
Figure 3: Predicted vs. measured latency for GCN based latency predictors with hand-crafted layer representations and COBRA. We do not show the plot of BRP-NAS++, because it is almost similar to BRP-NAS. Especially for low-latency networks the performance of COBRA is very good. Although the test set also contains networks with very large latency, we only show low latency models here, because the prediction errors are most pronounced for them.

details are given in Section A.1.1. The latency for each random parameter configuration is measured on an NVIDIA 1080TI GPU.

The second dataset is used to train the GCN and is constructed from randomly generated DNN code and the DNN latencies that have also been measured on an NVIDIA 1080TI GPU. In total, we collect a dataset of 2000 random ResNet-like models (He et al., 2016) and split the dataset into 1350 training, 150 validation and 500 test samples. ResNet-like means that we not only allow skip-connections between neighbouring, but also between any two arbitrary network layers. The number of network layers is random, but is never larger than 226. Also the individual layer types and their configuration, e.g., the number of output channels and the stride of a convolution, are chosen randomly. For more details about the degrees of freedom that we allow, please refer to Section A.2.

Figure 4 shows the histogram of the latencies in the collected datasets. Most of the collected layer implementations have a low latency. More specifically, 20% of them have a latency that is lower than 1ms and 80% have a latency that is lower than 4.3ms. However, the latency distribution is heavily tailed. In particular, the maximum latency of a single layer is 1524.2ms. Further, most collected DNNs yield a latency inferior to 200ms. However, similar to single layer implementations, the corresponding latency distribution is heavily tailed.

Performance of COBRA: In our first experiment, we compare the performance of COBRA to three other methods for DNN latency prediction, namely: 1) The layer-wise sum that estimates the latency of a DNN by summing up the measured latencies of the individual network layers. It is also calibrated by a scaling factor to fit the latencies in the training set. Although this method has access to the measured latencies of all network layers of a DNN, it is known to perform poorly. Hence, it acts as a lower baseline. 2) A GCN based predictor that uses hand-crafted layer representations,



(a) Latency per layer implementation. This histogram is truncated at 4ms, corresponding to 80% of the latencies. Note, that it actually extends up to 1350ms.

(b) Latency per DNN. This histogram is truncated at 600ms, corresponding to 99.4% of the latencies. Note, that it actually extends up to 969.2ms.

Figure 4: Latency distribution of the collected datasets.

consisting of the one-hot encoded layer type and all the layer parameters. This representation is proposed by Zhang et al. (2021). The GCN architecture is similar to Dudziak et al. (2020), hence we call this method BRP-NAS. 3) The GCN based predictor which uses layer representations that also contain the measured latency of each individual layer. We call this method BRP-NAS++. Although latency measurements of single layers can be costly to obtain in practice, we add this feature to see how good a GCN predictor can perform if it is provided with all available information.

To train the predictors, we first search for optimized hyperparameters on the validation set, using Optuna (Akiba et al., 2019). The best hyperparameters are summarized in Table 5 in the appendix. Each predictor is then trained 10 times, starting from different random initializations and using both the training and the validation set.

Table 1 summarizes the root mean square error (RMSE), the mean absolute percentage error (MAPE), as well as the error bound at 1%, 5%, 10% and 25% of each latency prediction method. More specifically, the error bound at $k\%$ measures for how many samples from the test set the latency is predicted with less than $k\%$ error (higher is better).

COBRA clearly outperforms layerwise sum, BRP-NAS and BRP-NAS++, when comparing the MAPE and the error bounds. In particular, COBRA yields less than 5% error for 96.2% of the test networks. Compared to BRP-NAS++ that yields less than 5% error for only 92.7% of the test networks, COBRA is better by 3.5%. For less than 1% error, COBRA outperforms BRP-NAS++ by 8.1%.

Note that the layerwise sum yields the lowest RMSE of all the methods. That is because BRP-NAS, BRP-NAS++ and COBRA are trained to minimize the MAPE, which measures the average relative prediction error. In practice, estimators that achieve a small relative prediction error are more desirable than those that achieve a low absolute prediction error. In particular, we want highly accurate estimates for low latency networks and allow larger estimation errors for networks with a large latency. This behaviour is not captured by the RMSE, because it measures the absolute prediction errors.

Performance of the transformer encoder: In our second experiment, we evaluate how accurate the transformer encoder alone can predict the latency of single layers from their source code implementation. Table 2 gives the latency prediction on the test set. We observe that the transformer encoder can accurately predict the latency of layer implementation. Note, that it yields the lowest prediction errors for simple layers like `ReLU`, `Add` or `Linear`, where an error of less than 25% is obtained for 100% of the test samples.

Further, the transformer encoder struggles to predict the latency of convolutional layers. A convolution is defined by the input tensor shape, the number of channels, the kernel size, the stride, the padding, the group, and the dilation. Compared to all the other considered layers, it has the largest number of parameters and therefore is the hardest to predict.

Table 2: Latency prediction errors for single layer implementations, using the transformer encoder.

Type	MAPE	RMSE (ms)	error bound (%)			
			±1%	±5%	±10%	±25%
`Convolution`	0.28	1.01	2.41	13.17	26.72	57.61
`BatchNormalization`	0.06	0.21	9.50	54.96	86.36	98.45
`ReLU`	0.05	0.23	11.46	56.35	90.56	100.0
`Add`	0.06	0.27	10.0	49.0	81.0	100.0
`Linear`	0.022	0.001	31.31	94.44	98.48	100.0
`AveragePooling`	0.09	0.52	5.26	47.37	68.42	97.37
`MaxPooling`	0.09	0.23	10.0	35.0	60.0	100.0
`GlobalAveragePooling`	0.06	5.19	11.11	51.39	84.72	99.31
All	0.15	0.88	8.62	40.26	61.07	81.44

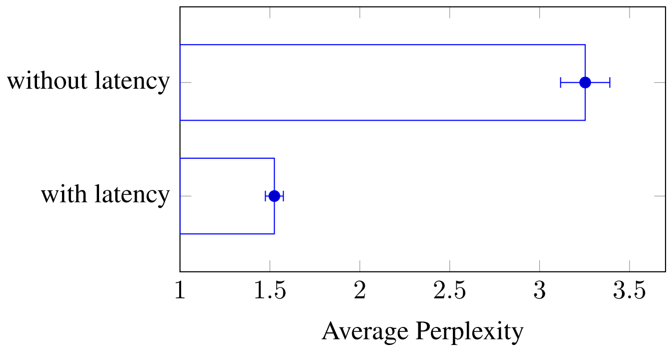


Figure 5: Average perplexity (PPL) of predicting masked parameter tokens for layer implementations, using the transformer encoder.

In our last experiment, we show why it is important to append the layer latency to the layer implementation, when pre-training the transformer encoder. More specifically, we train the transformer encoder twice, once with and once without the appended layer latency. For both, we report the average perplexity (PPL) for masked token reconstruction, assuming that only parameter tokens are randomly masked. This directly reflects how good the transformer encoder can capture the properties of dependence between the parameters, conditioned on a certain latency. As shown in Figure 5, the perplexity is lower if the layer latency is appended to the layer implementation. As argued before, all layer parameters are independent if we provide no latency information. Hence, it is impossible to predict them from the masked layer implementation. However, if we apply our training trick and also provide latency information, all layer parameters that influence the latency are conditionally dependent. This enables us to learn properties of dependence between the layer parameters and the layer latency. In particular, we can learn how masked parameters can be computed from others. Hence, it also yields a much lower perplexity.

5 Discussion and Conclusions

Our experiments demonstrated the feasibility of latency prediction directly from source code. In particular, we demonstrated that using a simple training trick, we can train a standard transformer encoder to predict DNN layer latencies from their source code implementation. Further, we could successfully use the layer embeddings generated by the transformer encoder to improve state-of-the-art GCN based latency predictors for DNNs. Our method COBRA has two advantages: 1) Compared to BRP-NAS, it uses learned, rather than hand-crafted features to encode individual function calls. Therefore, it is easy to extend it to new layer types. 2) Compared to BRP-NAS++, it does not require the measured latency of individual function calls, while still yielding a comparable performance.

Limitations of the current experiments: We are aware of some limitations of our current experimental setup that must be addressed in the future. Our current training and validation data is automatically generated and therefore heavily normalized. In particular, it does not capture the variability of a human authored source code, where there is for example lots of variations to the choice of variable names and how parameters are passed. Nevertheless, our experiments are very encouraging and demonstrate the feasibility of latency prediction from code. A study with human written code has to be conducted in the future to test the real-world performance of the system.

Due to time constraints, we only compared our method to BRP-NAS so far. Although BRP-NAS has proven to be a very good latency predictor for DNNs, fair comparisons to other state-of-the-art latency predictors like nn-Meter must be performed. Up to now, we also consider latency predictions for inference on a GPU. Therefore, the experiments must be extended to other platforms, such as embedded devices or mobile DNN accelerators.

Future directions of research: During our work on COBRA, we identified two research areas for machine learning on code that are in our opinion of general interest and therefore should be addressed in the future:

1) Compared to natural language, the behaviour of code heavily depends on numeric values such as bounds for counters in loops, indices when accessing data or parameters that are used for basic calculations. We think that current tokenizers from NLP typically lack the ability to represent these values in a suited way, what makes them not particularly well suited for machine learning on code. This issue was also raised by Geva et al. (2020). A simple way to address this issue could be to represent numbers with positional encodings. COBRA can most likely be improved, by using such ideas.

2) Another important challenge is dealing with extremely long sequences. As already discussed in the introduction, information about the behaviour of code is very distributed and can span multiple files and hundreds or even thousands of lines of code. Transformers cannot easily be scaled up to process such long sequences because the computational complexity of the attention mechanism grows quadratic with the sequence length. Therefore, we need good ways to scale these models up. One approach that we also used in this work is to split the code into fragments. Each fragment would be processed individually and the results are combined. However, we can also think of some kind of hierarchical processing with structured attention, where the attention is first computed across single lines of code, then across code blocks and finally across files.

REFERENCES

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *CoRR*, abs/2105.12787, 2021. URL `https://arxiv.org/abs/2105.12787`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. URL `https://arxiv.org/pdf/1812.00332.pdf`.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. URL `https://arxiv.org/pdf/1908.09791.pdf`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *CoRR*, abs/2001.00326, 2020. URL `http://arxiv.org/abs/2001.00326`.

Lukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Brp-nas: Prediction-based nas using gcns, 2020.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139`.

Mor Geva, Ankit Gupta, and Jonathan Berant. Injecting numerical reasoning skills into language models. In *ACL*, 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. Help: Hardware-adaptive efficient latency prediction for nas via meta-learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2815–2823, 2019.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

Chris Ying. Enumerating unique computational graphs via an iterative graph invariant. *CoRR*, abs/1902.06192, 2019. URL `http://arxiv.org/abs/1902.06192`.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 7105–7114, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL `http://proceedings.mlr.press/v97/ying19a.html`.

Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 81–93, New York, NY, USA, 2021. ACM. doi: 10.1145/3458864.3467882. URL `https://doi.org/10.1145/3458864.3467882`.

A Appendix

A.1 Training details

A.1.1 Transformer encoder

The transformer encoder is defined in a similar way to the BERT model (Devlin et al., 2019). The tokenizer and the positional encoding are the same, but the architecture is a lighter version of $\text{BERT}_{\text{BASE}}$, in particular with fewer layers and a smaller hidden size. All the hyperparameters of the transformer are summarized in Table 3.

Table 3: Training hyperparameters of the transformer encoder.

Number of layers	4
Hidden size	512
FFN inner hidden size	512
Attention heads	16
Max. sequence length	600
Batch size	64
Max epochs	200
Learning rate	0.001
Learning rate scheduler	Step decay
Learning rate decay factor	0.9 (per epoch)
Gradient clipping	0.5
Dropout probabillity	0.5
Masking probabillity	0.1
Adam ϵ	1e-08
Adam β_1	0.9
Adam β_2	0.999
$L2$ weight decay	0.0

We want to make the transformer learn the non-linearity dependencies between a specific layer implementation and its corresponding latency. We therefore generate a dataset composed of code snippets, where each snippet is the code implementation required to execute a layer with a specific configuration. To keep it simple, we use random sampling to select the layer configurations, i.e., for each sample we pick uniformly a random layer configuration on a specific range of values. The layers dataset is therefore very general, with various possible configurations and allowing even to have configuration with non-equal input width and height. Table 4 describes the different layer configurations as well as the number of samples for each type of layer. For each layer configuration, we also measure its latency.

A.1.2 GCN

We observe that a proper training of the different GCNs heavily depends on the values of the hyperparameters. We therefore use a hyperparameter optimization framework called Optuna (Akiba et al., 2019) to find good hyperparameters for each GCN. The list of all the hyperparameters is available in Table 5. In particular we observe that a batch size equal to 2 is giving the best performance on the test loss.

A.2 DNN dataset

We explain here more in details how the dataset of DNN architectures is built. The connections between layers are encoded by the adjacency matrix A. We consider therefore the set of all the binary upper triangular matrices of fixed size $N \times N$. Then, any operation of the set of predefined operations defined in Table 6 can be associated to a specific layer, with the objective of allowing various configurations of layers. The first layer and last layer are consecrated respectively to the input and the output.

One way to sample from this architecture space is just to draw uniformly an architecture from it, i.e., one adjacency matrix A and one operation per layer. However, to control the density of the drawn

Table 4: The different types of layers and configurations of the dataset of layer implementations used to train the transformer. H is the height, W the width, C the number of channels, K the kernel size, S the stride, P the padding, D the dilation, G the number of groups and M the number of tensors as input.

Type	# samples	Configuration space
`Convolution`	115,192	$H, W, C_{in}, C_{out} \in \text{range}(1, 1000)$ $K_1, K_2, S_1, S_2 \in \text{range}(1, 10)$ $P_1, P_2 \in \text{range}(0, 10)$ $D_1, D_2 \in \text{range}(1, 5)$ $G = \text{gcd}(C_{in}, C_{out})$
`BatchNormalization`	38,020	$H, W, C \in \text{range}(1, 1000)$
`ReLU`	64,734	$H, W, C \in \text{range}(1, 1000)$
`Add`	10,000	$H, W, C \in \text{range}(1, 1000)$ $M \in \text{range}(2, 10)$
`Linear`	20,164	$C_{in}, C_{out} \in \text{range}(1, 1000)$
`AveragePooling`	3,806	$H, W, C_{in}, C_{out} \in \text{range}(1, 1000)$ $K_1, K_2, S_1, S_2 \in \text{range}(1, 10)$ $P_1, P_2 \in \text{range}(0, 10)$
`MaxPooling`	2,108	$H, W, C_{in}, C_{out} \in \text{range}(1, 1000)$ $K_1, K_2, S_1, S_2 \in \text{range}(1, 10)$ $P_1, P_2 \in \text{range}(0, 10)$
`GlobalAveragePooling`	13,934	$H, W, C \in \text{range}(1, 1000)$

Table 5: Training hyperparameters of the different GCNs predictors obtained by the hyperparameter optimization framework.

	BRP-NAS	BRP-NAS++	COBRA
Number of layers	4	4	4
Hidden size	520	250	395
Batch size	2	2	2
Epochs	1500	1500	1500
Initial learning rate	0.001	0.001	0.001
Adam ϵ	1e-8	1e-8	1e-8
Adam β_1	0.9	0.9	0.9
Adam β_2	0.999	0.999	0.999
$L2$ weight decay	2e-5	2e-6	1e-6
Dropout ratio	0.0002	0.0008	0.0002

architectures, we introduce the probability p of having two arbitrary layers connected. Therefore large values of p involve more dense networks whereas $p = 0.5$ is uniform sampling. We fix $p = 0.4$ to avoid having almost only very dense networks. Once we have drawn the adjacency matrix and the operations of each layer, we perform a pruning step similar to in NAS-Bench-101 (Ying et al., 2019) to ensure that the resulting DNN is valid. First, we discard adjacency matrices which correspond to invalid architectures because of the input node and the output node being not connected. In addition,

Table 6: The different types of layer and configurations of the dataset of DNNs.

Type	Configuration

Conv+BatchNorm+ReLU	*num. channels* $\in$ {32, 64, 128, 256, 512}
	kernel size $\in$ {3, 5, 7}
	stride $\in$ {1, 2, 4}

Sepconv+BatchNorm+ReLU	*num. channels* $\in$ {32, 64, 128, 256, 512}
	kernel size $\in$ {3, 5, 7}
	stride $\in$ {1, 2, 4}

Avgpool2x2	$\varnothing$
Maxpool2x2	$\varnothing$

we remove useless dangling nodes, leading to architectures with very different sizes and therefore a wider range of latency than other dataset such as NAS-BENCH 201 (Dong & Yang, 2020). To avoid sampling the same architecture multiple times, we apply a graph hashing algorithm detecting whether two architectures are isomorphic or not (Ying, 2019). Finally, since we can add multiple layers together with different input shapes, we add convolution blocks (Conv+BatchNorm+ReLU) before adding layers to decrease the resulting input shape of each layer to the greatest common divisor among the different input shapes.