NNLQP: A Multi-Platform Neural Network Latency Query and Prediction System with An Evolving Database

Liang Liu[1*], Mingzhu Shen[1*], Ruihao Gong[1,2], Fengwei Yu[1], Hailong Yang[2†]
SenseTime Research[1], Beihang University[2]
China
{liuliang1,shenmingzhu,gongruihao,yufengwei}@sensetime.com,hailong.yang@buaa.edu.cn

ABSTRACT

Deep neural networks (DNNs) are widely used in various applications. The accurate and latency feedback is essential for model design and deployment. In this work, we attempt to alleviate the cost of model latency acquisition from two aspects: latency query and latency prediction. To ease the difficulty of acquiring model latency on multi-platform, our latency query system can automatically convert DNN model into the corresponding executable format, and measure latency on the target hardware. Powered by this, latency queries can be fulfilled with a simple interface calling. For the efficient utilization of previous latency knowledge, we employ a MySQL database to store numerous models and the corresponding latencies. In our system, the efficiency of latency query can be boosted by 1.8×. For latency prediction, we first represent neural networks with the unified GNN-based graph embedding. With the help of the evolving database, our model-based latency predictor achieves better performance, which realizes 12.31% accuracy improvement compared with existing methods. Our codes are open-sourced at https://github.com/ModelTC/NNLQP.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; **Hardware latency**; **Transfer learning**.

KEYWORDS

neural network, multi-platform, latency query, latency prediction

ACM Reference Format:
Liang Liu[1*], Mingzhu Shen[1*], Ruihao Gong[1,2], Fengwei Yu[1], Hailong Yang[2†]. 2022. NNLQP: A Multi-Platform Neural Network Latency Query and Prediction System with An Evolving Database. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3545008.3545051

1 INTRODUCTION

Deep neural networks (DNNs) have entered the era of large-scale deployment and application in various tasks, such as computer vision [12, 24, 31] and natural language processing tasks [5]. In practical industry production, the accuracy and efficiency of a DNN model are especially paid attention to. In recent years, numerous algorithm researchers and hardware designers have been devoted to improving the model efficiency while maintaining the model accuracy utilizing techniques like model compression [8, 20, 21, 29, 40, 43], neural network search [2, 34, 35], and special acceleration unit development [22, 25, 26]. Many of them focus on reducing the computation amount (FLOPS), params, or memory access of the model [23, 33]. However, these metrics are not good proxies for the latency feedback [7] since some hardware has unique designs (e.g., Tensorcore in Nvidia Turing GPU [6] and KB-level memory storage in STM MCU [23]). Therefore, only considering FLOPS, params, and memory access are not enough and an accurate and timely latency feedback is essential for model design and deployment. Considering hardware latency in DNN design can make better use of hardware features and greatly accelerate inference speed or improve accuracy, as proved by existing work [3, 23]. Currently, the mainstream latency acquiring methods could be classified into two categories: true latency evaluation on the target hardware or latency prediction with a meticulously designed predictor.

Collecting latency performance is difficult and limited by hardware availability. Besides, deploying massive neural networks from various hardware devices can also be expensive. Usually, deploying deep models onto multiple devices with different inference frameworks requires hardware expertise and many engineering efforts, which hinders the efficient latency acquisition. Moreover, the latency information acquired by such a complicated pipeline lacks persistent storage and thus cannot be reused effectively for a future query of the same network. Recently, some works propose latency benchmark and dataset which mainly focus on the NAS-Bench101 or NAS-Bench201 models within a relatively small search space such as HW-NAS Benchmark [19], BRP-NAS [7]. However, the datasets are static and no longer satisfy the practical need with the growing number of deployments.

For large-scale latency acquisition tasks such as Network Architecture Search (NAS) [2, 3, 35], it would be also unacceptable to acquire the latency of millions of models from the hardware. Therefore, latency prediction methods [3, 4, 7, 15, 16, 41] are proposed to collect latency with a relatively low cost within an acceptable error bound. There are two main challenges brought up with latency prediction: **(1)** Existing works (e.g., TPU-Performance [15, 16] and nn-Meter[41]) divide a whole model inference into multiple kernels which relies on the hypothesis that the latency of kernels can be summed up as the model latency. However, the hypothesis might be invalid due to hardware graph fusion and launch cost for each kernel and cause a gap that results in a large difference between the

*Equal contribution. †Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '22, August 29-September 1, 2022, Bordeaux, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545051

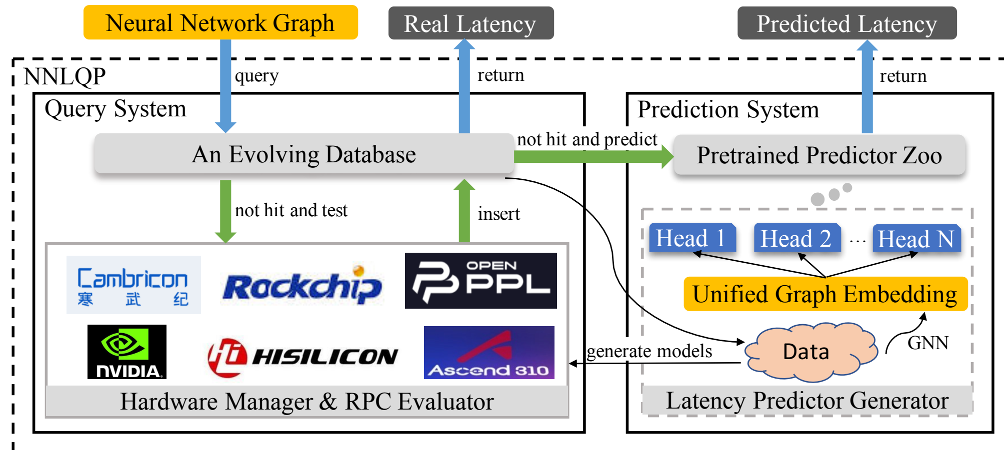


Figure 1: Overview of our multi-platform neural network latency query and prediction system (NNLQP). Bold colored arrows indicate the query/prediction procedure. Thin black arrows denote the scheme to obtain accurate and reusable latency predictors.

true latency and predicted latency (Figure 2). **(2)** Some recent works propose to predict model latency directly [7], but employ a simplified representation for the model in a specific search space [3, 7]. Thus some network topology information is missing and the representation method fails to fit another search space. Without a unified representation of neural networks, the large-scale latency information cannot be efficiently utilized to alleviate the problem of huge sampling space, thus resulting in poor prediction results.

To alleviate the aforementioned problems of latency acquisition and prediction, we propose a multi-platform neural network latency query and prediction system (NNLQP) with an evolving database, which is composed of the neural network latency query system (NNLQ) and the neural network latency prediction system (NNLP).

NNLQ takes the universal model format ONNX [1] as input, and automatically transforms ONNX into the corresponding executable format for supported hardware platforms. Then the latency evaluation task can be launched by a simple interface calling remotely without manually accessing the real-world device. To further utilize the plentiful history latency knowledge, we employ a MySQL database to store models and their latencies. A hash-based model encoding scheme is devised to realize efficient retrieval. If the latency of the current model is already stored in the database, NNLQ would return the hit record directly, which can significantly accelerate the latency acquisition process.

Benefiting from NNLQ, the latency knowledge of various models on various platforms is continuously accumulated in the database. With all latency data in our database, we can design and train a latency predictor with high accuracy and keep improving it for new model structures and platforms with the evolving database. NNLP can extract unified graph embeddings for deep neural networks, and predict model latency of multiple hardware platforms simultaneously.

To summarize, the contributions of our paper are three-fold:

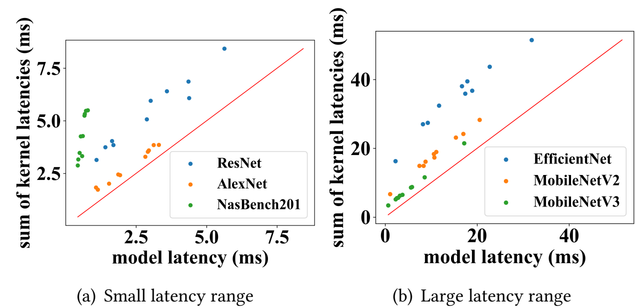
- NNLQP provides a model latency query and prediction system, which can give the latency feedback of neural networks on various hardware with a unified and efficient interface.
- The proposed predictor utilizes the plentiful and evolving data stored in the database of NNLQP and devises a unified graph embedding to avoid the inaccurate sum up of kernel latencies, improving the prediction accuracy by 12.31%.
- Equipped with the system, the overall efficiency of latency acquisition can be boosted by 1.8× and the accuracy gain for upstream NAS application can be at most 1.2% with much more accurate latency feedback.

2 RELATED WORK

Latency Benchmark and Dataset. TenSet [42] has been proposed to improve the performance of cost model for deep learning compilers such as TVM [4]. Recent works propose hardware benchmarks for the latency of the whole network such as BRP-NAS [7] and HW-NAS-Bench [19]. However, these datasets are static and hard to query and extend with the growing number of deployments.

Representation of Neural Networks. The representation of neural networks is important in latency prediction. Existing works usually embed different models into simple feature tensor. ProxylessNAS [3] encodes operation index for each node. BRP-NAS [7] only encodes the operation index and connections inside search cells (basic units in the NAS). OFA [2] encodes the choice of kernel size, depth, expand ratio, and resolution as a 128-dimension vector. All these NAS methods can only represent models inside their own search space. TPU-Performance [15, 16] and nn-Meter [41] divide the whole graph into different kernels and encode kernel features. The description in the model level is lacking and the representation cannot be generalized onto other architectures or search spaces, which would cause some network topology information go missing and further impact prediction results.

Latency Prediction Tool. Recent works realize the problem that getting hardware latency is highly expensive, especially for extensive models. Therefore, several works have been proposed to estimate latency at a low cost. **FLOPs-based:** the number of FLOPs or memory access (MAC) is usually used as a proxy for the actual latency on hardware devices, while these two metrics sometimes have a relatively low correlation with latency as explained in [7, 28]. **Lookup table-based:** Therefore, most NAS methods [3] use a lookup table to indicate the final latency of different models. Although it is highly efficient, it also has a large difference from the exact latency on target devices. **Learning-based methods:** Learning-based methods are becoming more popular, deep learning compilers like TVM [4] employs a performance model for an auto-tuner to evaluate candidate configurations in a search space. BRP-NAS [7] encodes the basic units in the NAS to represent the whole network which can not handle general cases for the model with different channels or depth settings.



(a) Small latency range (b) Large latency range

Figure 2: Kernel additivity validation. Points with different colors are all above the red line $y = x$. It indicates that the sum of the kernel latencies is greater than model latency.

3 BACKGROUND AND MOTIVATION

In this section, we summarize the obstacles for these two latency acquiring methods: true latency evaluation on the target hardware or latency prediction with a meticulously designed predictor.

3.1 Obstacles for Acquiring Model Latency

Multi-platform model latency query requires a complicated pipeline. With the increasing application scenarios, more and more DNN models need to be deployed on multiple platforms. It brings up new problems of obtaining the exact latency of multiple platforms quickly. Collecting the latency of DNN models from various hardware devices is a tedious process. First, the runtime environment of the hardware should be configured carefully. For example, suited versions of CUDA and TensorRT should be installed for a GPU machine with a specific driver. Then DNN models are transformed and compiled to the corresponding machine codes, which could cost a lot of time. Finally, machine codes of DNN models are evaluated on hardware devices. For multi-platform latency measurement, many engineering efforts are required, since inference libraries of different devices differ greatly. Moreover, limited by hardware availability, the exact latency feedback is even infeasible.

The model latency information lacks unified storage for later queries. Since the existing literature only collects latency data for a specific target, i.e., training and evaluating the predictor, they usually choose a simplified data format that neglects some information like the topology. This practice works for a small range of similar neural networks but will fail when facing neural networks out of the pre-defined scope. When the amount of neural networks increases, which is a common case in the upstream tasks (NAS), the dispersed and non-standard datasets can not be effectively reused, leading to a low query efficiency and obvious resource waste.

3.2 Limitations of Existing Latency Predictors

Recent prediction works rely on kernel additivity assumptions. They assume that kernels run sequentially and the model latency is equal to the sum of all its kernel latencies. However, this assumption is not always correct for some hardware platforms: (1) Some artificial intelligence chips have multi-stream mechanisms, and they could run multi-kernels parallelly; (2) Each kernel has its input and output tensors. For neighbor kernels, the memory access cost of input and output tensors may be overlapped while measuring latency; (3) Kernels are detected based on operator fusion rules only, but other graph optimizations may determine new kernels, such as operator rearrangement. Incomplete kernels may cause prediction bias. From Figure 2, we can conclude that the naive addition of kernel latencies can not effectively represent model latency (More discussions in Appendix A). Therefore, model-level latency prediction can be a better solution.

Model-based predictors rely on large-scale latency information. As aforementioned, acquiring large-scale latency information in a short time is non-trivial. Thus improving the prediction accuracy with a few data records become crucial. Recently some works [32] use FLOPS regression as an upstream pre-training task for efficient few-shot latency prediction. However, this task is easy to overfit and thus discounts the effect of transferring to latency prediction. What's more, even if we have accumulated enough latency data of different types of neural networks, the lack of general graph representation hinders current methods from taking advantage of all of them (illustrated in Section 2).

4 SYSTEM OVERVIEW

To overcome the aforementioned problems, we propose a neural network latency query and prediction system (NNLQP) with an evolving database, which is composed of the neural network latency query system (NNLQ) and the neural network latency prediction system (NNLP) as shown in Figure 1. NNLQ supports automatic model deployment and latency measurement for various platforms, and realizes permanent storage of model and latency with a MySQL database. NNLQ can also retrieve models and latency fast by a hash-based model encoding. Based on accumulated true latency records in our database, we train NNLP to predict latency fast. NNLP uses a GNN-based backbone to extract unified graph embeddings for DNN models, and attaches multi-heads to predict latency for multiple platforms simultaneously. Benefiting from the shared backbone and transfer learning, NNLP can effectively improve the latency prediction accuracy of unseen structures and unseen platforms.

Table 1: Supported platforms in our NNLQ.

Type	Hardware	Software	Data Type
GPU	T4/P4	trt5.0/trt7.1	fp32/fp16/int8
CPU	cpu	openppl	fp32
ASIC	hi3559A	nnie11	int16/int8
ASIC	hi3519A	nnie12	int16/int8
ASIC	atlas300	acl	fp16/int8
ASIC	mlu270	neuware	int16/int8
ASIC	rv1109	rknn	int16/int8

5 NEURAL NETWORK LATENCY QUERY

To ease the difficulty of model latency acquisition on multiple platforms and utilize large-scale latency information efficiently, we propose a neural network latency query system NNLQ. We introduce NNLQ from two aspects: multi-platform model latency acquisition, and model storage and latency query.

5.1 Multi-Platform Model Latency Acquisition

The proposed NNLQ can perform automatic model deployment and latency measurement on multiple platforms. Users provide ONNX models and target platforms as query input, and NNLQ returns the true latency on the target hardware through the following steps:

Step 1: model transformation. ONNX models are automatically converted into the corresponding graph descriptions that are suited for the inference library of the target platform, such as INetworkDefinition in the TensorRT toolkit. Then the inference toolkit is used to compile switched models into executable codes and pack all dependent libraries.

Step 2: device acquisition. The system manages various hardware devices through the remote procedure call (RPC) interface, and if there are idle devices for the target platform, the system acquires the control right of the device. We support many devices with different types as shown in Table 1 which covers a wide range of CPU, GPU, and ASIC devices. For details of platforms, please see Appendix B.

Step 3: latency measurement. The system uploads the executable code and its dependent libraries to the target hardware and runs the model to get latency. The exact execution time will be returned if successful or error messages will be returned if failed. After the latency measurement is completed, the control right of the device is released. The latency measurement system runs on the server, which can work continuously and improve the utilization of devices.

5.2 Model Storage and Latency Query

Our NNLQ employs MySQL as a storage database to provide unified storage of DNN models and their latencies. To achieve fast retrieval of DNN models, we propose a graph hash encoding method, which encodes the model into a unique hash representation based on the model structure and operator attribute information.

The graph hash encoding: The deep neural network can be regarded as a directed acyclic graph (DAG) of operator nodes: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. First, the hash encoding H_v of node $v \in \mathcal{V}$ is defined as follows:

$$H_v = f_{hash}(f_{sort}(A_v) \oplus f_{sort}(\{H_u | u \in Suc(v)\})) \tag{1}$$

where A_v denotes the set of node attributes, $Suc(v)$ denotes the successor nodes of node v, f_{sort} is the sort function, f_{hash} is the hash encoding function, and $\oplus$ is the Concat function. The node hash encoding is determined by its attribute values and the hash encodings of all its successor nodes. So we can obtain a unique hash encoding of the node v. We calculate the node hash encodings in reverse topological order. It is guaranteed that for each node, its encoding is computed after all its successors. Then we define the hash encoding of the whole network:

$$H_{\mathcal{G}} = f_{hash}(f_{sort}(\{H_u | Pre(u) = \emptyset\})) \tag{2}$$

of which, $Pre(u)$ represents the predecessor node set of node u, and $Pre(u) = \emptyset$ represents that node u does not have any predecessor node, and it is the node connected with input. So we can get the unique hash encoding of the DNN model $H_{\mathcal{G}}$. The graph hash computation is shown as the left part of Figure 3. For models like recurrent neural networks (RNNs) with cycles, the loop will finally be unfolded. So they are still DAGs and can be encoded by our hashing method. The encoding value characterizes the network topology and node attributes. The same node hash encoding means that the sub-graphs composed of its successor nodes are the same. By comparing hash encodings, we can quickly distinguish whether two models contain the same network structure and node attributes.

Model-based database: Our MySQL database can store latency records of various DNN models for multiple platforms. The Entity-Relationship (ER) diagram is shown as Figure 4. We define a model table to store DNN models (ONNX format without weights), and save the graph hash encoding to realize the fast retrieval. Regardless of large networks or small ones, the graph hash key is always stored with 8 bytes. Each model record uses the storage of hundreds of bytes. In the platform table, hardware name, software name, and data type are stored. Each platform record is stored with 152 bytes. Besides, we defined a latency table to store latency information including cost, batch size, and memory access. The latency table adopts model id and platform id as its foreign keys. We also store the host/device memory for future analysis. Each latency record is stored with 52 bytes. As the latency query task grows, the database can continuously accumulate model latency knowledge.

Latency query: NNLQ provides a caching mechanism for neural network latency. When users query a latency, the system will respond quickly if the record hits inside the database, otherwise, the system will launch the latency measurement on hardware. Therefore, NNLQ can make good use of history latency knowledge and provide users with a query-able latency database.

6 NEURAL NETWORK LATENCY PREDICTION

Benefiting from the NNLQ system, we can continuously accumulate latency knowledge of various models on various platforms. With all these stored latency data, we can design and train a latency predictor within reasonable error bounds.

6.1 Unified Graph Embedding

To predict model latency accurately, it is important to extract features of the neural network. We propose a unified graph embedding

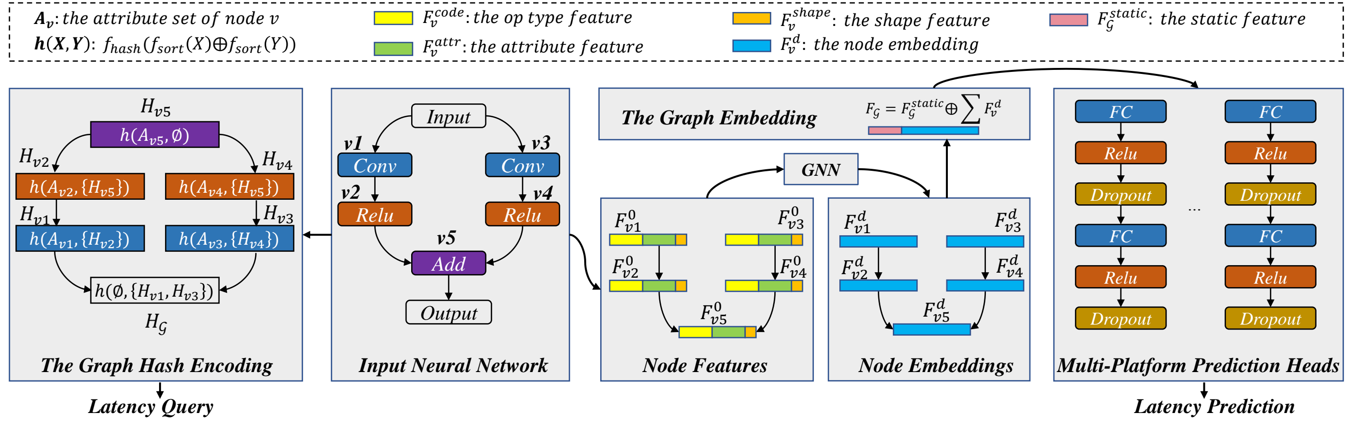


Figure 3: Latency query and latency prediction in NNLQP. The figure shows the graph hash encoding computation, the graph embedding computation, and multi-platform prediction heads.

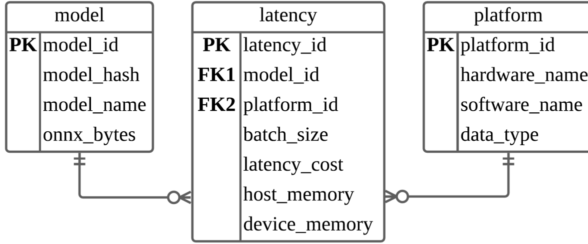


Figure 4: The Entity-Relationship diagram of the database in NNLQ. PK is primary key and FK is the foreign key.

that can be the used as the representation of operators, kernels, and neural networks. The method takes the ONNX model $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as input, then extracts node features, computes GNN-based node embeddings, and gets the unified graph embedding. The computation step is shown as the right part of the Figure 3.

Node features. We first extract node features, and the node feature of $v \in \mathcal{V}$ is defined as follows:

$$F_v^0 = F_v^{code} \oplus F_v^{attr} \oplus F_v^{shape} \tag{3}$$

where F_v^{code} is the one-hot vector indicating the node operator type, F_v^{attr} is the node attribute vector including the kernel size, stride, and other fields, F_v^{shape} is the output shape encoding, and $\oplus$ is the vector concatenation operation. Node features cover factors that affect the operator latency. For example, different types of operators, different attribute values, and different sizes of feature maps lead to different latencies. In practice, we calculate F_v^{attr}, F_v^{shape} by applying the mean and variance for normalization.

GNN-based node embeddings. Then we extract node embeddings with the graph neural network (GNN). GNN integrates the features of neighbor nodes by an aggregation algorithm and several GNN layers. We use an inductive method GraphSAGE [11] that can generate efficient node embeddings for unseen data. With d SAGEConv layers, the node embedding of the i-th layer is defined as follows:

$$F_v^i = L_2(W_1^i \cdot F_v^{i-1} + W_2^i \cdot \sum_{u \in \mathcal{N}(v)} F_u^{i-1}) \tag{4}$$

where W_1^i, W_2^i denote two parameters of the i-th feed forward function, $\cdot$ denotes the matrix multiplication, L_2 denotes the l2-normalized function, $\mathcal{N}(v)$ denotes the neighbor node set of node v, and $\sum$ denotes the mean aggregation method. When $i = 0$, F_v^i is the node feature defined as Equation 3.

The graph embedding. Finally, we can define the unified graph embedding of $\mathcal{G}$:

$$F_{\mathcal{G}} = F_{\mathcal{G}}^{static} \oplus \sum_{v \in \mathcal{V}} F_v^d \tag{5}$$

where $F_{\mathcal{G}}^{static}$ is the overall static feature that is determined by the original model and contains four values: batch size, FLOPs, parameters, and memory access, $\sum$ is the accumulation function, which reduces node embeddings and obtains the graph feature with a uniform length. By concatenating the static feature and the graph feature, we can get a unified graph embedding of a DNN model.

6.2 The Multi-Platform Predictor and Transfer Learning

With large-scale latency information stored in the database, we can train latency predictors. The proposed predictor NNLP uses the shared GNN backbone to extract the graph embedding and attaches multi-heads to predict latency for multiple platforms. So it is convenient to train the predictor with single-platform training, multi-platform training, and transfer learning.

Single-platform training. We can define a dataset $\{(\mathcal{G}_i, y_i, P)\}$, where y_i denotes the latency ground truth on platform P of model $\mathcal{G}_i$. Given model $\mathcal{G}$, the unified graph embedding $F_{\mathcal{G}}$ is extracted by the GNN-based encoder $f(\mathcal{G}; \alpha)$, where α is the parameter to be learned. The prediction head $g(F_{\mathcal{G}}; \beta)$ predicts the latency on platform P, where β is the parameter to be learned. As shown in Figure 3, the prediction head is composed of Fully Connected (FC)

Algorithm 1: Training multi-platform latency predictors

Input: platforms $\{P_1, P_2, ..., P_m\}$; training dataset
$\mathcal{S} = \{(\mathcal{G}_1, y_1, p_1), ..., (\mathcal{G}_i, y_i, p_i), ..., (\mathcal{G}_n, y_n, p_n)\}$
Output: $\alpha, \beta_{P_1}, \beta_{P_2}, ..., \beta_{P_m}$
for $\mathcal{G}_i, y_i, p_i \in \mathcal{S}$ **do**
  Extract the graph embedding: $F_{\mathcal{G}_i} = f(\mathcal{G}_i; \alpha)$;
  Predict latency on platform p_i: $y_i' = g(F_{\mathcal{G}_i}; \beta_{p_i})$;
  Compute loss: $\mathcal{L}_i = (y_i - y_i')^2$;
  Update $g(;\beta_{p_i})$ by gradient $\frac{\partial \mathcal{L}_i}{\partial \beta_{p_i}}$;
  Update $f(;\alpha)$ by gradient $\frac{\partial \mathcal{L}_i}{\partial \alpha}$.

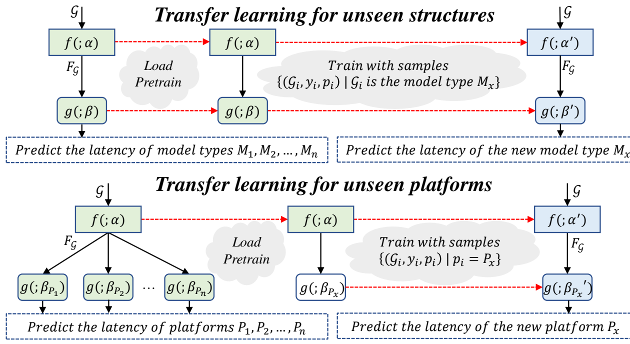


Figure 5: The transfer learning in NNLP. $f(;\alpha)$ is the shared GNN-based backbone. $g(;\beta)$ is the prediction head.

layers, Relu layers, and Dropout layers. While training, the mean square error (MSE) function $\mathcal{L} = (y_i - y_i')^2$ is used to compute loss.

Multi-platform training. As our evolving database accumulates latency from multiple platforms, we can investigate multi-platform latency prediction. As shown in Figure 3, our NNLP realizes the multi-platform latency prediction by the shared GNN-based encoder $f(;\alpha)$ and multi-heads $g(;\beta_{P_1}), g(;\beta_{P_1}), ..., g(;\beta_{P_m})$. The multi-platform training strategy is displayed in Algorithm 1. For each sample $(\mathcal{G}_i, y_i, p_i)$, $p_i \in \{P_1, P_2, ..., P_m\}$ is the target platform. While training, each iteration will update the parameters of both the shared GNN-based encoder and the predictor head for platform p_i. The proposed multi-platform latency prediction method can not only reduce predictor number for different platforms but also share model structure knowledge among different platforms.

Transfer learning. Since our NNLP contains a GNN-based backbone that can be shared, it is convenient to perform transfer learning for unseen structures and unseen platforms. The transfer learning in NNLP is shown as Figure 5. For transfer learning on an unseen structure type M_x, we first load pre-trained parameters for $f(;\alpha)$ and $g(;\beta)$, and then fine-tune parameters α', β' with a new sample set $\{(\mathcal{G}_i, y_i, p_i) | \mathcal{G}_i \text{ is the model type } M_x\}$. For transfer learning on an unseen platform P_x, we first load the pre-trained parameters of the multi-platform predictor for $f(;\alpha)$, and then fine-tune parameters α', ${\beta_{P_x}}'$ with a new sample set $\{(\mathcal{G}_i, y_i, p_i) | p_i = P_x\}$. Compared with learning from scratch, transfer learning enables training with fewer new samples and saving the training cost.

7 UNIFIED INVOKING INTERFACE

We implement NNLQP based on NetworkX [10], Flask [9], and Pytorch [27]. We adopt NetworkX to extract node attributes and topology information of DNN models, employ Flask to implement serving logic, and use Pytorch to construct our latency predictor. The NNLQP consists of 10,573 lines of Python code with 8,334 lines for NNLQ and 2,239 lines for NNLP. NNLQP provides users with a unified latency query and prediction interface as follows:

```
import NNLQP
params = {
    "model_path": "model.onnx",
    "batch_size": 1,
    "platform_name": "cpu-openppl-fp32",
}
true_latency = NNLQP.query(**params)
pred_latency = NNLQP.predict(**params)
```

where *params* include the ONNX path, batch size, and platform name. Users can query the real latency by invoking the interface *NNLQP.query* and predict multi-plaform latency by invoking the interface *NNLQP.predict*.

8 EXPERIMENTS

8.1 Experimental Setup

Dataset. We construct a latency dataset to evaluate the performance of latency prediction methods. The constructing approach is similar to nn-Meter. We select 9 state-of-the-art DNN models, AlexNet [18], VGG [36], GoogleNet [37], ResNet [12], SqueezeNet [14], MobileNetV2 [33], EfficientNet [39], MobileNetV3 [13], MnasNet [38] and then transform each one to get 2,000 variants with various kernel sizes and output channels. 2,000 models with the highest accuracy in NASBench201 are also added to our dataset. Therefore, the collected dataset contains 20,000 ONNX models, of which 18,000 models share 9 kinds of topology structures, another 2,000 models have different topologies. We then use NNLQ to perform automatic multi-platform latency measurement. NNLQ runs each model 50 times on the target platform and takes the average result as the latency ground truth.

Setup. The proposed unified graph embedding model and latency predictor are implemented using Pytorch. During training, the Adam [17] method is used to optimize parameters, the learning rate is set to 0.001. The batch size is 16 and the average loss of the batch samples is adopted as the basis for backward propagation. Our predictors are trained on a machine with a GTX1660 GPU.

Metrics. To evaluate deviations between latency predictions and ground truths, we use Mean Absolute Percentage Error (MAPE) and Error Bound Accuracy (Acc(δ)). MAPE denotes the mean absolute percentage deviation and the lower is better. Acc(δ) denotes the percentage of samples whose predictions and ground truths are within an error bound δ and the higher is better. (see Appendix C)

8.2 System Efficiency

We first analyze the cost of obtaining the model latency through different methods in NNLQP: query true latency by NNLQ and predict latency by NNLP. The NNLQ can cache history latency results, and it returns the hitting record directly. Only queries without hit

Table 2: The comparison of the time cost of querying model latency and predicting model latency. The test is based on 100 models and 9 platforms. Hit-$a\%$ means that $a\%$ of queries already stored in the database, and the remaining $1-a\%$ of queries need to be measured on actual hardware. We also compare the cost of our NNLP and the prior predictor FLOPs+MAC.

Platform	Cost of Query or Predict Latency(s)					Speedup Relative to Query-Hit-0%			
	Hit-0%	Hit-50%	Hit-100%	FLOPs+MAC	NNLP	Hit-50%	Hit-100%	FLOPs+MAC	NNLP
cpu-ppl2-fp32	15734.8	8788.37	219.54	10.35	11.04	1.79	71.67	1520.09	1425.36
hi3559A-nnie11-int8	9212.54	5359.11	197.55	10.55	11.2	1.72	46.63	873.19	822.7
gpu-T4-trt7.1-fp32	8490.74	4666.53	197.23	7.73	8.06	1.82	43.05	1098.93	1053.33
gpu-T4-trt7.1-int8	8111.45	4248.67	207.75	8.27	8.61	1.91	39.04	980.81	942.24
gpu-P4-trt7.1-fp32	9007.61	4853.23	171.04	8.73	9.21	1.86	52.66	1032.21	978.4
gpu-P4-trt7.1-int8	9077.02	5126.96	232.67	8.94	9.73	1.77	39.01	1015.87	933.27
hi3519A-nnie12-int8	8991.36	5432.36	196.54	9.29	10.2	1.66	45.75	968.26	881.37
atlas300-acl-fp16	11819.63	6333.05	151.47	10.77	11.34	1.87	78.03	1097.64	1042.39
mul270-neuware-int8	11116.97	5636.52	162.42	9.85	10.77	1.97	68.44	1128.86	1031.79
Average	10173.57	5604.98	192.91	9.38	10.02	1.82	52.74	1084.06	1015.62

will invoke the latency measurement on the hardware device. We choose 100 DNN models from 10 families and obtain their latencies on 9 platforms. We randomly select 0%/50%/100% to evaluate the query efficiency of different hit ratio. So the sizes of models are relatively uniform and will not have biased distribution.

The total time cost of different methods is shown in Table 2. It reveals that: (1) The database cache mechanism in NNLQ can effectively improve the efficiency of true latency acquisition. On average, the speedup ratio of Hit-100% to Hit-0% is 52.74 times. We analyze the actual hit ratio of the current system which is about 53%, so the overall speedup is about 1.8. Up to now, our NNLQ stores 63 platform records, 200k+ model records and 700k+ latency records. The total database size is about 10GB; (2) The average speedup with latency prediction in NNLP reaches 1015 times relative to Hit-0%; (3) The latency prediction is faster than Hit-100% (10.02s vs 192.91s). Because the query requires calculating the graph hashing using CPU, accessing the remote database, and checking if the same record existed in the database, while prediction can be conducted locally with GPU. So the query is much slower; (4) The prediction cost of NNLP is close to that of FLOPs+MAC (10.02s vs 9.38s). Compared with the less complex method FLOPs+MAC, our NNLP contains the GNN structure. The GNN is computed on Nvidia GPU and the inference is efficient.

8.3 Comparison with Related Works

We use latencies of 20,000 models on the platform gpu-gtx1660-trt7.1-fp32 as the dataset. To validate whether a method can predict efficient results for unseen structures, samples inside train and test contain different model types. For model-based methods FLOPs, FLOPs+MAC, BRP-NAS [7], and our NNLP, the ONNX graph and its latency are enough to training. However, for kernel-based methods nn-Meter [41] and TPU [16], a kernel dataset is required. We split 20,000 graphs into 14 kinds of kernels based on fusion rules. The statistics of the split kernels are shown in Appendix D. For each kernel family, we randomly select 2,000 or 1,000 (only for the family whose number is less than 2,000) kernels to obtain latency by NNLQ, and then split them by the ratio of 7:3 for train and test, respectively. The details of compared methods are shown in Appendix E.

The comparison of different latency prediction methods is shown in Table 3. The NNLP achieved the best average prediction effect with average MAPE 10.66%, which is 4.69% lower than that of the second-best method nn-Meter. The average Acc(10%) of NNLP is 59.73%, which is 12.31% higher than that of nn-Meter. The FLOPs method gets the worst performance. However, FLOPs+MAC gets 10.44% MAPE improvement relative to FLOPs. It shows that memory access is an important factor that determines model latency for the current platform. Even though nn-Meter and TPU get the best performance of some model families, such as MobileNetv2 and MobileNetV3, NNLP performs better than nn-Meter and TPU overall. Why can not kernel-based methods realize satisfying results? Because the relationship between model latency and the sum of kernel latencies varies with the model type. Our NNLP uses the unified graph embedding directly, avoiding the problem of unreliable additivity. BRP-NAS performs worse than NNLP, even though it utilizes the same node features as NNLP. It shows that BRP-NAS can not extract useful graph embedding of the entire model.

8.4 Ablation Analysis

To measure the impact of different components in our unified graph embedding method, we conducted ablation analysis: (1) wo/F_v^0: to check the effect of node features, therefore only the static features are used for latency prediction; (2) wo/gnn: to check the effect of the GNN model, we remove the GNN and therefore the node feature is directly used as the node embedding $F_v^d = F_v^0$, and then reduce node embeddings into graph embeddings as Equation 5 to predict latencies; (3) wo/$F_{\mathcal{G}}^{static}$: to check the effect of graph static features, we remove $F_{\mathcal{G}}^{static}$ and only use F_v^d for latency prediction.

As illustrated in Table 4, MAPE of the model wo/F_v^0 is 31.61%, which has a big performance drop. Compared with predictors involved GNN-based node embeddings, the prediction performance of wo/gnn is significantly decreased (MAPE increases from 10.66% to 25.15%), which shows that GNN can extract effective latency-related features. The model wo/$F_{\mathcal{G}}^{static}$ results in MAPE 23.59%. It indicates that the global static features contribute much to latency prediction. The prediction accuracy of wo/F_v^0, wo/gnn, and wo/$F_{\mathcal{G}}^{static}$ gradually increases. It shows that for the importance of latency prediction, node features > GNN > static features. For all model families, the whole NNLP achieves the best performance compared with other methods, which further verifies that our designed

Table 3: Comparison with related works such as nn-Meter [41], TPU [16], BRP-NAS [7] on 10 different model families. The 'Model Family = ResNet' denotes that only ResNet [12] models are used for test and all the other models are used for train.

Metric	Model Family	FLOPs	FLOPs+MAC	nn-Meter	TPU	BRP-NAS	**NNLP**
MAPE ↓	ResNet	21.18%	18.91%	15.58%	20.05%	15.84%	**6.75%**
	VGG	69.34%	66.63%	19.47%	38.73%	30.95%	**7.79%**
	EfficientNet	58.36%	53.96%	18.93%	**16.74%**	51.97%	21.33%
	MobileNetV2	37.42%	35.27%	**6.43%**	12.68%	20.42%	6.93%
	MobileNetV3	64.64%	57.13%	35.27%	**9.97%**	58.13%	16.57%
	MnasNet	40.31%	35.96%	10.69%	11.61%	17.26%	**9.45%**
	AlexNet	44.65%	15.45%	**7.20%**	10.55%	31.68%	9.74%
	SqueezeNet	29.89%	23.19%	18.69%	24.60%	42.55%	**8.45%**
	GoogleNet	30.76%	32.54%	11.71%	**8.10%**	25.48%	10.83%
	NasBench201	80.41%	33.52%	9.57%	58.94%	13.28%	**8.76%**
	Average	47.70%	37.26%	15.35%	21.20%	30.76%	**10.66%**
Acc(10%) ↑	ResNet	26.50%	29.80%	39.45%	27.30%	39.80%	**78.25%**
	VGG	4.80%	2.10%	26.50%	2.60%	13.20%	**70.45%**
	EfficientNet	0.05%	0.05%	23.40%	17.00%	0.10%	**24.65%**
	MobileNetV2	6.90%	8.05%	**80.75%**	33.95%	29.05%	76.00%
	MobileNetV3	0.05%	0.05%	23.45%	**64.25%**	13.85%	35.55%
	MnasNet	6.20%	9.80%	60.95%	44.65%	34.30%	**63.10%**
	AlexNet	6.55%	40.50%	**75.45%**	57.10%	15.20%	62.35%
	SqueezeNet	16.10%	21.35%	36.20%	25.65%	11.85%	**65.90%**
	GoogleNet	12.75%	9.80%	47.40%	**69.00%**	12.55%	50.05%
	NasBench201	0.00%	10.55%	60.65%	2.50%	43.45%	**67.10%**
	Average	7.99%	13.20%	47.42%	34.40%	21.34%	**59.73%**

Table 4: Ablation study with different graph embedding methods. The metric is MAPE↓.

Model Family	**NNLP**	wo/F_v^0	wo/gnn	wo/$F_{\mathcal{G}}^{static}$
ResNet	**6.75%**	16.21%	18.62%	11.98%
VGG	**7.79%**	64.24%	30.23%	33.50%
EfficientNet	**21.33%**	36.28%	22.34%	46.10%
MobileNetV2	**6.93%**	20.57%	24.42%	12.72%
MobileNetV3	**16.57%**	54.92%	28.59%	18.76%
MnasNet	**9.45%**	22.58%	28.18%	20.67%
AlexNet	**9.74%**	50.40%	15.99%	30.46%
SqueezeNet	**8.45%**	16.62%	23.20%	14.21%
GoogleNet	**10.83%**	20.76%	18.13%	29.36%
NasBench201	**8.76%**	13.54%	41.76%	18.12%
Average	**10.66%**	31.61%	25.15%	23.59%

graph embedding benefits from F_v^0, GNN , $F_{\mathcal{G}}^{static}$, and shows great advantages in latency prediction.

8.5 Method Universality

Can NNLP predict the latency of the kernel? The proposed NNLP takes ONNX as input, so it can be applied to different levels of neural networks, such as ops, sub-graphs and whole networks. We also perform the kernel latency prediction by NNLP and compare results with nn-Meter and TPU which predict latency based on kernels as shown in Table 5. From the table, our NNLP can predict latency results for kernels, with slightly better performance 7.67% than nn-Meter 8.33% and TPU 8.01% in terms of MAPE. NNLP can extract efficient latency-related features for kernels containing few nodes.

Table 5: Comparison of different kernel latency prediction methods. The MAPE↓ metric is presented.

Kernel Family	nn-Meter	TPU	**NNLP**
AveragePool	**4.89%**	7.39%	6.94%
Concat	5.01%	3.22%	**2.64%**
Conv+Add+Relu	5.09%	4.99%	**4.93%**
Conv+Add	5.24%	**4.36%**	4.86%
Conv+Clip	7.21%	7.98%	**6.37%**
Conv+Relu	11.35%	**9.34%**	10.73%
Conv	18.55%	**11.84%**	13.25%
Flatten	15.13%	13.56%	**12.54%**
Gemm	**5.06%**	7.63%	6.99%
GlobalAveragePool	**4.94%**	6.79%	6.01%
MaxPool	**2.46%**	5.71%	4.69%
ReduceMean	7.91%	**7.37%**	7.53%
Relu	17.44%	14.79%	**13.97%**
Sigmoid+Mul	6.29%	7.17%	**5.96%**
Average	8.33%	8.01%	**7.67%**

Can NNLP predict the model latency of multiple platforms simultaneously? Our NNLP method supports multi-platform joint training. It shares the unified graph embedding and adopts multi-heads to predict latency for multiple platforms simultaneously. To validate the effectiveness of multi-platform prediction, we compare the performance of multi-models and a single-model with multi-heads. We use head in our predictor to denote the task specific part. One head predicts the latency of one hardware. In the experiment, we selected 9 hardware platforms for test. For each platform, we query latencies of 2,000 models from the database, and samples are divided into train and test by the ratio 7 : 3. As Table 6, the average performance of multi-models and single is close: The

Table 6: Comparison of multi-platform latency prediction by the multi-models and single-model with multi-heads. The metric is Acc(10%)↑.

Platforms	Multi-models	Single-model
cpu-openppl-fp32	89.92%	90.18%
hi3559A-nnie11-int8	96.08%	94.58%
gpu-T4-trt7.1-fp32	84.64%	86.67%
gpu-T4-trt7.1-int8	76.82%	69.96%
gpu-P4-trt7.1-fp32	83.14%	82.45%
gpu-P4-trt7.1-int8	80.12%	76.76%
hi3519A-nnie12-int8	93.99%	93.35%
atlas300-acl-fp16	70.51%	71.47%
mul270-neuware-int8	50.18%	50.18%
Average	80.60%	79.51%

Acc(10%) is 80.60% vs 79.51%. The latency prediction accuracy of each platform is also close. Besides, we compare the prediction cost of multi-models and single-model. While predicting the latency of 100 models on 9 platforms, the multi-models costs 93.41 seconds in total, and the single-model costs only 10.59 seconds. Compared with the multi-models, the single-model saves about 9 times of prediction cost. It shows that our NNLP can reduce the predictor quantity for multi-platform latency prediction, shorten the training cycle, and achieve performance the same as multi models. It also indicates that NNLP can extract platform-independent latency features of DNN models, which can better guide multi-platform joint training.

8.6 Transfer Learning with Pre-trained Model

Transfer learning for unseen structures. NNLP is a model-level latency prediction method, and test models are totally unseen while training. In some scenarios, with the help of few samples containing unseen model types, we expect that predictors can get better performance. With the pre-trained model, NNLP can transfer the model knowledge to an unseen type and achieve better results. When the test model family is ResNet, we train the pre-trained model with 18,000 models containing all other 9 families. Then we apply the pre-trained model to initialize weights and use extra ResNet samples for NNLP fine-tuning. We compare the prediction results of transfer learning with pre-trained model and general learning that trains from scratch. Figure 6 shows the comparison of 5 model families. For all model families, all curves trend upwards. As the training samples increases, the prediction accuracy of NNLP continues to improve. Orange curves are above the blue one. It indicates that for different amounts of training samples, transfer learning always achieves better performance. Latency knowledge learned by NNLP can be well transferred to unseen models. The pre-trained model can help NNLP improve the upper limit of prediction accuracy. Besides, the fewer training samples, transfer learning can achieve greater improvement. For ResNet, when the number of training samples is 32, the accuracy improvement is 30.8%, but when the number is 1,000, the improvement is 1.7%. It shows that transfer learning is more useful with the support of few training samples of unseen structures. For comparison, we also performed transfer experiments with the prior method FLOPs+MAC, please see Appendix F for details.

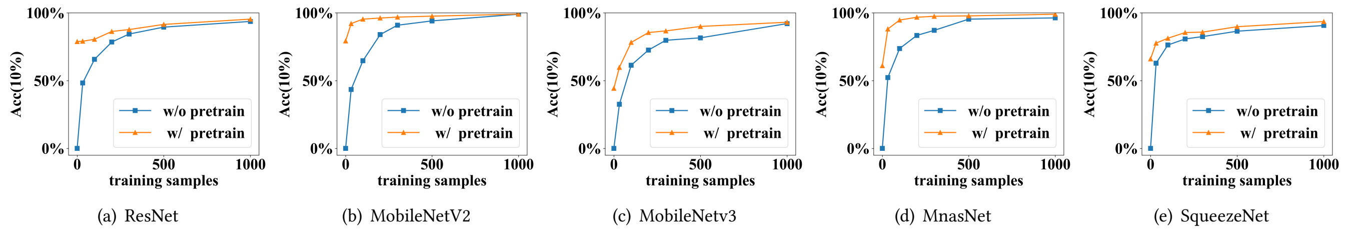
Transfer learning for unseen platforms. NNLP can also be applied to transfer learning for unseen platforms. The experiment is based on 9 platforms. For the target platform, we train the multi-heads pre-trained model based on the other 8 platforms. Then we use a certain number of target platform samples to fine-tune the predictor. We compare the prediction results of the model that is applied pre-trained model or not. Figure 7(a)-(d) display results of 4 platforms. The improvement brought by pre-trained model is different for different platforms. For hi3519A-nnie12-int8, a small count of samples can get better improvement. For cpu-openppl-fp32 and atlas300-acl-fp16, moderate samples achieve better improvement. For gpu-T4-trt7.1-fp32, all numbers of samples bring obvious improvement. Figure 7(e) displays the average Acc(10%) on 9 platforms. Orange curves are above the blue one. It indicates that NNLP can transfer the model latency knowledge learned from other platforms to a new platform, and increase its accuracy limit.

Transfer learning for different task models. We intend to explore how the pre-trained latency predictor of classification models contributes to the latency predictor of detection models. Detection models such as RetinaNet [24] usually employ the classification model ResNet34 [12] as the backbone. However, the latency difference between detection and classification models is huge due to the difference of task specific part. As shown in Figure 8, NNLP can well transfer the latency knowledge learned from the classification task to the detection task. With our powerful pre-trained embedding, the data required to achieve satisfactory accuracy reduces from 1,000 to 50, contributing to a 20 $\times$ superior data efficiency, which makes the fast transfer with few samples possible.

8.7 Further Verification on NAS Tasks

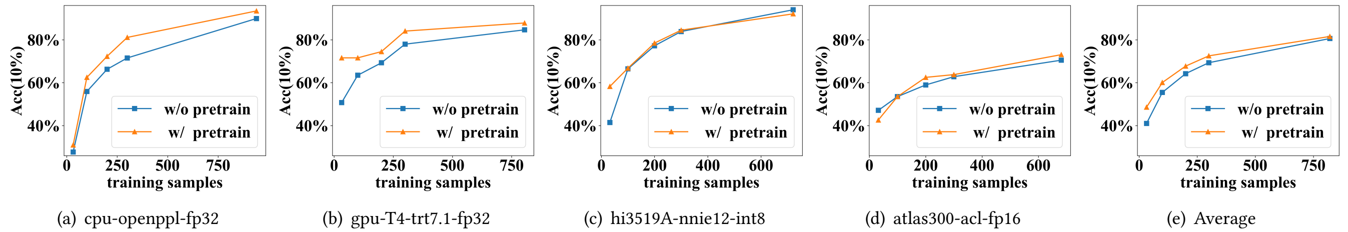
With relatively low time cost, we have a much higher possibility for finding models which meets the latency requirements in NAS tasks. Given the latency requirement of the model (e.g., 5ms), the improvement of the predictor accuracy can increase the probability of finding a model that meets the latency requirements. For example, with the accurate latency feedback, we are only able to test 1k models, but with the latency predictor, we can get the latency of 10k models. With the help of an accurate latency predictor, we guarantee that the prediction error is less than 10% for 95% of models. Therefore, we can get more models that meet the latency requirements, which will lead to an increased probability of finding a higher-precision model. With the help of an accurate latency predictor and accurate accuracy predictor, we are able to find models with higher task accuracy.

As shown in Figure 9, the pareto front chosen from 1,000 models which are sampled from OFA supernet [2] with different metric. As the latency range is relatively large, the Kendall Tau correlation coefficient between true latency and other three indicators FLOPs, Lookup Table and Predict is 0.87, 0.91 and 0.92. With given computation budget around 300M, it turns into 0.38, 0.53 and 0.73 respectively. Therefore, for models with a large latency range, only FLOPs pareto is far from the true latency pareto front. However, with a small latency range, the Lookup table method fails to find optimal pareto front models. With the same latency, the accuracy gain of the pareto front models from our accurate predictor is 1.2%



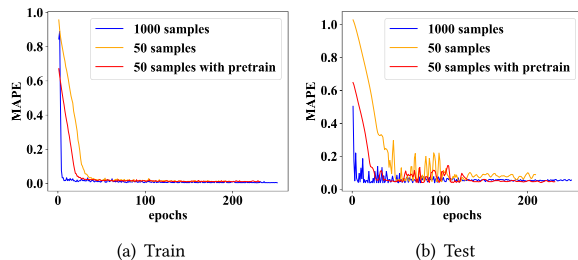
(a) ResNet (b) MobileNetV2 (c) MobileNetv3 (d) MnasNet (e) SqueezeNet

Figure 6: The experiment of transfer learning on unseen structures. We compare the prediction accuracy of NNLP with and without pre-trained model. The number of training samples is set to 32, 100, 200, 300, and etc. The training time costs corresponding to 32/100/200/300 samples are 59/97/150/195 seconds respectively.



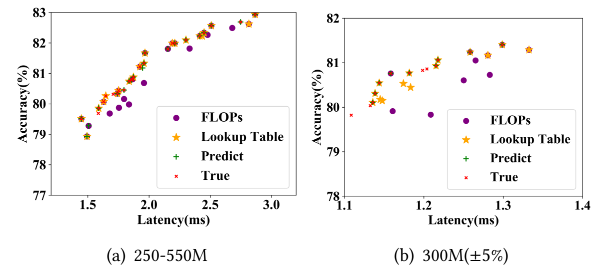
(a) cpu-openppl-fp32 (b) gpu-T4-trt7.1-fp32 (c) hi3519A-nnie12-int8 (d) atlas300-acl-fp16 (e) Average

Figure 7: The experiment of transfer learning on unseen platforms. We compare the prediction accuracy of NNLP with and without pre-trained models. The number of training samples is set to 32, 100, 200, 300, and etc.



(a) Train (b) Test

Figure 8: Transfer learning results from classification models to detection models. The test MAPE for 1,000 samples, 50 samples, 50 samples with pre-trained model is 0.038, 0.044, 0.040.



(a) 250-550M (b) 300M(±5%)

Figure 9: Comparison of different pareto front models from 1,000 models in the FLOPs (or lookup table latency, predicted latency, true latency)-Accuracy curve.

compared with FLOPs pareto and 0.6% compared with Lookup table pareto.

9 HOW DOES NNLQP HELP MODEL DESIGN?

The true and accurate latency feedback is essential for model design and deployment. Therefore, the developer can use our system to help with deep neural network design by alleviating the cost of NN latency query and prediction in following cases: (1) For models intend to be deployed on multiple platforms, NNLQ improves deployment efficiency to obtain true latency feedback. (2) With the true latency feedback, we can get a high-level decision in the initial stage of model design. (3) When the model design cannot meet the requirements, the intervention of NAS is required, so that the prediction function of NNLP can help with finding models with higher task accuracy. (4) If different models or tasks are required, the evolving latency database can further improve the efficiency of latency prediction.

NNLQ can free researchers from the tedious process of collecting the latency of DNN models from various hardware devices: In practice, researchers intend to deploy models on multiple devices such as Apple and Samsung and so on. These mobile phones also have different chips, such as Apple A, Qualcomm Snapdragon, MTK. Even for Qualcomm chips, there are four different types ARM/GPU/DSP/NPU. To get the accurate latency feedback, we need to deploy models on different platforms adapting to different hardware. For example, we can use TensorRT to get

Table 7: The total test models, time cost and speedup for different methods with latency measurement only or latency prediction with or without transfer.(k=1,000, m=1,000,000, T=once prediction cost, 1000T=once true latency test cost)

	measurement	prediction	test models	time cost	speedup
latency measurement	1k	0	1k	(1m + 0) × T	1×
latency prediction without transfer	1k	10k	10k	(1m+10k) × T	0.99×
latency prediction with transfer	50	10k	10k	(50k+10k) × T	16.7×

the latency feedback on NVIDIA GPU, while we cannot use TensorRT to transform model into hardware format for other hardware devices. Most hardware has different deploying format. Therefore, the proposed model latency query system NNLQ can perform automatic model deployment and latency measurement on multiple platforms. We use ONNX as a unified representation and integrated different toolkits from different vendors into one framework. So the deployment can be hardware-agonistic and free the users from deployment details. ONNX model and target platform are provided as query input, and NNLQ returns the true model latency on the target hardware through the three steps: model transformation, device acquisition and latency measurement.

Our neural network latency query system can provide some high-level decision in the model design:

- Which operators are not suitable: for example, hard swish is not supported on openppl and therefore should be avoided.
- On the choice of backbone to achieve better latency-accuracy trade-off: For example, RegNetX-200M [30] and ResNet18 [12] have similar ImageNet accuracy which is 68.7 and 70.3, but the latency of RegNetX-200M is 150% of ResNet18 on P4 int8. Therefore, we should choose ResNet18 compared with RegNetX-200M.
- In the choice of hardware for inference speedup: Given the same model-ResNet18 + data type int8 + batch size 1, the latency on P4 is 2 times of the latency on T4. If these two devices are available, changing deployed device from P4 to T4 can bring 50% speedup. Besides, atlas300 is faster than mlu270 under the same setting.
- In the choice of data type for possible accuracy degradation: for the vision transformer models, the speed up brought by int8 compared with FP32 is less than 5%. To avoid the potential accuracy degradation, you can choose FP32 directly.

In the network architecture search process, our neural network latency prediction system can reduce search cost and find models with higher task accuracy. If the current model is not able to be deployed on some required hardware, we need to redesign a model which is general across different platforms. In hardware-aware NAS, models are selected with the help of hardware feedback, therefore, if the hardware feedback takes a long time to be acquired, this could increase the search cost and hinder the use of hardware-aware NAS. NAS needs to test a large number of models, and true latency measurement is very slow. Our latency prediction is able to predict accurate model latency with 1,000 times improved efficiency as shown in Table 2.

Developers can further use the data from our evolving database to reduce the cost of latency predictor training. Since the prediction process brings a possible gap in the true latency and predicted latency, improving the performance of the latency prediction allows us to simulate the true latency feedback as accurately as possible. The comparison of time cost is listed in the Table **??**. If the training cost of the predictor is high, we may not achieve the purpose of improving efficiency, but if we use historical information with our evolving database, we can get the highly accurate latency predictor with less cost, while getting more model latency.

10 CONCLUSION

In this work, we investigate latency acquiring problems for two aspects: latency query and latency prediction. For latency query, we design a unified interface to measure latency on diverse platforms, and construct an evolving database to store models and their latencies. For latency prediction, benefiting from the large-scale latency knowledge, we design a unified graph embedding that can efficiently transfer for new network structures and platforms. Equipped with our NNLQP system, the efficiency of latency acquisition has been improved obviously.

ACKNOWLEDGMENTS

This work is partially supported by SenseTime Research Fund for Young Scholars and partially supported by National Natural Science Foundation of China (No. 62072018).

We sincerely thank Yanfei Wang, Jian Hu, Pan Huang, Peng Wang, and Qize Wu from SenseTime for their help in the building of the NNLQP system.

REFERENCES

[1] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[2] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-All: Train One Network and Specialize it for Efficient Deployment. arXiv:1908.09791 [cs.LG]

[3] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[6] TensorRT Documentation. 2021. Optimizing for Tensor Cores. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#optimize-tensor-cores.

[7] Łukasz Dudziak, Thomas Chau, Mohamed S Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D Lane. 2020. Brp-nas: Prediction-based nas using gcns. *arXiv preprint arXiv:2007.08668* (2020).

[8] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. 2019. Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks. In *The IEEE International Conference on Computer Vision (ICCV)*.

[9] Miguel Grinberg. 2018. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.".

- [10] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
- [11] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems.* 1025–1035.
- [12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]
- [13] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision.* 1314–1324.
- [14] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
- [15] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. 2019. Learned TPU cost model for XLA tensor programs. In *Proc. Workshop ML Syst. NeurIPS.* 1–6.
- [16] Samuel J Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2020. A Learned Performance Model for Tensor Processing Units. *arXiv preprint arXiv:2008.01040* (2020).
- [17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).
- [19] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Lin. 2021. Hw-nas-bench: Hardware-aware neural architecture search benchmark. *arXiv preprint arXiv:2103.10584* (2021).
- [20] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. 2021. BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction. In *International Conference on Learning Representations.* https://openreview.net/forum?id=POWv6hDd9XH
- [21] Yuhang Li, Mingzhu Shen, Jian Ma, Yan Ren, Mingxin Zhao, Qi Zhang, Ruihao Gong, Fengwei Yu, and Junjie Yan. 2021. MQBench: Towards Reproducible and Deployable Model Quantization Benchmark. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1).* https://openreview.net/forum?id=TUplOmF8DsM
- [22] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing.. In *Hot Chips Symposium.* 1–44.
- [23] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.
- [24] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV]
- [25] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW).* IEEE, 522–531.
- [26] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63. https://doi.org/10.1109/MM.2021.3058217
- [27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.
- [28] Evgeny Ponomarev, Sergey Matveev, and Ivan Oseledets. 2020. LETI: Latency Estimation Tool and Investigation of Neural Networks inference on Mobile GPU. *arXiv preprint arXiv:2010.02871* (2020).
- [29] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary neural networks: A survey. *Pattern Recognition* (2020), 107281. https://doi.org/10.1016/j.patcog.2020.107281
- [30] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 10428–10436.
- [31] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2016. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv:1506.01497 [cs.CV]
- [32] Jaehun Ryu and Hyojin Sung. 2021. MetaTune: Meta-Learning Based Cost Model for Fast and Efficient Auto-tuning Frameworks. arXiv:2102.04199 [cs.LG]
- [33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 4510–4520.
- [34] Mingzhu Shen, Kai Han, Chunjing Xu, and Yunhe Wang. 2019. Searching for accurate binary neural architectures. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops.* 0–0.
- [35] Mingzhu Shen, Feng Liang, Ruihao Gong, Yuhang Li, Chuming Li, Chen Lin, Fengwei Yu, Junjie Yan, and Wanli Ouyang. 2021. Once Quantization-Aware Training: High Performance Extremely Low-bit Architecture Search. arXiv:2010.04354 [cs.CV]
- [36] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 1–9.
- [38] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2820–2828.
- [39] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning.* PMLR, 6105–6114.
- [40] Xiuying Wei, Ruihao Gong, Yuhang Li, Xianglong Liu, and Fengwei Yu. 2022. QDrop: Randomly Dropping Quantization for Extremely Low-bit Post-Training Quantization. In *International Conference on Learning Representations.* https://openreview.net/forum?id=ySQH0oDyp7
- [41] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. 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.* 81–93.
- [42] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E. Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. TenSet: A Large-scale Program Performance Dataset for Learned Tensor Compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1).* https://openreview.net/forum?id=aIfp8kLuvc9
- [43] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards Unified INT8 Training for Convolutional Neural Network. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

A KERNEL ADDTIVITY VALIDATION

To validate whether the additivity assumption is reliable, we conducted latency tests on a GPU platform. We test 60 DNN models containing 6 types: ResNet, AlexNet, NasBench201, EfficientNet, MobileNetV2, and MobileNetV3. The target latency platform is the GTX1660 GPU hardware and the TensorRT software. We collect the operator fusion rules and split each model into fine-grained kernels. Then we query both model and kernel latencies by NNLQ. We compare the model latency and the sum of its kernel latencies, and the scatter plot is shown as Figure 2.

In Figure 2, the x-axis denotes the latency of the neural network, the y-axis denotes the sum of latencies of kernels contained in the model, the red line denotes function $y = x$, and the points with different colors mark the models with different model types. From the figure, we can see: (1) The points of different colors are all above the line $y = x$. It indicates that on this hardware, the sum of the kernel latencies is greater than model latency for all tested graph structures. The phenomenon shows that the additivity assumption is not reliable for the GXT1660 GPU + TensorRT platform. The latency overlap between neighborhood kernels may cause unreliability; (2) For points with the same color, the relationship between model latency and the sum of kernel latencies is linear approximately. The greater the model latency, the greater the sum of related kernel latencies. However, different model types show different linear slopes, and the relationship between the model latency and the sum of kernel latencies will vary as the network structure changes. Therefore, the additivity assumption of the specific platform is unreliable, and for various model types, it is difficult to correct the sum of kernel latencies to the model latency.

B DETAILS OF TARGET PLATFORMS

The experiments in this work involve the following platforms: (1) cpu-openppl-fp32: Intel CPU Xeon Gold 6246 with openppl [1] inference library; (2) hi3559A-nnie11-int8/hi3519A-nnie12-int8: Hi3559A or Hi3519 board with hisvp nnie library; (3) gpu-T4-trt7.1-int8/gpu-T4-trt7.1-fp32/gpu-P4-trt7.1-int8/gpu-P4-trt7.1-fp32: Nvidia GPU T4 or P4 with TensorRT7.1 library. TensorRT[2] is a high-performance inference library developed by NVIDIA; (4) atlas300-acl-fp16: Atlas300 with acl ascend310 [3] library. The acl is a neural network inference software developed by HUAWEI for the hardware named Atlas; (5) mul270-neuware-int8: MUL270 with neuware [4] library, which is developed by Cambricon. The fp32, fp16, and int8 denote the data type of tensor operation when running the DNN model. (6) rv1109-rknn-int16/int8: rv1109 is an ASIC hardware of Rockchip. Its runtime library is rknn [5].

C METRICS

MAPE is a non-negative number and the smaller value represents the better prediction. The MAPE is defined as:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y_i'|}{y_i} \times 100\% \tag{6}$$

[1]https://github.com/openppl-public/ppl.nn
[2]https://github.com/NVIDIA/TensorRT
[3]https://support.huawei.com/enterprise/zh/doc/EDOC1100206822/12030271
[4]https://developer.cambricon.com/index/document/details/classid/3/cid/1/id/41.html
[5]https://github.com/rockchip-linux/rknn-toolkit

Table 8: The statistics of kernels split from 20,000 models.

Kernel Family	Number	Percentage
AveragePool	1164	0.32%
Concat	32617	8.91%
Conv+Add+Relu	15672	4.28%
Conv+Add	15497	4.23%
Conv+Clip	7340	2.01%
Conv+Relu	219123	59.88%
Conv	38013	10.39%
Flatten	3318	0.91%
Gemm	12498	3.42%
GlobalAveragePool	4526	1.24%
MaxPool	8887	2.43%
ReduceMean	1290	0.35%
Relu	4605	1.26%
Sigmoid+Mul	1406	0.38%
All	365956	100.00%

Acc(δ) is between 0% and 100%, and the larger value reflects the better prediction. The Acc(δ) is defined as:

$$Acc(\delta) = \frac{1}{n}\sum_{i=1}^{n} pos(\delta - \frac{|y_i - y_i'|}{y_i}) \times 100\% \tag{7}$$

where $pos(z)$ is 1 when $z \leq 0$, and 0 otherwise.

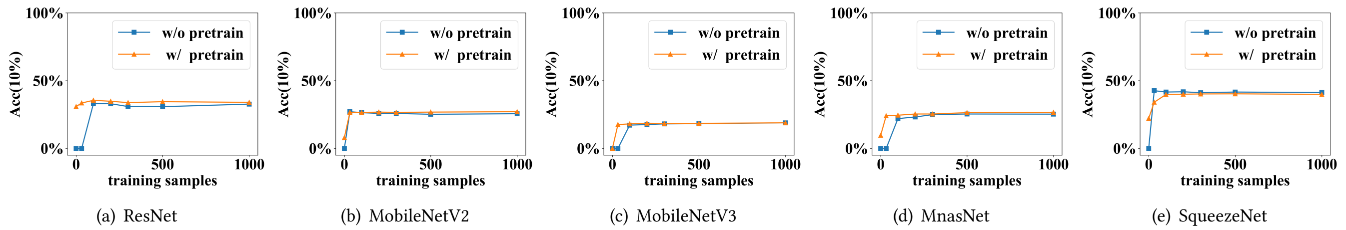
D THE STATISTICS OF SPLIT KERNELS

Table 8 displays the statistics of kernels that are split from 20,000 models. There are 365,956 kernels in total. On average, each model is split into about 18 kernels based on our fusion rules. The Conv+Relu family has the most kernels, accounting for 59.88% of the total. It shows that for DNN models, the pattern Conv+Relu is one of the most widely used structures.

E IMPLEMENTATION DETAILS IN COMPARION WITH RELATED WORKS

We compared our prediction method with related works: **(1) FLOPs/FLOPs+MAC:** we directly use the FLOPs feature or FLOPs+MAC features to predict latency by linear regression method; **(2) nn-Meter:** we first adopt the random forest regression to predict the latency of all kinds of kernels, and then sum kernel latencies as the model latency. The extracted kernel feature and regression models are defined the same as the nn-Meter official project[6]. Due to the unreliability of the additivity assumption, we apply the linear regression method to correct the summation result; **(3) TPU:** We first use GraphSAGE to predict the latency of kernels. The same as nn-Meter, we correct the sum of kernel latencies by the linear regression method, and obtain the model latency; **(4) BRP-NAS:** The original BRP-NAS is only proper to the model composed of specific blocks. It can not be applied to our latency dataset directly. We take the node features and network topology defined in NNLP as inputs. We use the official GNN backbone of BRP-NAS[7] to predict latency.

[6]https://github.com/microsoft/nn-Meter
[7]https://github.com/SamsungLabs/eagle



(a) ResNet (b) MobileNetV2 (c) MobileNetV3 (d) MnasNet (e) SqueezeNet

Figure 10: The experiment of transfer learning on unseen structures for the FLOPs+MAC.

F TRANSFER LEARNING FOR FLOPS+MAC

For comparison, we also performed transfer learning with the prior method FLOPs+MAC. The experiment setup is the same as NNLP. The transfer learning results on unseen structures are shown as Figure 10. Orange curves and the blue ones are almost overlapped, which shows that the transfer learning of FLOPs+MAC can not bring improvement for latency prediction compared with learning from scratch. In addition, regardless of the number of training samples, the prediction accuracy of the FLOPs+MAC is always poor (Acc(10%)<50%). The FLOPs+MAC model contains only one linear layer, which does not has a backbone that can be shared. So we can not predict latency on multiple platforms simultaneously. Therefore, we can not conduct transfer learning on unseen platforms for FLOPs+MAC either.