BOOM: Use your Desktop to Accurately Predict the Performance of Large Deep Neural Networks

Qidong Su
qdsu@cs.toronto.edu
University of Toronto &
Vector Institute & CentML
Toronto, Canada

Jiacheng Yang
jiacheng.yang@mail.utoronto.ca
University of Toronto &
Vector Institute
Toronto, Canada

Gennady Pekhimenko
pekhimenko@cs.toronto.edu
University of Toronto &
Vector Institute & CentML
Toronto, Canada

ABSTRACT

The intensive computational requirements of training deep neural networks (DNNs) have significantly driven the adoption of DNN accelerators like Graph Processing Units (GPU). However, selecting the most suitable GPU from all candidates with drastically different specifications and prices is still a challenging problem. While directly measuring the performance of DNN training tasks on every candidate is prohibitive, and not always available due to hardware shortage, an accurate performance predictor can assist in the decision-making. However, most existing performance predictors cannot predict the GPU memory footprint in an accurate, generalizable, and interpretable manner, which is crucial to the feasibility and performance of running the DNN model on real GPUs. Moreover, many optimizations for DNN training, such as mixed precision training and checkpointing, can significantly impact performance. However, such *hardware-dependent* optimizations are not considered by existing performance predictors.

In this work, we propose a novel performance predictor containing (1) a memory footprint predictor with better generalizability and interoperability; (2) a runtime predictor supporting hardware-dependent optimizations. Experiments show that our memory footprint predictor achieves an average error of 2.7% on CNN models and 0.9% on transformers, and the runtime predictor achieves an average error of 10.5% on the CNN and transformer models.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

performance profiling, deep neural networks, machine learning

1 INTRODUCTION

Recent advances of deep learning (DL) in various fields such as computer vision [25, 31, 74], natural language processing [14, 23, 75, 76], recommendation systems [83], and game playing [51, 68] have spurred the development of accelerators for deep neural networks (DNN) (e.g., GPUs [43], TPUs [40], and NPUs [2, 3, 8, 39, 47–49, 54]).

Among them, GPUs are one of the most popular families of accelerators. To support different budgets and scenarios such as training and inference, the hardware vendors and cloud service providers offer a wide spectrum of GPUs with different prices and specifications, including computation capability and device memory size. For example, public cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) have different pricing strategies [1, 57] for GPUs with different computing capabilities. While it manifests a more accessible spectrum of GPUs to users, it is also critical for users to choose the most suitable GPU depending on their time or budget constraints.

One way to achieve this target is to manually deploy the DNN model on different GPUs to analyze the feasibility and the performance of the DNN workload. However, this method is not always feasible due to the limited availability of hardware resources [55]. It requires access to every possible GPU in advance, but the desired devices are not always available due to the limited quota. To make things worse, this method is often laborious and not cost-efficient. If a DNN memory and performance predictor was available, choosing the most suitable GPU could be done automatically without deploying and benchmarking the DNN model on actual GPUs, and users only need to pick up a candidate from the predicted price-performance Pareto optimal boundary.

To satisfy the needs, many existing works propose performance prediction techniques [28, 30, 52, 64, 87]. However, they are still not satisfactory due to two main limitations. First, many works merely focus on performance prediction but ignore the prediction of memory usage, while existing memory usage predictors suffer from heavy human intervention [29] or insufficient generalizability [28]; Second, existing performance predictors are unaware of optimizations that depend on hardware specifications, which limits the applicability of DNN performance predictors and hence leads to inaccurate and ineffective predictions.

Memory usage is a decisive factor in the performance of DNN applications, which can affect choosing the optimal GPU model. It decides the upper bound of the model size and the maximal batch size, which influences the throughput significantly, especially for a tight memory budget and small batch sizes, as shown in Figure 2. Without an accurate memory predictor, the performance prediction could be invalidated as the execution of the DNN may not even be feasible on a particular GPU. The recent breakthrough of large models such as GPT [62] has further deteriorated the feasibility of existing DNN predictors since their increasing model size implies a more stringent requirement of on-device memory [58, 65]. Furthermore, memory is also an important indicator for decision-making in optimizations since many DNN optimizations are trading between memory and runtime, which, in turn, results in inaccurate performance predictions. For example, gradient checkpointing [11, 17, 37, 45, 84] discards some intermediate activation values during the forward pass. It can significantly reduce the memory footprint of DNN training, which enables more economic GPU models, as shown in Table 1. If the performance predictor does not

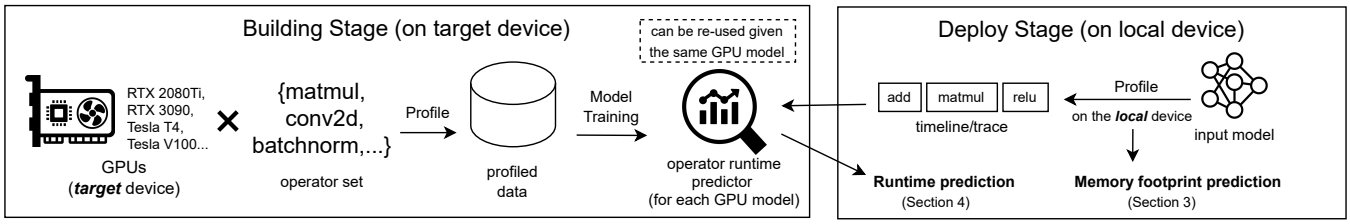


Figure 1: Overview of BOOM. The whole workflow of BOOM consists of two stages: (1) Building stage (discussed in Section 4), where it builds operator runtime predictor for each target device (GPU model) based on the collected runtime information. In BOOM, we use an NN-based predictor for complicated operators like convolutions, while using linear projection for simple operators like element-wise operators. The procedure of data collecting and prediction model training is carried out *only once* for each GPU model. (2) Deploy stage, where the predictor built in the previous stage predicts the performance of the input model, hyper-parameters, and workloads. In this stage, the predictor can be reused if the GPU model is the same. BOOM runs the model on the local device and collect traces and runtime information. With our proposed *fake memory allocator* (discussed in Section 3), we can run models exceeding the local device's memory constraint, which is a bottleneck for prior works like Habitat and Skyline. Memory footprint is also done during this procedure. BOOM then calls the operator runtime predictor with collected traces to produce runtime prediction.

Table 1: How memory capacity and hardware-dependent optimizations affect the choice of the optimal GPU models. Gradient checkpointing (*ckpt* in the table) enables training a GPT-like language model on much cheaper (4× cheaper) commodity GPUs (e.g. RTX 3090 or 4090) by reducing memory footprint. Without it, users have to use expensive high-end GPUs such as A100. The price data is from `vast.ai` [9], a GPU rental service.

	Memory	Throughput	Throughput/$
A100	40 GiB	100%	100%
RTX 3090	24 GiB	OOM	OOM
RTX 3090 + **ckpt**	24 GiB	60%	**380%**
RTX 4090	24 GiB	OOM	OOM
RTX 4090 + **ckpt**	24 GiB	80%	**260%**

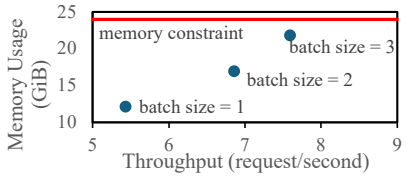


Figure 2: Memory capacity determines the maximum batch size, which significantly affects the final throughput. Measured on running $GPT2_{Large}$ on an RTX 3090. If we overestimate memory usage by 10%, we will wrongly predict that it will run out of memory, and thus we have to fall back using batch size=2, which causes 10% throughput drop.

consider memory usage, these options will be neglected, causing potentially unnecessary overspending.

Existing DNN memory prediction techniques [28, 29] try to address these challenges by directly analyzing the computational graphs. However, they need either heavy engineering effort and expert knowledge, or building an end-to-end ML pipeline, making it hard to maintain and adapt to new operators and optimizations. Another method is profile-then-extrapolate, which runs the model on devices we already have in hand (*local device*, e.g. a commodity GPU and extrapolate [80] based on the profiling result. Their implementation is much simpler, but they are limited by the memory constraint of the local devices.

In this work, we propose a new cross-device DNN memory and performance predictor called BOOM (**B**efore **O**ut-**O**f-**M**emory), of which the overview is shown in Figure 1. For memory predictions, BOOM leverages two insights that 1) *we can measure memory footprint by tracing calls to the device memory allocator* and 2) *the correctness of computational result is not necessary for accurate memory footprint measurement* since the control flows of DNN workloads usually do not depend on the data in the device memory. BOOM can hence allow the device memory allocator to reuse allocated memory, which enables allocation exceeding the memory constraint of the local device. As a result, BOOM only needs to measure the memory footprint to produce accurate DNN memory usage predictions.

For more accurate performance predictions, we observe that the vanilla MLPs used in existing ML-based predictors fail to capture the highly discontinuous patterns of GPU performance curves. To handle this problem, we propose a data augmentation technique, which significantly improves its generalization ability to handle the discontinuous pattern of GPU performance curves. Compared with previous approaches, BOOM produces more accurate DNN performance predictions on the target accelerator with the consideration of hardware-dependent optimizations, which have not been well-studied to our best knowledge.

Our main contributions can be summarized as follows:

- We are the first to identify the problem of existing approaches on accurate memory and performance prediction of DNNs that involve hardware-dependent optimizations;
- We propose a novel memory prediction method by overriding the memory allocator, which is more generalizable, explainable, and extensible to new operators and optimizations, with the least

human interference. It achieves an average error of 2.7% on CNN models, 0.9% on transformers, and 3.5% on NAS models.
- We improve the accuracy of runtime prediction by data augmentation and extend it to support more optimizations. It achieves an average error of 10.5%.

2 BACKGROUND

2.1 DNN Optimizations

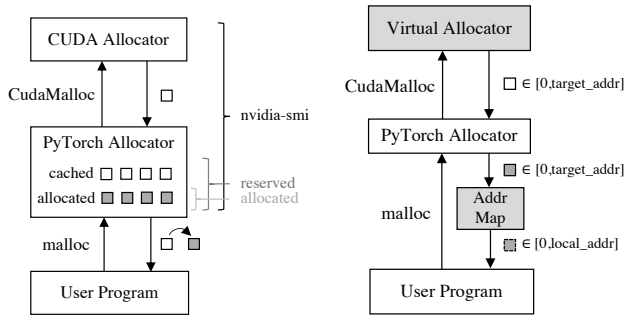
DNN training has been notorious for being costly in time, money, and energy [56, 72, 86], and the techniques to improve the efficiency of DNNs training are highly demanded. There are many works targeted at optimizing DNNs. Table 2 lists some optimizations for DNN training. Among these, many are optional and not enabled in popular DL frameworks by default because they are trading off among several metrics. While they have the potential to improve one metric, they also run the danger of harming others. Users need to decide whether to turn them on, considering the workload and hardware. For example, checkpointing discards some intermediate results during forward computation and re-computes them when needed, trading computational redundancy for a lower memory footprint. It is helpful when we train a large model on a device with a tight memory constraint. We call such optimizations *hardware-dependent optimizations*.

2.2 Framework-Level Memory Management

Instead of directly using CUDA's memory management APIs like `cudaMalloc` and `cudaFree`, DL frameworks usually have their own device memory management between CUDA and the user programs [29] to leverage more DL-specific optimizations. For example, PyTorch uses its own caching allocator [7, 60]. When the user program requests to release a tensor, PyTorch will cache its memory space for potential future reuse instead of directly calling `cudaFree`, which might cause slow down. PyTorch's allocator also has parameters specifically tuned for DL to avoid fragmentation, such as 512 bytes as the minimal unit of allocation. Figure 3a shows how the PyTorch allocator bridges user programs and lower-level libraries.

2.3 Profiling Tools of DNNs

The foundation of performance prediction is to accurately measure the duration of each component of a DNN model, where profiling tools play an essential role. DL frameworks usually provide their own profiling utilities. Compared with hardware-level profilers (e.g. Nsight Systems [6] and Nsight Compute [5]), they have a higher-level interface with richer semantic information specific to the DNN workloads. For example, PyTorch has a built-in profiling tool [60] producing a clear timeline structure, which can be seen as the result of *serialization* of the computational graph. It provides extra DNN-specific information like the call stack from the top-level module to basic operators and links between GPU kernels and their CPU callers, which significantly simplifies further analysis. Our runtime predictor is built on top of PyTorch's profiler.



(a) PyTorch Memory Allocator **(b) Fake Memory Allocator**

Figure 3: Two-layer allocator: (a) shows how PyTorch's allocator acts as an intermediate role between the user programs and the underlying CUDA allocator. It also shows the ranges of memory included by different memory usage measurement APIs; (b) shows how we enable the allocator to 'allocate' more space than the real device memory. The grey blocks are modified components. We replace the CUDA allocator with a virtual allocator running on a larger *target address space* and use an address mapping component to map it back to the *local address space* which is accessible to user programs.

3 MEMORY USAGE PREDICTOR

In this section, we analyze the challenges of DNN memory prediction and then propose a novel memory footprint prediction technique that allows the allocator to "allocate" more space than the local device memory limit, thereby reducing the memory prediction problem into memory measurement.

3.1 Challenges

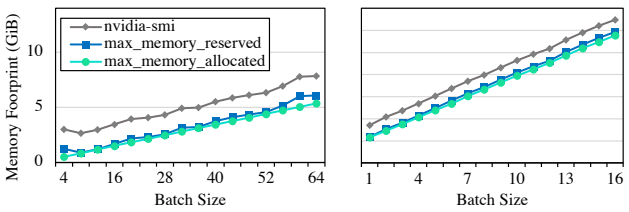
The difficulties of accurate DNN memory prediction include:

Distribution of memory usage across the software stack. Memory usage is distributed across different layers of the software stack including drivers, libraries, frameworks, and user programs, which leads to different results when profiling the memory usage at different layers. For example, in PyTorch, `max_memory_allocated` returns the memory used by all tensors, while `max_memory_reserved` returns all memory managed by PyTorch caching allocator [7], including freed but cached memory areas and fragmentation. `nvidia-smi` is a tool provided by NVIDIA to monitor the status of GPUs, including temperature, clock, and memory. Its total memory usage numbers are the most accurate and inclusive. Figure 4 visualizes the non-negligible difference (at gigabyte granularity) of measured memory footprints via different APIs. Hence, an accurate memory predictor must align with the most inclusive API, i.e., `nvidia-smi`.

Sensitivity of memory footprint to details in user code. The final memory footprint is sensitive to many tiny options and details in the code, and it is hard to model all of them manually. For example, enabling the `set_to_zero` option when zeroing the gradient tensor at the beginning of each training iteration might lead to different behaviors of the memory allocator. Moreover, the algorithm selection of cuDNN [19] can affect memory usage, which has several options exposed to the users. However, existing memory

Table 2: Optimizations for DNN. 👍 means improvements such as reducing computational redundancy or memory usage, and 👎 means deterioration such as more redundancy, more memory usage, or lower accuracy. "?" means the effects depend on specific techniques. "=" means remaining unchanged.

Optimization	Examples	Redundancy	Memory	Accuracy
Gradient Checkpointing	Checkmate [37], DTR [45]	👎	👍	=
Offloading	vDNN [67], SwapAdvisor [33]	👎	👍	=
Graph Transformations	TASO [38], PET [77]	👍	?	=
Compression	JPEG-ACT [26], Gist[36]	?	👍	?
Low Precision	QNN [35], AMP [22]	👍	👍	👎
Sparsification	Sparse Transformers [13, 15]	👍	👍	=
High-performance Kernels	TVM [16], CUTLASS [44]	👍	?	=
Vertical Fusion	DNNFusion [59]	👍	👍	=
Horizontal Fusion	HTFA[78], Retiarii[82]	👍	👎	=



(a) ResNet50 **(b) $BERT_{BASE}$**

Figure 4: Memory footprint measured by different APIs

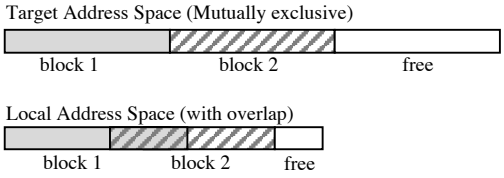


Figure 5: How we map the larger target address space to the smaller local address space by allowing overlapping between memory blocks, done in address mapping in Figure 3b.

predictors suffer from either prohibitive engineering efforts of manually encoding rules or the lack of generalizability and extensibility. We will discuss it in detail in Section 7.

3.2 Key Idea: Fake Memory Allocator

The key idea of our memory prediction technique is to actually "run" the DNN model and measure the memory footprint, which naturally solves the two challenges mentioned above. However, it requires that the *local device* we have in hand has sufficient memory, which cannot be satisfied with large input models. To make things worse, running large models is also time-consuming. To address these two problems, we adopt two techniques: 1) using a fake memory allocator to loosen the memory constraint of the local device; 2) skipping unnecessary tensor computation. They are based on the following two observations:

Observation 1: Large blocks of memory, including all tensors, are managed by the framework memory allocator. By tracking the calls to the memory allocator, we can record all memory allocation and free requests, hence the memory usage throughout the execution.

Observation 2: The control flow and memory access pattern of most DNN models do not depend on exact tensor values. Specifically, if we view DNN models in the graph-kernel hierarchy, most models have no graph-level control flow. For example, sequential models like VGG [69] are always executed sequentially. While some other models [31, 73] have branches in the computational graph, both branches need to be fully executed regardless of the inputs. At the kernel level, the order of all memory accessing instructions of most kernels is exactly identical if the shape of inputs is fixed, regardless of the exact values stored in the tensor.

Based on these two observations, our first idea is to replace the CUDA memory allocator with a *fake allocator* that works on an imaginary address space that has the same size as the target device memory (*target memory space*), as shown in Figure 3b. It has the same APIs as the original CUDA allocator, so the caching mechanism of PyTorch still works properly with it.

However, since the address space is imaginary, we must map it to an accessible memory region on the local device in order to prevent user programs from invalid memory accesses. This address mapping process is done before the address is returned to downstream programs, as shown in Figure 3b. Specifically, our solution to make mapping a larger memory space onto a smaller region possible is to lift the constraint of *mutual exclusivity*. As shown in Figure 5, memory blocks in the target memory space are mutually exclusive, but they are allowed to overlap with each other after address mapping.

As a result of lifting mutual exclusivity, we lose the guarantee of the correctness of numeric computations. Nevertheless, this is acceptable as we are only concerned with precise memory consumption measurement. Observations 1 and 2 lead to a corollary that the exact tensor data will not affect when and where the memory allocation and free will be called. Thus, if we manipulate the memory allocator and return a wrong but accessible address, the kernels should still work normally. While the results will be numerically

incorrect, given Observation 2, the calls to the memory management functions will not be affected, and hence we can book-keep the memory usage correctly.

Furthermore, since the computation results are not important for memory usage measurement, we can skip the computation of an operator whenever it does not affect memory measurement. It also enables our memory predictor to support operators with constraints on their inputs (e.g. logarithm, which requires the input to be positive) since the numerical operations will not be triggered. We skip the CUDA computation by a) override `cudaKernelLaunch`, b) remove the calls to libraries such as cuBlas and cuDNN.

There are still exceptions where we cannot skip the computation, such as the operators involving algorithm selections of cuDNN (e.g. convolution) since different algorithms can lead to different memory footprints. Hence, the fake allocator still guarantees a range of accessible memory for them.

3.3 Implementation

In this section, we discuss the implementation details of the memory consumption predictor. Shown as the two grey blocks in Figure 3b, it consists of two parts: 1) a virtual allocator running on the address space of the target device; 2) an address mapping component translating addresses from the target device address space to that of the local device. We call the combination of them a *fake allocator*.

Two-level Fake Allocator. We override the CUDA allocator with a purely virtual allocator working on the target address space. It simulates the behavior of a normal allocator on the target device with a potentially larger address space. The addresses it returns always satisfy mutual exclusivity. We call them *target addresses*.

For the framework allocator, to map a target address to an accessible local address, it will first allocate a large memory pool (we call it the *scratchpad*) when being launched, onto which the target address will be projected. We organize the scratchpad as a ring. We maintain a pointer starting from the address 0 (w.r.t. the scratchpad). Each time there is an allocation request, we return the address where the pointer is as the local address, increase the local address and its corresponding target address to the address map, and move forward the pointer as many bytes as the requested size. If the pointer exceeds the end of the scratchpad, we take the remainder of the pointer divided by the scratchpad size as the new pointer, which means it restarts from the head. If the pointer is too close to the end of the scratchpad that the remaining space is less than the requested size, we need to move the pointer back to ensure the requested size of bytes after it is accessible. Finally, if the allocator finds that the address has already been occupied by an entry in the map, it will look backward to find an empty slot.

The dependency of these two levels is not unidirectional. When the framework allocator calls `cudaFree`, we need to find the target address which the requested local address was translated into. So we need another map from the local address to its corresponding target address.

Infeasibility of a single-level fake allocator. Another way to manipulate the allocator is to directly override `cudaMalloc` and `cudaFree`. This solution is simple and appealing as it is non-intrusive — we do not need to modify even a single line of the framework.

However, this solution is infeasible for a fundamental reason – the conflicts between the framework allocator and the lower-level allocator. The correctness of the behavior of framework-level allocators relies on the mutual exclusivity of the lower-level allocator, which the fake allocator has broken. A counterexample is shown in Figure 6, where PyTorch's caching allocator splits a freed but cached memory block and returns the split point (marked with the red triangle) as the response to an allocation request with a smaller size. However, the lower-level allocator is unaware of this allocation. Without mutual exclusivity, our fake allocator might still return the split point for another allocation request even if it is already used, causing a collision.

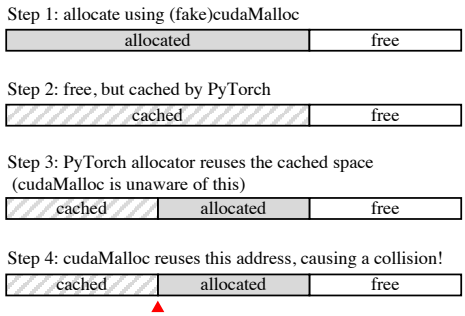


Figure 6: An example of collisions of the framework allocator and the lower level allocator. As they are decoupled, the framework allocator cannot signal to the lower-level allocator that an address is used, causing collisions.

Memory footprint beyond `cudaMalloc`. The memory usage measurement given by the fake allocator is supposed to be equal to the total amount of memory managed by the framework (which can be queried using `torch.cuda.max_memory_reserved` in PyTorch). But there is still a gap between it and the number given by `nvidia-smi`. The gap is mainly due to the memory consumed by the drivers, libraries, and CUDA contexts [29]. The gap measured by three different APIs is shown in Figure 4. We find that the gap is a constant depending on what libraries the task involves. For instance, vision tasks requiring cuDNN libraries usually require more memory due to the extra dependencies. Fortunately, the gap is a constant which can be measured by running a small matrix multiplication and a small convolution.

4 RUNTIME PREDICTOR

In this section, we discuss the challenges of runtime prediction of deep learning. Then we will propose a new runtime prediction technique to address these challenges.

4.1 Challenges

Hardware-dependent optimizations. As mentioned in Section 2.1, there are many optional *hardware-dependent* optimizations left for DL developers or auto-tuning policies to decide whether to turn them on. These optimizations could influence the runtime dramatically. However, to the best of our knowledge, existing works [30, 52, 53, 64, 71] do not take those optimizations into consideration,

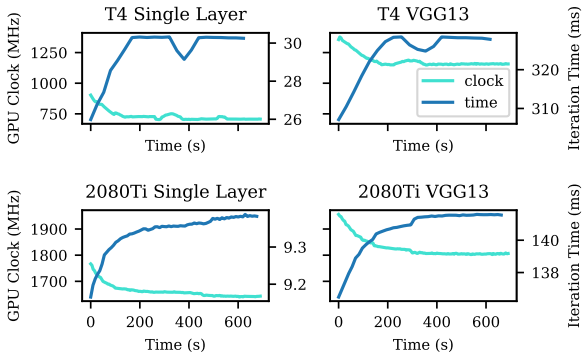


Figure 7: How GPU frequency and the running time of a single convolution layer and a whole model change as time goes by. Note that we use the same Y-axis scale for GPU frequencies for different workloads on the same device, showing different clocks they are running under.

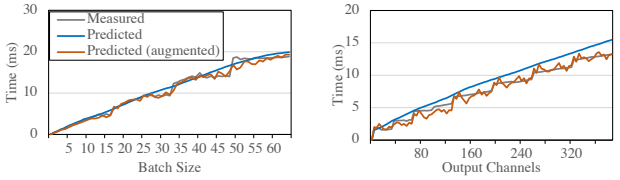


Figure 8: Discontinuous patterns of performance curves (measured on RTX2080Ti). The batch size is 32 in the right plot.

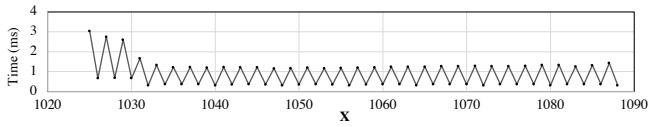


Figure 9: Discontinuous patterns are more obvious with Tensor Cores enabled. Measured running time of batched matrix multiplication with input shapes [7, 128, 1024] and [7, 1024, X] on Tesla T4. The X-Axis is X.

severely restricting their usefulness in real-life optimization scenarios. While Daydream [87] supports predicting the performance of DNNs after optimization, it only runs on the local device, making it unsuitable for cross-device performance prediction.

Consistency of GPU frequencies. Many existing performance prediction techniques adopt the assumptions [30, 37, 38] that 1) the total runtime of the DNN model is approximately equal to the sum of the runtime of each operator, and 2) the runtime of each operator can be measured independently, i.e. the duration of an operator as a part of a DNN model should be the same as that if we measure it separately. However, these two assumptions hold only if the GPU clock is consistent between individual operator measurements and the whole model execution. We have observed the measurement results can be time-varying and dependent on the specific workloads. We find that there are at least two scenarios during the measurement worth noticing (see Figure 7):

(1) In a long-term task, we have observed a decrease in GPU frequency over time. Consequently, even when repeating the same task, the measured time gradually increases before reaching a stable value. It often requires several hundred warm-up iterations to achieve a steady GPU frequency. The difference between the measurements of the first several iterations and the stable value can be up to more than 10%, as demonstrated in Figure 7.
(2) GPUs might use different frequencies for different workloads. In our case, it is possible for a single-operator and a whole-model measurement to run under different clock frequencies due to their different computational intensities. As a result, different GPU clock frequencies are observed. The difference between the GPU clock speeds during two different tasks can be up to 25% as shown in Figure 7.

In comparison, Habitat does not account for the consistency of GPU frequencies, so its predictions must be calibrated using both the predicted and measured runtime on the local device. This process involves training a prediction model specifically for the local device and running the model on the device itself, which takes more time and is limited by the local device's memory capacity.

Discontinuous patterns of GPU performance curves. We observe that when we change one dimension of the input shape, the runtime does not necessarily scale continuously, but shows discontinuous patterns like step functions, as shown in Figure 8. Directly using vanilla MLPs as in existing works [30, 41] to fit the performance curve will incur large errors as it is too smooth, especially at discontinuities. The situation even deteriorates if we enable Tensor Cores because libraries like cuBLAS [4] and cuDNN [19] will only leverage Tensor Cores when the input shapes satisfy some constraints (e.g., multiples of 4 or 8). So the performance curves are highly discontinuous, as shown in Figure 9.

One argument is that although vanilla MLPs cannot fit the discontinuous patterns perfectly, they still capture the general trend of performance curves and the prediction error of operators' runtime might offset with each other. However, the conjecture is not true, due to two reasons:

(1) Certain hyperparameters remain constant across all layers, such as the batch size. If the predictor consistently overestimates the target values for a particular batch size, the final outcome will be an accumulated overestimation.
(2) The discontinuous points in performance curves are usually powers of two or multiples of them (e.g., 32, 64, 96). Unfortunately, these numbers are also the most common parameters used by DL models.

Therefore, the predictor should be able to capture the discontinuous patterns.

4.2 Our Design

Since neural networks can be represented by computational graphs consisting of operators, runtime prediction can be done by predicting each operator's runtime and subsequently aggregating them. One key difference between memory and runtime prediction is that

the runtime is additive while the memory footprint is not. The runtime of the whole model can be decomposed into operators, while the memory usage changes with time as the program allocates and releases memory. We use machine learning only in runtime prediction because predicting the runtime of individual operators is much simpler than building a predictor that takes whole models as inputs.

Cost analysis of building ML-based predictor. ML-based prediction may introduce certain overheads, which mainly come from two aspects: 1) data collection; 2) model training. Since we use a very light-weight NN-based model for our predictor that can be trained in a short time, the major overhead comes from data collection. For BOOM's runtime predictor, we collect data on the *target hardware*, which is performed *only once* for each GPU model. Once the predictor is trained, users can leverage BOOM to obtain predictions for DNN inference and training jobs across various neural networks and hyper-parameters. This predictor can also be reused by other users who have the same GPU model. Although the data collection process can take several hours, this overhead is relatively minor compared to the typically lengthy upgrade cycles of GPU models. This step can be carried out by the vendor, cloud-service providers, or volunteers who can access the target device, as the target device is treated as a black box. The whole pipeline of the ML-based predictor is shown in Figure 1. More discussion about the cost of building datasets and the required amount of data can be found in Section 5.2.

Complex operators. For each kind of large operator (e.g., `conv2d`, `linear`, `bmm`), we build an 8-layer MLP model which takes the parameters of the input operator (e.g., the number of input/output channels, the size of filters, the size of input tensor) as the inputs, and produces the predicted run time of the individual operator. The training dataset is collected by measuring the runtime of each operator with randomly generated inputs and parameters on the *target* device. We use the same hyper-parameters and range of shapes as Habitat [30], with additional features about data types. For each operator, we uniformly sample hyper-parameters and shapes of input tensors, discard invalid inputs, and measure the time of three repetitive runs after three warm-up iterations.

Simple operators. DNNs contain numerous small element-wise operators, especially in activation layers and optimizers. They are usually memory-bound. We find that the runtime of element-wise operators is approximately equal to the memory I/O divided by the memory bandwidth, i.e. it scales linearly with I/O. For other small operators that the NN-based predictor does not cover, we treat them as element-wise operators, which might introduce some errors. However, the error would not affect the final result too much due to their small proportion in total runtime.

4.3 Implementation

In this section, we discuss how we handle the challenges we mentioned in Section 4.1.

Hardware-Dependent Optimizations. Many hardware-dependent optimizations can be represented in the graph-operator hierarchy. As a proof of concept, we showed how to support several popular optimizations such as changing batch size, automatic mixed precision training (AMP), and gradient checkpointing. For the rest of the optimizations, we discuss our approach to their prediction in Section 6.

- **Changing batch size:** Changing batch size does not change the timeline, but only scales the length of each operator. Batch size is already an input feature of the NN predictor, so it can naturally support variable batch sizes.
- **Low-precision:** We build an operator performance dataset for each precision and add one dimension of the input feature to represent the data type to the NN predictor. Some type casting operators might be introduced, which are treated as element-wise operators.
- **Gradient Checkpointing:** Checkpointing discards some intermediate computational results during forward propagation and re-computes them during backward propagation when they are needed. Its modification to the computational graph is limited to adding some forward computation operators in the re-computation phase. The PyTorch profiler can capture these modifications and include them in the timeline, and the newly added operators can be supported by the operator runtime predictor.

Consistency of GPU clocks. During sampling and measuring, the GPU keeps executing kernels with high computational intensity, leading to a lower clock speed as shown in Figure 7. We add sleep instructions between two measurements and adjust the interval to ensure the GPU clock speed is aligned with real-world scenarios. During testing, we need a long period of warm-up to ensure the GPU clock speed converges to be stable.

Data Augmentation. We enhance the ability of MLPs to fit discontinuous curves by data augmentation. In Figure 8 we find the period of discontinuity in terms of batch size is 16, so we add the remainder of the batch size divided by 16 to the input feature. We further convert it into a one-hot representation to strengthen its expressive power. So the feature representing the batch size $f_{\text{batch\_size}}$ can be formally written as (where $\oplus$ means concatenation):

$$f_{\text{batch\_size}} = [\text{batch\_size}] \ \oplus \ \text{onehot}(\text{batch\_size}\%16)$$

We apply similar augmentations to other parameters including input/output channels and image size, and concatenate the augmented features. Divisors other than 16 to get optimal accuracy also can be used. Since the time for training an NN-based predictor is negligible compared to that used in data sampling, we can perform a grid search to find the optimal parameters. This hyper-parameter is fixed after the prediction model is built. More details of the hyper-parameters used can be found in Table 3.

5 EVALUATION

BOOM is designed for *cross-device* performance prediction for DNN models, encompassing both memory and runtime prediction. Therefore our evaluations aims to determine: 1) BOOM can accurately predict the memory footprint of DNN models with different options and optimizations (batch size, AMP, gradient checkpointing); 2) BOOM can accurately predict the runtime of DNN models with different optimizations; 3) BOOM can predict the performance of DNN models exceeding the memory constraint of the local device. Additionally, we also compare BOOM with state-of-the-art baselines in terms of functionality and prediction accuracy.

Table 3: Details of feature engineering. Listed features are used to train the prediction model together with the one-hot representation of the resulting modulo (the remainder when divided by a chosen constant). Kernel size and stride in conv2d are also encoded as one-hot vectors.

batched matmul ($bsz \times L \times M \times N$)		**conv2d**	
bsz	$bsz \,\%\, 8$	bsz	$bsz \,\%\, 32$
L	$L \,\%\, 8$	$in\_channels$	$in\_channels \,\%\, 64$
M	$M \,\%\, 8$	$out\_channels$	$out\_channels \,\%\, 64$
N	$N \,\%\, 8$	$image\_size$	$image\_size \,\%\, 4$

Table 4: Hardware platforms we use in evaluation

GPU	GPU RAM	Micro Arch	FP32 TFLOPS
RTX 2080Ti	12 GiB	Turing	13.45
RTX 3090	24 GiB	Ampere	35.58
T4 (AWS)	16 GiB	Turing	8.14
V100 (AWS)	16 GiB	Volta	14.13
RTX 2070	8 GiB	Turing	7.47

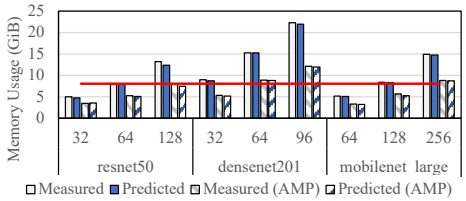
Hardware. We use four different NVIDIA GPUs as the target device: RTX 2080 Ti, RTX 3090, T4, and V100. We use RTX 2070 as our local device. These devices belong to different generations of microarchitecture and are offered in different forms. RTX 2070, RTX 2080, and RTX 3090 are offered as bare-metal machines, while Tesla T4 and Tesla V100 are offered as cloud instances (G4 and P3 respectively) by Amazon Web Services. They also vary in memory size and computation power (represented as FP32 TFLOPS in the table), among which the local device is the weakest one.

Software. We use PyTorch 1.12 and CUDA 11.6. Since we modified the source code of PyTorch, we need to build it from the source.

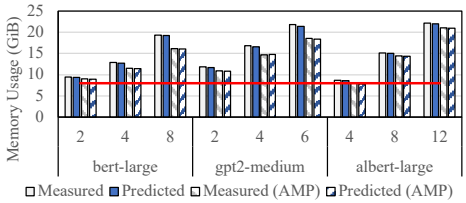
Baselines. We choose state-of-the-art performance prediction tools as baselines to compare in our evaluation. However, there are no open-source cross-device memory predictors available to the best of our knowledge. To address this, we re-implemented DNNPerf [28] based on the description in their paper. For runtime prediction, Daydream [87] and Skyline [80] are not designed for cross-device prediction. Habitat [30] is a cross-device runtime predictor that employs wave scaling and ML-based prediction. Therefore, we select Habitat as the baseline for comparison.

5.1 Evaluating Memory Predictor

In this section, we evaluate our memory predictor on different models including CNNs and transformers. We also evaluate the predictor with Automatic Mixed Precision (AMP). We implement a two-layer fake memory allocator in PyTorch as shown in Section 3a and block the launch of many operators, with less than 500 lines of code committed. We use RTX 2070 (8 GiB) as our local device and RTX 3090 (24 GiB) as the target device to highlight the substantial difference in GPU memory capacity.



(a) Memory usage prediction on CNNs



(b) Memory usage prediction on transformers

Figure 10: Memory footprint prediction of different input models and batch sizes. The memory constraint of the local device (8 GiB) is highlighted in red.

CNNs. We tested our memory predictor on three CNN models from Torchvision [21], namely ResNet50 [31], DenseNet201 [34] and $\text{MobileNetV3}_{\text{Large}}$ [32]. We turn on the cuDNN benchmark option and exclude the first iteration where the benchmark happens. We choose three different batch sizes for each model. Figure 10a shows the result. Note that the GPU memory of the local device is 8 GiB. So the results beyond 8 GiB show the ability of the predictor to simulate models exceeding the local device memory constraint. The overall prediction error of the test cases is 2.7%.

Transformers. We use three transformer models for sequence prediction from HuggingFace's `transformers` library [79], namely $\text{BERT}_{\text{LARGE}}$ [23], $\text{GPT2}_{\text{MEDIUM}}$ [62] and $\text{ALBERT}_{\text{LARGE}}$ [46]. We set the sequence length as 512 and vary the batch sizes. Figure 10b shows the result. The overall prediction error of the test cases is 0.9%. We find the error of predicting the memory footprint of transformer models is smaller than CNN models and we attribute it to the inconsistency generated in the algorithm selection of cuDNN in different hardware.

Comparison to state-of-the-art baseline. We compare our predictor with the ML-based memory predictor in DNNPerf [28]. Since it's not open-source, we implement it according to the description in the paper. We collect 9,062 CNN models from NATS-Bench [24], a NAS dataset, and measure their memory footprint. We randomly sample 70% data as the training dataset and leave the rest for testing. We also measure ResNet18 with the image size as 224×224 and batch sizes from 8 to 128 as the unseen testing set.

As shown in Table 5, BOOM achieves a comparable accuracy on the NAST dataset while winning by a wide margin on the unseen ResNet tests. Note that the NAST training and testing sets are very similar, so it is an optimistic measurement for DNNPerf. It shows our method has a stronger generalization ability.

Table 5: Accuracy comparison of memory footprint prediction between BOOM and DNNPerf

	BOOM	DNNPerf
NAST	3.5 %	3.3%
ResNet (unseen)	6.6%	25.2%

5.2 Evaluating Runtime Predictor

End-to-end prediction accuracy. We test our runtime predictor on three CNN models (VGG19 [69], ResNet50 [31], InceptionV3 [73]) from Torchvision and two transformer models $\text{BERT}_{\text{BASE}}$ and GPT2 from Hugging Face `transformers`, with different batch sizes in both FP32 and AMP training. We measure the wall-clock time of 100 contiguous end-to-end training iterations after 100 iterations of dry run to avoid the GPU clock inconsistency problem (Section 4.1).

For FP32 training, we compare BOOM with Habitat, of which the result is shown in Figure 11. The prediction error given by BOOM is 7.8%, while the error of Habitat is 13.5%. Since Habitat does not support automatic mixed precision (AMP), we only report the error of BOOM, as shown in Figure 12. BOOM achieves an average error of 13.3% on AMP. The overall testing error including FP32 and AMP training of BOOM is 10.5%.

In conclusion, BOOM can achieve higher accuracy than Habitat, support predicting DNNs exceeding the local device memory, and support more features such as AMP.

Data amount and operator runtime prediction accuracy. Sampling and measuring are the most costly stages in the process of building an NN-based prediction model. Hence, it is important to estimate the required amount of data for training an accurate enough model. We randomly sample subsets of different sizes from the entire training set, train NN predictors on them respectively, and test them on the testing set. Figure 13 shows the testing error-data amount curves. We observe that with a dataset size of 1.5×10^5, the testing error can be lower than 15%. The testing error converges around the data amount equal to 4×10^5 (including both FP32 and AMP). For other operators, a dataset with a size of 10^4 can produce a satisfactory result. We utilize this as a reference for data collection on other platforms.

5.3 Evaluating the Support of Checkpointing

In this section, we test the accuracy of BOOM's runtime and memory usage predictor on training with the optimization.

We use RTX 2070 (8 GiB) as the local device and RTX 3090 (24 GiB) as the target device. We use two models $\text{GPT2}_{\text{MEDIUM}}$ and $\text{GPT2}_{\text{LARGE}}$, with three different batch sizes. These models are so large that they cannot run on the target device. Figure 14 shows the result. The memory usage predictor achieves an average error of 1.3%, and the runtime predictor 4.5%, showing our predictors can work well with gradient checkpointing. Note that the memory footprint is larger than the local device memory (8 GiB) in some test cases even after enabling gradient checkpointing, showing BOOM can work with models exceeding the local device memory.

The required code modifications to support gradient checkpointing are minimal. For memory prediction, BOOM seamlessly supports it without any changes to the source code. For runtime prediction, the checkpointed operators are wrapped as a `CheckpointFunction` by PyTorch, of which the operators can be extracted by a few additional lines of code.

5.4 Prediction Time

We measure the time spent by BOOM on end-to-end prediction for different models and compare it with Habitat, as shown in Figure 15. While BOOM eliminates most of the unnecessary computation for performance prediction, it also suffers from the overhead of switching between the fake and original PyTorch environment. These two factors offset each other, hence BOOM and Habitat have comparable prediction overhead. In some cases where computation is heavier such as $\text{BERT}_{\text{BASE}}$, BOOM runs faster because it eliminates more unnecessary computations. In comparison, Habitat requires fully running the model on the local device to calibrate its predictions, which cannot be skipped. In conclusion, the prediction overhead of BOOM is around 10 seconds and acceptable in practice.

6 DISCUSSION

Besides the optimizations we mentioned in Section 4.3, many other optimizations can also be supported under our framework as extensions. We leave these as future work.

Distributed. BOOM can be extended to support distributed training. One possible solution is to replace distributed primitives such as `all_reduce` with their corresponding fake versions, similar to BOOM's memory fake allocators. This will let us track the network traffic volume and calculate the communication cost using network bandwidth accurately.

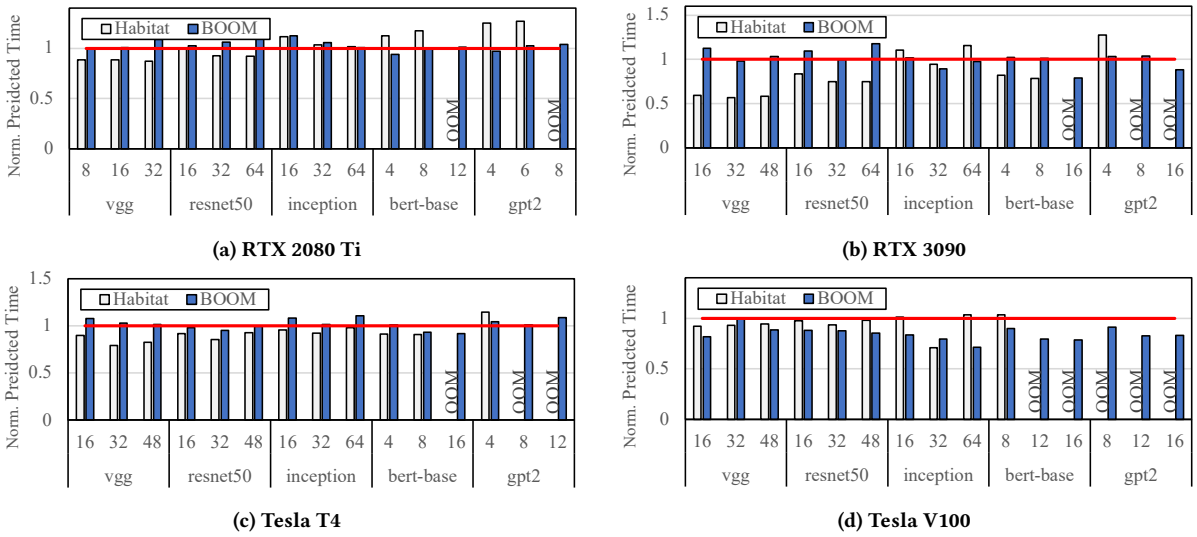
Offloading. One solution to insufficient device memory is to swap some weight or activation tensors to the host device and load them back to the device when needed [33, 61, 65–67]. For memory prediction, if the CPU memory is large enough that we can view it as infinitely large, then BOOM can naturally support it. Otherwise, we need to manipulate the CUDA memory copy APIs.

Sparsification. Sparsification is a class of techniques to accelerate the training and inference of DNNs. Some of them use fixed sparsified operators [15, 20], which can be handled by the NN-based operator performance predictor. Memory usage prediction can also be naturally supported by BOOM.

Kernel Fusion. Kernel fusion merges contiguous operators in the computational graph [59]. It can reduce redundant data movements by reusing intermediate results of contiguous operators, which can be approximated by the amount of I/O and device memory bandwidth. Memory usage prediction of both kinds of kernel fusion can be naturally supported by BOOM.

7 RELATED WORK

Memory prediction of deep learning. Existing DNN memory prediction techniques can be categorized into (i) *computational graph-based* methods and (ii) *profile-then-extrapolate* methods. However, they are all not satisfactory in terms of handling the aforementioned challenges (Section 3.1). The former takes the computational graphs as the input and directly analyzes them to make predictions. Both analytical models [29, 64] and ML-based memory predictors [28] require substantial engineering efforts, either to



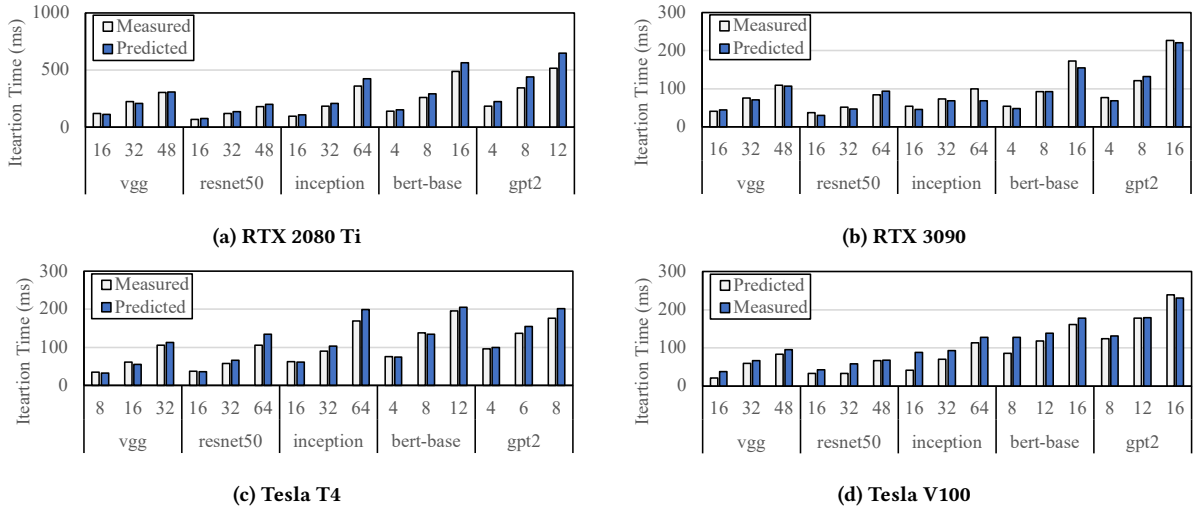
(a) RTX 2080 Ti

(b) RTX 3090

(c) Tesla T4

(d) Tesla V100

Figure 11: Predicted runtime given by BOOM and Habitat, normalized with respect to the ground truth. Ground truth 1 is highlighted in red. OOM represents the data points where Habitat fails due to insufficient memory of the local device.



(a) RTX 2080 Ti

(b) RTX 3090

(c) Tesla T4

(d) Tesla V100

Figure 12: End-to-end evaluation of the runtime predictor with the Automatic Mixed Precision (AMP).

manually model the memory footprint, or build a machine learning pipeline. It is also possible to predict memory footprint in a profile-then-extrapolate way(e.g. Skyline [80]), where we first run the DNN program on the *local device* users already have in hands, and extrapolate based on collected runtime data (like memory footprint). However, such a strategy would not work for large models where the memory footprint exceeds the device memory constraint, even with a batch size equal to one.

Runtime prediction of deep learning. Daydream[87] is a heuristic-based performance predictor for DNN optimizations applied on a *fixed* device. Lin *et al.* [52] models CPU and GPU time in a more fine-grained way, specifically for recommendation models. PerfNetRT [50] proposes a platform-aware performance predictor for optimized DL inference tasks, while our work supports model training. Some of them target a specific class of kernels like convolutions [53]. Habitat [30] uses heuristics for simple operators and an ML-based method for complex operators where different kernels might be selected for different input shapes. However, it does not consider optimizations. Habitat also needs to run the model on the local device to collect the trace, which is constrained by the limited memory, while BOOM solves this problem with the fake memory allocator. Skyline [80] predicts DNN runtime based on local profiling results and linearly extrapolating, which requires running the model on local devices.

Cost model in auto-tuning for DL models. Many DL optimization techniques adopt auto-tuning, which often relies on a cost

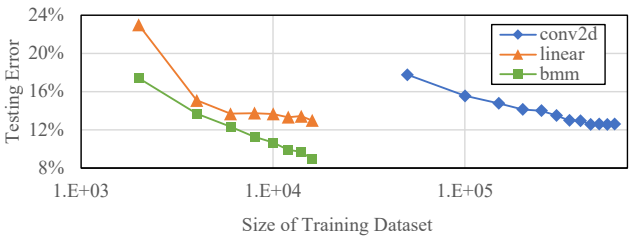
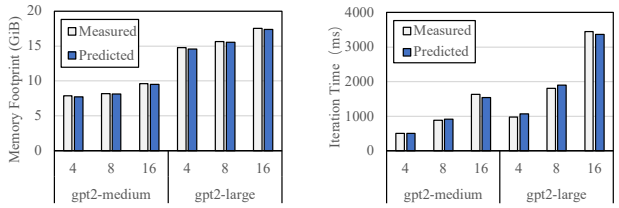


Figure 13: How data amount affects the quality of the NN predictor, measured on RTX 2080Ti. To get an accurate enough runtime prediction model, we need around 4×10^5 data points for 2D convolutions and 10^4 for the rest.



(a) Memory Usage Prediction **(b) Runtime Prediction**

Figure 14: End-to-end evaluation of the memory and runtime predictor with gradient checkpointing.

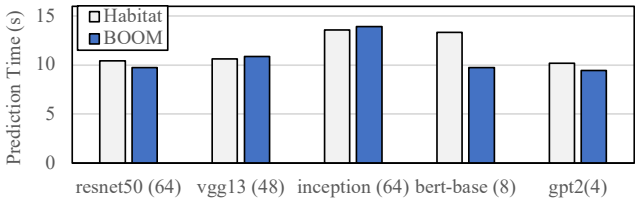


Figure 15: Time required to predict the runtime of different models. The numbers in parentheses are the batch size we use in each experiment.

model to predict the runtime. Auto-schedulers [10, 12, 18, 27, 42, 81, 85] for Tensor compilers like Halide [63] and TVM [16] often have an ML-based cost model. Srivastava [71] and Singh *et al.* [70] propose using neural networks to predict the end-to-end model inference performance optimized by tensor compilers. These compiler-based optimizations are mostly targeted at sub-operator levels like loops, which is different from our work which works on the operator level.

8 CONCLUSION

We propose BOOM, a novel performance predictor supporting accurate memory usage and runtime prediction of DNN tasks. The memory usage predictor replaces the GPU memory allocator with a fake allocator which enables running large models beyond the memory constraint of the local device, hence reducing memory usage prediction to memory usage measurement. The runtime predictor supports predicting the runtime of models after hardware-dependent optimizations, such as changing batch size, mixed-precision training, and gradient checkpointing. Experiments show that our memory footprint predictor achieves an average error of 2.7% on CNN models and 0.9% on transformers, and the runtime predictor achieves an average error of 10.5% on the CNN and transformer models for FP32 and mixed precision training. For DNN training with gradient checkpointing, BOOM achieves an error of under 5%.

ACKNOWLEDGMENTS

We want to express our sincere gratitude to the anonymous PACT reviewers and the shepherd for their valuable and constructive feedback and suggestions. The authors with the University of Toronto are supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

REFERENCES

[1] 2023. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.
[2] 2023. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia/.
[3] 2023. AWS Trainium. https://aws.amazon.com/machine-learning/trainium/.
[4] 2023. cuBLAS. https://docs.nvidia.com/cuda/cublas/.
[5] 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.
[6] 2023. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.
[7] 2023. PyTorch Memory Management. https://pytorch.org/docs/stable/notes/cuda.html#memory-management.
[8] 2023. SambaNova DataScale. https://sambanova.ai/products/datascale/.
[9] 2024. vast.ai. https://vast.ai/.
[10] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.
[11] Muralidhar Andoorveedu, Zhanda Zhu, Bojian Zheng, and Gennady Pekhimenko. 2022. Tempo: Accelerating Transformer-Based Model Training through Memory Footprint Reduction. *arXiv preprint arXiv:2210.10246* (2022).
[12] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. 2021. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems* 3 (2021), 181–193.
[13] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).
[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[15] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. 2021. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029* (2021).
[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 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.
[17] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).
[18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).
[19] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[20] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).
[21] Torch Contributors. 2021. Torchvision. models. *visited June* 28 (2021).
[22] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj D. Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan,

Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesús Corbal, Nikita Shustrov, Roman Dubtsov, Evarist Fomenko, and Vadim O. Pirogov. 2018. Mixed Precision Training of Convolutional Neural Networks using Integer Operations. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=H135uzZ0-

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[24] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. 2021. NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2021). https://doi.org/10.1109/TPAMI.2021.3054824 doi:10.1109/TPAMI.2021.3054824.

[25] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[26] R David Evans, Lufei Liu, and Tor M Aamodt. 2020. Jpeg-act: accelerating deep learning via transform-based lossy compression. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 860–873.

[27] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2022. Tensorir: An abstraction for automatic tensorized program optimization. *arXiv preprint arXiv:2207.04296* (2022).

[28] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2021. *Runtime Performance Prediction for Deep Learning Models with Graph Neural Network*. Technical Report. Technical Report MSR-TR-2021-3. Microsoft.

[29] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1342–1352.

[30] X Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A {Runtime-Based} Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 503–521.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[32] 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.

[33] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.

[34] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[35] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[36] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 776–789.

[37] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems* 2 (2020), 497–511.

[38] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[39] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413* (2019).

[40] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and evaluation of the first tensor processing unit. *ieee Micro* 38, 3 (2018), 10–19.

[41] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the computational cost of deep learning models. In *2018 IEEE international conference on big data (Big Data)*. IEEE, 3873–3882.

[42] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems* 3 (2021), 387–400.

[43] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE micro* 31, 5 (2011), 7–17.

[44] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, Duane Merrill, Aniket Shivam, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Matt Nicely. 2022. CUTLASS. https://github.com/NVIDIA/cutlass

[45] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).

[46] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[47] Rebecca Lewington. 2021. An AI Chip With Unprecedented Performance To Do the Unimaginable. (2021).

[48] Xiaoyao Liang. 2019. Ascend AI Processor architecture and programming.

[49] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing.. In *Hot Chips Symposium*. 1–44.

[50] Ying-Chiao Liao, Chuan-Chi Wang, Chia-Heng Tu, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. 2020. PerfNetRT: Platform-Aware Performance Modeling for Optimized Deep Neural Networks. In *2020 International Computer Symposium (ICS)*. IEEE, 153–158.

[51] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[52] Zhongyi Lin, Louis Feng, Ehsan K Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Arun Kejariwal, and John D Owens. 2022. Building a Performance Model for Deep Learning Recommendation Model Training on GPUs. *arXiv preprint arXiv:2201.07821* (2022).

[53] Guodong Liu, Sa Wang, and Yungang Bao. 2021. SEER: A Time Prediction Model for CNNs from GPU Kernel's View. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 173–185.

[54] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 393–405.

[55] Yao Lu, Song Bian, Lequn Chen, Yongjun He, Yulong Hui, Matthew Lentz, Beibin Li, Fei Liu, Jialin Li, Qi Liu, Rui Liu, Xiaoxuan Liu, Lin Ma, Kexin Rong, Jianguo Wang, Yingjun Wu, Yongji Wu, Huanchen Zhang, Minjia Zhang, Qizhen Zhang, Tianyi Zhou, and Danyang Zhuo. 2024. Computing in the Era of Large Generative Models: From Cloud-Native to AI-Native. arXiv:2401.12230 [cs.DC]

[56] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2020. Mlperf training benchmark. *Proceedings of Machine Learning and Systems* 2 (2020), 336–349.

[57] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Analysis and exploitation of dynamic pricing in the public cloud for ml training. In *VLDB DISPA Workshop 2020*.

[58] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[59] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[60] 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).

[61] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905.

[62] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[63] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[64] Aditya Rajagopal and Christos-Savvas Bouganis. 2021. perf4sight: A toolflow to model CNN training performance on Edge GPUs. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 963–971.

[65] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[66] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 598–611.

[67] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[68] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[69] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[70] Shikhar Singh, James Hegarty, Hugh Leather, and Benoit Steiner. 2022. A graph neural network-based performance model for deep learning applications. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 11–20.

[71] Ajitesh Srivastava, Naifeng Zhang, Rajgopal Kannan, and Viktor K Prasanna. 2020. Towards high performance, portability, and productivity: Lightweight augmented neural networks for performance prediction. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 21–30.

[72] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 13693–13696.

[73] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[74] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[75] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[77] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. {PET}: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 37–54.

[78] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. *Proceedings of Machine Learning and Systems* 3 (2021), 599–623.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[80] Geoffrey X Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 126–139.

[81] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 833–845.

[82] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A Deep Learning {Exploratory-Training} Framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 919–936.

[83] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.

[84] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1089–1102.

[85] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[86] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. [n. d.]. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 88–100.

[87] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for {DNN} Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 337–352.