Estimating GPU Memory Consumption of Deep Learning Models

Yanjie Gao
Microsoft Research
China
yanjga@microsoft.com

Yu Liu*
Microsoft Research
National University of Singapore
Singapore
liuyu@comp.nus.edu.sg

Hongyu Zhang
The University of Newcastle
Australia
hongyu.zhang@newcastle.edu.au

Zhengxian Li
Microsoft Research
China
v-zhli10@microsoft.com

Yonghao Zhu
Microsoft Research
China
v-yonghz@microsoft.com

Haoxiang Lin†
Microsoft Research
China
haoxlin@microsoft.com

Mao Yang
Microsoft Research
China
maoyang@microsoft.com

ABSTRACT

Deep learning (DL) has been increasingly adopted by a variety of software-intensive systems. Developers mainly use GPUs to accelerate the training, testing, and deployment of DL models. However, the GPU memory consumed by a DL model is often unknown to them before the DL job executes. Therefore, an improper choice of neural architecture or hyperparameters can cause such a job to run out of the limited GPU memory and fail. Our recent empirical study has found that many DL job failures are due to the exhaustion of GPU memory. This leads to a horrendous waste of computing resources and a significant reduction in development productivity. In this paper, we propose DNNMem, an accurate estimation tool for GPU memory consumption of DL models. DNNMem employs an analytic estimation approach to systematically calculate the memory consumption of both the computation graph and the DL framework runtime. We have evaluated DNNMem on 5 real-world representative models with different hyperparameters under 3 mainstream frameworks (TensorFlow, PyTorch, and MXNet). Our extensive experiments show that DNNMem is effective in estimating GPU memory consumption.

CCS CONCEPTS

• **Software and its engineering → Extra-functional properties**.

KEYWORDS

deep learning, memory consumption, estimation model, program analysis

*This author's work is done as an intern at Microsoft Research.
†Corresponding author.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417050

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3417050

1 INTRODUCTION

In recent years, deep learning (DL) has rapidly become one of the most successful machine learning techniques and is widely integrated into a variety of software-intensive systems (such as computer vision systems, natural language processing systems, games, etc.). To accelerate the training, testing, and deployment of DL models (aka deep neural networks or DNNs), GPU (Graphics Processing Unit) is widely adopted by the developers. Enterprises also build dedicate DL platforms such as Amazon SageMaker [4] and Microsoft Azure Machine Learning [5] with a large number of GPUs, providing support for DL frameworks like TensorFlow (TF) [1], PyTorch [35], and MXNet [9].

However, since the GPU memory consumed by a DL model is often unknown to developers before the training or inferencing job starts running, an improper model configuration of neural architecture or hyperparameters can cause such a job to run out of the limited GPU memory and fail. For example, as shown in Figure 1, if a PyTorch ResNet50 [18] training job with a batch size of 256 is scheduled on the NVIDIA Tesla P100 GPU, it will trigger an *OOM* (out-of-memory) exception because the DL model requires 22 GB of GPU memory while P100 has only 16 GB in total.

According to our recent empirical study on 4960 failed DL jobs in Microsoft (Section 2.1), 8.8% of the job failures were caused by the exhaustion of GPU memory, which accounts for the largest category in all deep learning specific failures. Therefore, knowing the accurate GPU memory consumption (aka memory footprint) in advance is very important to reduce OOM failures and save precious platform resources including GPU/CPU/storage, by helping developers choose an optimal model configuration or facilitating DL frameworks to better utilize the mechanisms of dynamic memory management [17] (*e.g.*, GPU memory swapping). This ability can also benefit AutoML tools in enhancing the search efficiency

(*e.g.*, excluding those model configurations that do not satisfy the memory requirement) and DL platforms in optimizing job planning and scheduling (*e.g.*, scheduling a group of DL jobs that can maximize the GPU memory usage).

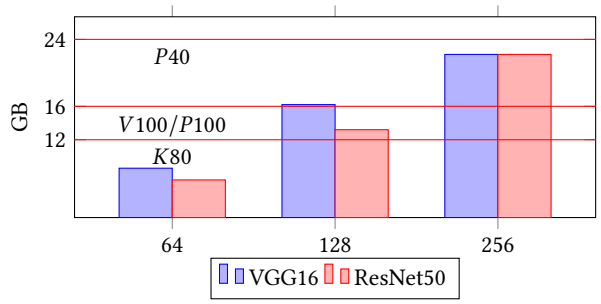

Figure 1: GPU memory consumption of training PyTorch VGG16 [42] and ResNet50 models with different batch sizes. The red lines indicate the memory capacities of three NVIDIA GPUs.

There are already many program analysis based techniques [2, 6, 7, 12, 22, 46, 47] for estimating memory consumption of C, C++, and Java programs. For example, Albert *et al.* [2] presented a parametric inference on the notion of object lifetime to inferring memory requirements of Java-like programs. Heo *et al.* [19] proposed a resource-aware (*e.g.*, memory size) flow-sensitive analysis that can adjust behaviors by coarsening program abstraction. However, existing work cannot be directly applied to DL models for the following three main reasons:

(1) The hybrid programming paradigm adopted by DL frameworks hides the internal execution of a DL model from the high-level programs written by developers, therefore making it difficult to track the precise GPU memory usage.
(2) It is hard to analyze the GPU memory usage of low-level framework operators (*e.g.*, *Conv2d*), since they are usually implemented with proprietary NVIDIA cuDNN, cuBLAS, or CUDA APIs and nested loops.
(3) There are many hidden factors within the framework runtime, which could observably affect the final GPU memory consumption, including allocation policy (*e.g.*, tensor alignment, fragmentation, reservation, and garbage collection), internal usage (*e.g.*, CUDA context), implementation choice (*e.g.*, multiple convolution algorithms in cuDNN [33]), operator scheduling, etc.

Simple workarounds cannot precisely estimate the GPU memory consumption. First, the *Shape Inference* capability of DL frameworks [25, 34] could be adapted for the estimation, by statically adding up all the GPU memory allocated to the initial input, operator weights, intermediate outputs, and final output. However, it does not take into account the above-mentioned hidden framework factors, leading to large estimation errors (Section 5.2). Second, running a DL job for a while and profiling it dynamically may help estimate how much GPU memory is required. Nevertheless, such a resource-consuming workaround cannot avoid GPU OOM either and is unaffordable in the scenario of hyperparameter tuning, where a large number of possible neural architectures and hyperparameter combinations exist.

This paper presents DNNMem, an accurate estimation tool for GPU memory consumption of DL models. Our key observation is that the algorithmic execution of a DL model can be represented as iterative forward and backward propagation on its *computation graph*. Such a graph is a directed acyclic graph (DAG), where each node is an invocation of a mathematical function called *operator* (*e.g.*, matrix addition) and each edge specifies the execution dependency. GPU memory is allocated to tensors (*e.g.*, operator inputs/outputs, and learnable parameters) and temporary buffers (*e.g.*, cuDNN workspace), and is later released by the framework's built-in *memory allocator* [15] along with the execution of operators. Hence, estimating GPU memory consumption can be reduced to the calculation of memory required by each operator on the computation graph in accordance with a graph traversal order. For an operator, DNNMem defines an analytic and framework-independent memory cost function since the operator is well defined with similar implementations across different frameworks. DNNMem also extracts many of the above-mentioned runtime factors from each supported framework to refine the estimation. For example, it analyzes the liveness of tensors to handle GPU memory deallocation. DNNMem is general and applies to not only single-device training but also data-parallel training and model inference.

We have implemented DNNMem and evaluated it on 5 real-world representative models (VGG16 [42], ResNet50 [18], InceptionV3 [43], LSTM [20], and BERT [14]) with different hyperparameters under 3 mainstream DL frameworks (TensorFlow, PyTorch, and MXNet). The average estimation errors are below 16.3%, confirming the effectiveness of our proposed approach. The results also show that DNNMem is robust to the choices of neural architectures, hyperparameters, and DL frameworks.

In summary, this paper makes the following contributions:

(1) We systematically explore how GPU memory is consumed by DL models.
(2) We propose and implement DNNMem, which can accurately estimate the GPU memory consumption of a DL model.
(3) We perform comprehensive evaluations of DNNMem on a variety of DL models and frameworks. The results show the effectiveness and robustness of DNNMem.

2 BACKGROUND AND MOTIVATION

2.1 The Out-of-Memory Problem in DL Practice

We recently conducted an empirical study on 4960 failed DL jobs collected from the Philly platform in Microsoft within a three-week period [51]. Every day, thousands of jobs from both research and product teams are executed on Philly, including machine translation, reading comprehension, object detection, gaming, advertisement, etc. For each failed job, we collected all related information including input data, source code, job scripts, execution logs, and runtime statistics for analysis. Failures in our study manifested as unexpected runtime errors that led to job termination.

In our empirical study, we analyzed the categories and the root causes of DL job failures. Our study shows that 8.8% of the total failures were caused by the exhaustion of GPU memory, which accounts for the largest category in all deep learning specific failures. The DL models with sophisticated network structures and large

batch sizes may improve the model learning performance but also significantly increase memory consumption. Since GPU memory is relatively limited, developers need to size the model very carefully.

In fact, the OOM problem is not specific to the DL jobs in Microsoft. Another empirical study on 2716 Stack Overflow posts also listed OOM as one of the six major effects of deep learning bugs [21]. Therefore, knowing the accurate GPU memory consumption in advance is very important to reduce out-of-memory failures and save precious platform resources. A memory usage estimation tool is very useful in this regard.

2.2 A Motivating Example of Our Approach

We motivate the design of DNNMem by describing how GPU memory is used and calculated for a simplified PyTorch training program. Developers use *deep learning frameworks* such as TensorFlow (TF) [1], PyTorch [35], and MXNet [9] to design layered data representations called deep neural networks (DNNs) or deep learning models. These frameworks provide both high-level programming interfaces and basic building blocks for model construction. DL models are essentially mathematical functions, which can be formalized as *tensor-oriented computation graphs*. Inputs and outputs of the graph nodes are ***tensor*** (multi-dimensional array of numerical values) variables. The shape of a tensor is the element number in each dimension plus element data type. Each node represents the invocation of a mathematical operation called an ***operator*** (*e.g.*, matrix addition). Since a node is completely decided by its invoked operator, we may use "node" and "operator" interchangeably in the rest of the paper. Each operator may additionally contain some numerical *learnable parameters* (*i.e.*, weights[1]). A graph edge pointing from one output of operator A to one input of B delivers a tensor and specifies the execution dependency.

Figure 2 shows the sample PyTorch training program, which sets up a sequential model using the framework built-in *Conv2d* (2D convolution), *AvgPool2d* (2D average pooling), and *Linear* (fully-connected layer) operators (lines 5-12). The original code does not give enough clues on how the training is processed. Under the hood, PyTorch constructs a computation graph depicted in Figure 3 and applies *iterative forward and backward propagation* on it to learn the optimal weights. Such a graph is augmented with some system crafted operators for backward propagation (*e.g.*, *AvgPool2d_BP* in the middle of Figure 3).[2] Under forward propagation (the left of Figure 3), input data (*Data_X*) is fed through the neural network and manipulated by the above developer-specified operators. Produced output activations and input labels (*Data_Y*) are then propagated backward to compute weight gradients. Finally, an optimizer is responsible for weight update to minimize the *loss* (*e.g.*, the difference between actual and expected outputs), marking the end of one iteration. Popular optimization algorithms include Adam [23], RMSProp [45], and SGD (stochastic gradient descent) [3].

During the training, operators apply for necessary GPU memory on demand to store the following dimensions of tensors, denoted by the ovals in Figure 3:

```
1  import torch
2  class Net(nn.Module):
3      def __init__(self):
4          super(Net, self).__init__()
5          self.conv = nn.Conv2d(3, 8, 3)
6          self.pool = nn.AvgPool2d(2, 2)
7          self.fc = nn.Linear(1800, 10)
8      def forward(self, x):
9          x = self.conv(x)
10         x = self.pool(x)
11         x = x.view(x.size(0), -1)
12         x = self.fc(x)
13         return x
14
15 model = Net().cuda()
16 for epoch in range(500):
17     ...
18     outputs = model(inputs)
19     loss = criterion(outputs, labels)
20     loss.backward()
21     optimizer.step()
```

Figure 2: A sample PyTorch training program which constructs a sequential DL model using *Conv2d* (2D convolution), *AvgPool2d* (2D average pooling), and *Linear* (fully-connected layer) operators.

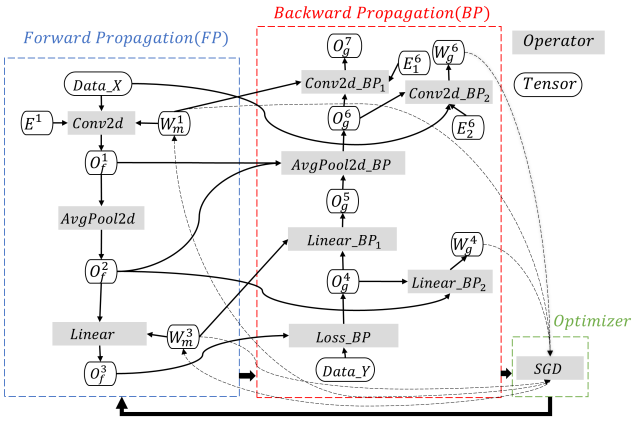

Figure 3: Computation graph for training the DL model in Figure 2. Ovals represent tensors in which W stands for weight tensor, O for In/Out tensor, and E for ephemeral tensor. Rectangles are operators.[2] Dash lines denote weight updates by SGD.

(1) *Weight Tensor.* This dimension includes operator weights (*e.g.*, W_m^1), and weight gradients (*e.g.*, W_g^6) computed under backward propagation for updating weights.
(2) *In/Out Tensor.* This dimension includes the initial input (*Data_X* for features and *Data_Y* for labels) and operator inputs/outputs. Outputs are further distinguished to forward outputs (*e.g.*, O_f^1), and output gradients (*e.g.*, O_g^6) for calculating weight gradients. We do not draw operator inputs because they are identical to the corresponding predecessors' outputs. Note that they may occupy separate GPU memory buffers under certain circumstances (*e.g.*, in model-parallel training).

[1] Weight biases are included.

[2] We split the backward propagation of *Linear* into two logical operators $Linear\_BP_1$ and $Linear\_BP_2$ to clearly demonstrate the computation of weight and output gradients. The same is to $Conv2d\_BP_1$ and $Conv2d\_BP_2$.

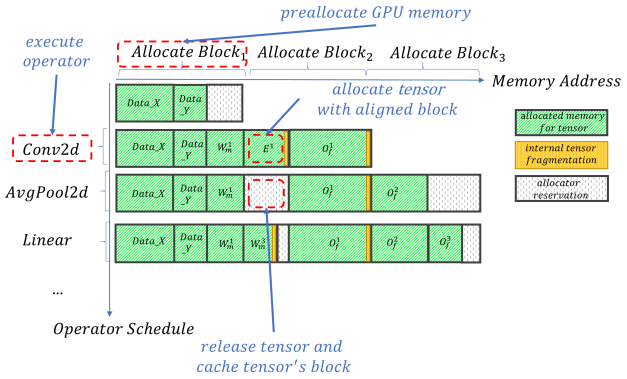

Figure 4: GPU memory allocation during the operator execution.

(3) *Ephemeral Tensor.* This dimension includes variables used by cuDNN/cuBLAS/CUDA APIs such as cuDNN workspace (*e.g.*, part of E^1) and declared CUDA random numbers.

In addition, a DL model also requires some *resident buffers*. For example, extra GPU memory is allocated for tensors to meet the alignment requirements (*i.e.*, internal tensor fragmentation). Others include the CUDA context,[3] runtime reservation, etc.

Figure 4 illustrates how GPU memory is possibly consumed in training the above motivating DL model. The vertical axis represents the operator execution ordering such that *Conv2d* executes first, *AvgPool2d* is the second, *Linear* then follows, etc. The horizontal axis shows the consumed GPU memory when a certain operator is executing. The GPU memory consumption of a DL model is the total GPU memory consumption applied by the framework from the device, which can be logically viewed as a continuous area divided into memory blocks (rectangles in Figure 4). Green parts are the allocated memory for in-use tensors. Yellow parts are the internal tensor fragmentation if the original tensor sizes do not align to a power of two. The gray parts are the reserved memory by the framework allocator. For example, after a tensor is out of use, its memory block could be cached instead of returning to the GPU immediately. Since the CUDA context is allocated when DL frameworks initialize, we do not draw it on the figure.

Initially, before the operator execution (the first line in Figure 4), GPU memory is applied for the two initial input tensors $Data\_X$ and $Data\_Y$ and extra memory (the rightmost gray rectangle) is pre-allocated to make future allocation more efficient. When *Conv2d* executes, the framework allocator pads a little more GPU memory as the internal tensor fragmentation to the ephemeral tensor E^1 since its size is not aligned. After *Conv2d* has finished, E^1 reaches the end of its life and is then released. However, the corresponding memory block is cached and will be re-allocated to W_m^3 when *Linear* starts. The remaining space (the gray rectangle next to W_m^3) is too small for any later tensors, therefore it becomes an external tensor fragmentation and waits for being garbage-collected.

[3]The CUDA context contains managing information to control and use GPU devices.

[4]H, W, and C represent the height, width, and channel dimensions of an image input tensor, respectively.

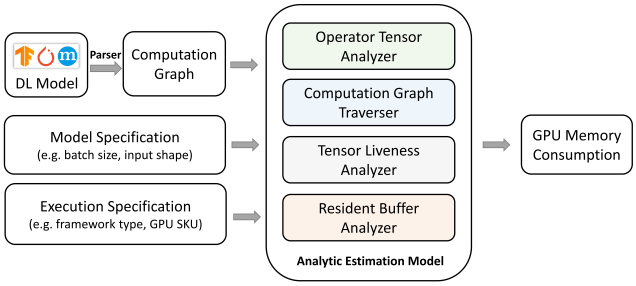

Figure 5: Architecture of DNNMem.

Table 1: Selected settings in the model (upper part) and execution specifications. Mark "*" represents the default value.

Specification Settings	Example Values	Affected Symbols
Framework	TF * / PyTorch / MXNet	$M_{ctx}, MR(u)$
Input Shape	(H:224, W:224, C:3)[4]	$O(u)$
Batch Size	128	$O(u)$
Optimization Algorithm	SGD * / Adam	$W(u)$
Precision Format	Float32 * / Double	M_{CG}
Execution Mode	single-* / multi-device	$W(u)$
GPU SKU	P40	M_{ctx}
cuDNN Workspace Limit	4 GB	$E(u)$

3 PROPOSED APPROACH

To fully understand how GPU memory is used by a DL model, we classify the allocated GPU memory into 4 dimensions and present them in Table 2. Our key observation is that the algorithmic execution of a DL model is represented by frameworks as iterative forward and backward propagation on its computation graph. Propagation follows the execution dependency between operators, being specified by the graph edges. The operator scheduling (*i.e.*, in which ordering the framework executes operators) is influential to the GPU memory consumption since it could affect memory deallocation, preservation, garbage collection, etc. DNNMem reduces the operator scheduling to the computation graph traversal. Currently, since DL frameworks schedule one operator after another,[5] we assume that operators are traversed *sequentially*. Therefore, DNNMem adopts an analytic approach that formalizes the estimation of GPU memory consumption as the calculation of memory required by each operator on the computation graph in accordance with a topological (linear) graph traversal ordering (Section 4.1). Such an ordering is pre-generated by referring to the framework implementations [17, 31, 37]

Figure 5 illustrates the architecture of DNNMem. It accepts the on-disk serialized model file(s), a model specification, and an execution specification as the input and then reports the estimated GPU memory consumption. The model specification includes input tensor shape and hyperparameter values (*e.g.*, kernel size of some convolutional operator). The execution specification contains runtime information such as execution mode (*e.g.*, single-device)

[5]MXNet can be configured to execute several operators simultaneously (*i.e.*, bulk execution).

Table 2: Classification of allocated GPU memory.

Dimension	Category	Description
Weight Tensor	Weight	Learnable parameters of operators
	Weight Gradient	Gradients computed under backward propagation for updating weights
In/Out Tensor	Initial Input	Input data items in mini batches
	Operator Input	Inputs of an operator (identical to the corresponding predecessors' outputs if the predecessors reside on the same GPU)
	Forward Output	Outputs of an operator computed under forward propagation (including the model's final output such as O_f^3 in Figure 3)
	Output Gradient	Gradients under backward propagation for calculating weight gradients
Ephemeral Tensor	cuDNN Workspace	Additional GPU memory used by cuDNN APIs
	Temporary Tensor	Temporary variables used in operator implementation
Resident Buffer	CUDA Context	Managing information to control and use GPU devices
	Internal Tensor Fragmentation	Extra memory allocated for alignment
	Allocator Reservation	(1) Released yet unreclaimed tensors (2) Pre-allocated memory (3) External tensor fragmentation (4) Miscellaneous reservation (*e.g.*, the fusion buffer used by Horovod)

and GPU SKU (Stock Keeping Unit) (*e.g.*, GPU type, and memory capacity). Some specification settings are shown in Table 1.

DNNMem implements a *front-end parser* for each supported DL model format, using the framework built-in model deserialization APIs. Such a parser is responsible for reading the input DL model from the disk file(s) and reconstructing it to the corresponding computation graph.

DL frameworks may allocate GPU memory in advance before the operator execution (*e.g.*, the CUDA context, initial input tensors, and weight tensors of TensorFlow models). DNNMem defines two allocation policies: `ALLOC_ON_START` (at the initializing phase) and `ALLOC_ON_DEMAND` (at the first use). Before the graph traversal, DNNMem counts tensors and resident buffers with the `ALLOC_ON_START` policy to calculate an *initial* GPU memory consumption.

During the graph traversal, DNNMem calculates a *current* GPU memory consumption for the operator under visiting. As tensors have their lifecycles, DNNMem first computes the set of *unreleased* tensors which are still in GPU memory. This set consists of those alive tensors being dependent by the visiting and subsequent operators. Also, the framework may hold certain dead tensors for a while, therefore they should also be counted. DNNMem defines two release policies: `RELEASE_ON_EXIT` (at the finalizing phase) and `RELEASE_ON_DEATH` (right after being out of use). At present, according to the framework implementations, only operator weights are set to `RELEASE_ON_EXIT` since they will be released only after the training finishes. Thus, unreleased tensors can be captured with the liveness analyzer as well as the release policy information (Section 4.3). Next, DNNMem analyzes tensors being allocated by the visiting operator. Instead of performing program analysis on the source code of operators, DNNMem defines an analytic and framework-independent memory cost function for each operator (Section 4.2), which returns a list of allocated tensors with type and memory size. In this paper, we only consider single-device or data-parallel training in which an operator and its predecessors are placed on the same GPU. Hence, input tensors of the operator are excluded because they are identical to the predecessors' outputs. DNNMem handles the internal tensor fragmentation by padding extra memory according to the alignment requirements. It is possible that several operators may share weights (*i.e.*, aliasing) [36]. DNNMem identifies them from their operator names and counts the shared weight tensors only once.

The GPU memory occupied by the CUDA context is assumed to be constant, being pre-computed by the GPU SKU, framework type and version, etc. DNNMem finally identifies how GPU memory is managed and reserved by the framework runtime, which serves for increasing the performance of memory allocation (Section 4.5).

When the graph traversal completes, the maximum consumption among all operators is reported as the GPU memory consumption of the DL model. Note that our methodology requires that the GPU memory consumption across training iterations is identical. Therefore, the computation graph should be *deterministic* without control-flow operators (*e.g.*, loops, and conditional branches) and dynamic graph changes (*e.g.*, PyTorch employs the define-by-run approach). Otherwise, users may unroll loops (as well as RNNs [50]) statically with a user-specified or framework-default count, or supply multiple deterministic computation graphs (*e.g.*, several model files) to tackle this problem.

4 IMPLEMENTATION

4.1 Estimation on Computation Graph

Formally, the computation graph of a DL model is represented as a directed acyclic graph (DAG):

$$CG = \langle \{u_i\}_{i=1}^{n}, \{(u_i, u_j)\}, \{p_k\}_{k=1}^{m} \rangle$$

Each node u_i is an operator, while a directed edge (u_i, u_j) delivers an output tensor of u_i to u_j as input and specifies the execution dependency between the two operators. Each p_k is a hyperparameter such as input tensor shape, batch size, learning rate, etc. As mentioned before, we suppose that CG is *deterministic* without control-flow operators.

Let $S = \langle u_{i_1}, u_{i_2}, \cdots, u_{i_n} \rangle$ be a topological (linear) ordering extended from the above graph edge ordering such that $u_{i_j} \prec_S u_{i_k} \implies (u_{i_k}, u_{i_j}) \notin CG$. We call S the *operator schedule*, which represents the actual runtime execution of operators. S is pre-generated by reference to the framework implementations [17, 31,

37]. DNNMem then follows S to traverse the computation graph CG sequentially. Suppose that u is the operator under visiting, the current GPU memory consumption for u consists of 3 parts: previously allocated but still in-use tensors, newly allocated tensors for u, and resident buffers of the CUDA context and allocator reservation. The first two kinds of tensors are called the *unreleased* tensors.

Let MF_{init} and MF be the functions that return the initial and current GPU memory consumption. Let MU, MR, and M_{ctx} be the functions that return the memory size of unreleased tensors, memory size of allocator reservation, and GPU memory occupied by the CUDA context, respectively. Function UT computes the set of all unreleased tensors, and MT returns the allocated memory size of a tensor t. Note that MT counts in the internal tensor fragmentation. We use M_{CG} to denote the GPU memory consumption of the computation graph CG, and calculate it as follows:

$$
\begin{aligned}
MF_{init} &= M_{ctx} + \sum MT(t) \quad t \text{ has } \texttt{ALLOC\_ON\_START} \\
MU(u) &= \sum_{t \in UT(u)} MT(t) \\
MF(u) &= MU(u) + MR(u) + M_{ctx} \\
M_{CG} &= \max\{MF_{init}, MF(u_i) \mid u_i \in CG\}
\end{aligned}
$$

The above abstraction and formalization are general to different frameworks and DL models in estimating GPU memory consumption. Users can also adapt the estimation model to new devices and frameworks by using another operator schedule, associating different allocation/release policies to tensors, or modifying the above functions such as MR, M_{ctx}, etc. functions.

4.2 Memory Cost Functions of Operators

Knowing how GPU memory is allocated and used by an operator from its source code is challenging using traditional program analysis techniques. This is because operators are usually implemented by DL frameworks with NVIDIA cuDNN, cuBLAS, or CUDA API invocations (black box) and nested loops.

Instead, we define an analytic and framework-independent memory cost function for each operator by reference to the framework implementations. Our solution is technically feasible for two reasons. First, frequently-used operators are well-defined with clear syntax and semantics. Second, DL frameworks implement them similarly by calling NVIDIA APIs. The memory cost function returns a set of allocated tensors with category and shape (in terms of parameters such as batch size, input tensor shape, the filter number, and so on). Most of the concrete parameter values are fetched from the previously mentioned user specifications, while the input tensor shape can be inferred by Shape Inference.

We suppose that u is the operator under visiting and MC is its memory cost function. Let W, O, and E be the functions that return the sets of u's weight/output/ephemeral tensors. As mentioned in Section 3, we exclude input tensors because only single-device and data-parallel training are considered. Thus,

$$MC(u) = W(u) \cup O(u) \cup E(u)$$

Weight tensors include operator weights (W_m) under forward propagation and weight gradients (W_g) under backward propagation:

$$W(u) = W_m(u) \cup W_g(u)$$

Output tensors consist of forward outputs (O_f) and output gradients (O_g):

$$O(u) = O_f(u) \cup O_g(u)$$

Ephemeral tensors contain three parts:

(1) cuDNN workspace (E_w), which is the additional GPU memory buffer used by cuDNN APIs such as `cudnnConvolutionForward()` in the implementation of framework operators. Larger workspace brings better performance. DNNMem invokes standard interfaces such as `cudnnGetConvolutionForwardWorkspaceSize()` to obtain the amount of cuDNN workspace required. In addition, frameworks may limit the workspace size in case of GPU memory shortage. For example, TensorFlow exports an environment variable `TF_CUDNN_WORKSPACE_LIMIT_IN_MB` to set the upper bound of cuDNN workspace. Thus, DNNMem returns the smaller value.
(2) CUDA data structures (E_v), which are miscellaneous data structures used by CUDA APIs like CUDA random numbers.
(3) Temporary tensors (E_p), which are temporary variables used in the implementation of framework operators. For example, we observe through runtime logs that TensorFlow's convolution operator uses two temporary tensors with the same sizes as the weight and output tensors, respectively.

Thus,

$$E(u) = E_w(u) \cup E_v(u) \cup E_p(u)$$

Note that not all types of tensors are allocated for the operator u.

Let us use the motivating example in Figures 2 and 3 to illustrate how such memory cost functions look like. The following symbols are used to denote each operator's hyperparameters and tensor shapes. S_f is for the precision format of the data type. N represents batch size. H_o, W_o and C_o are output height, width, and channel. H_i, W_i, C_i are input height, width, and channels. H_f and W_f are filter height and width. F_o represents the size of each output sample. Since cuDNN workspace depends on a specific cuDNN convolutional algorithm (denoted by $\mathcal{A}$. *e.g.*, GEMM, FFT, and Winograd), the symbol of the workspace is represented as $E_w^{\mathcal{A}}$.

Table 3 lists all allocated tensors of the operators used in our motivating example and their sizes. Although the developer may specify only three operators in code, DL frameworks automatically insert auxiliary ones into the computation graph for backward propagation. For example, $Conv2d\_BP_1$ and $Conv2d\_BP_2$ are framework-crafted operators for calculating output and weight gradients to update the weights of the developer-specified *Conv2d* operator. The *Linear* (FullyConnected) operator can be implemented by matrix multiplication and addition. The RNN [50] operator needs to consider the weight sharing of stacked cells.

Currently, DNNMem provides memory cost functions for 70+ frequently-used operators. Although operators represent different mathematical operations, they may share the same or similar memory cost functions according to how they manipulate the input data. For example, operators such as *ReLU* and *Sigmoid* perform in-place updates by default (*i.e.*, activation functions). They do not require any additional GPU memory and thus share the same *zero* memory cost function. Another example is that the listed memory cost function of operator *Conv2d* is adapted for *Conv1d* and *Conv3d* with little change needed since their principles are similar. In this way,

Table 3: Allocated tensors and their sizes in the motivating example.

Operator Category	Operator	Tensor Category	Tensor Size
Convolution	*Conv2d*	Weight	$W_m^1 = S_f \times \left(C_i(u) \times H_f(u) \times W_f(u) \times C_o(u) + C_o(u)\right)$
		Forward Output	$O_f^1 = S_f \times N \times C_o(u) \times H_o(u) \times W_o(u)$
		cuDNN Workspace	$E^1 = E_w^{\mathcal{A}(FP)}(u)$
	Conv2d_BP$_1$	Output Gradient	$O_g^7 = Data\_X$
		cuDNN Workspace	$E_1^6 = E_w^{\mathcal{A}(BP_1)}(u)$
	Conv2d_BP$_2$	Weight Gradient	$W_g^6 = W_m^1$
		cuDNN Workspace	$E_2^6 = E_w^{\mathcal{A}(BP_2)}(u)$
AveragePooling	*AvgPool2d*	Forward Output	$O_f^2 = S_f \times N \times C_o(u) \times H_o(u) \times W_o(u)$
	AvgPool2d_BP	Output Gradient	$O_g^6 = O_f^1$
FullyConnected	*Linear*	Weight	$W_m^3 = S_f \times (C_i(u) \times H_i(u) \times W_i(u) \times F_o(u) + F_o(u))$
		Forward Output	$O_f^3 = S_f \times N \times F_o(u)$
	Linear_BP$_1$	Output Gradient	$O_g^5 = O_f^2$
	Linear_BP$_2$	Weight Gradient	$W_g^4 = W_m^3$

Table 4: Operators share the same memory cost functions.

Category	Example Operators
Activation	ReLU, LeakyReLU, Sigmoid, Tanh, ELU
Convolution	Conv1d, Conv2d, Conv3d
Pooling	MaxPooling, AvgPooling
Elementwise	Add, Mul, Mod, And
RNN	VanillaRNN, LSTM, GRU
Constant	DataInput, Constant
Misc	Assert, Ignore

more operators could be supported in DNNMem. Table 4 shows some operators that share the memory cost functions.

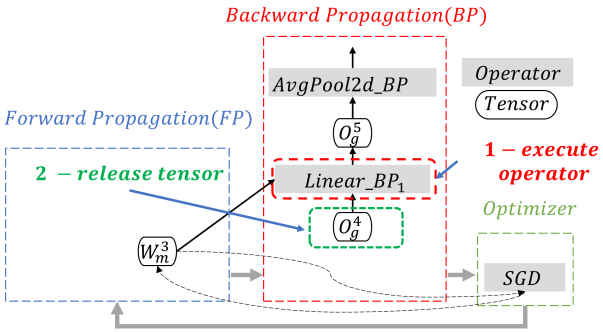

Figure 6: Tensor liveness example.

4.3 Unreleased Tensors

Algorithm 1 demonstrates how to compute the unreleased tensors during graph traversal. We suppose that the computation graph, traversal order, and operator u under visiting are given. First, DNNMem identifies the visited operators on the computation graph and obtains their tensors from the memory cost functions. Next, DNNMem enumerates each of such tensors to check if it has the policy `RELEASE_ON_EXIT` set or it is still *live*. If satisfied, this tensor is added to the set of unreleased tensors. Finally, DNNMem adds all the tensors of u too.

The liveness of a tensor is computed by verifying whether it will be used by any of the current and later operators (*i.e.*, there exists an edge on the computation graph). Figure 6 highlights the dependencies between certain tensors and operators. Suppose that the operator under visiting is *Linear_BP*$_1$, and the immediate successor is *AvgPool2d_BP*. W_m^3 and O_g^4 are used by *Linear_BP*$_1$, so they are live. When we proceed to visit *AvgPool2d_BP*, O_g^4 is then dead assuming that *Linear_BP*$_1$ and *Linear_BP*$_2$ have been visited before. Although the weight tensor W_m^3 looks also dead, it is set `RELEASE_ON_EXIT` thus cannot be released since DL frameworks will keep it in GPU memory for later weight updating.

Since DNNMem is extensible, users can add memory optimization strategies (*e.g.*, SWAP [38] and gradient checkpoint [11]) as extensions to Algorithm 1 to simulate more application scenarios.

4.4 Memory Block Management

As mentioned in Section 3, tracking the memory blocks is indispensable to handle the impact factors of the DL framework runtime (*e.g.*, policies of memory pre-allocation, and reallocation). DNNMem implements a linked-list based memory block manager and the best-fit with coalescing (BFC) algorithm.

When visiting an operator during the computation graph traversal, memory allocation is simulated for each of the operator's tensors. DNNMem searches the list for the first free block fitting the tensor size (with alignment). If such a block is larger than the requested size such that the residual space exceeds a threshold, it is split and the remainder will be inserted into the list right after. Otherwise, the full block should be returned. However, there may be no suitable free blocks anymore. DNNMem then simulates applying for fresh memory from the GPU device by creating a new block data structure and appending it to the list tail. Memory pre-allocation is handled by correctly setting the size of such a new block. For

Algorithm 1: Compute the set of unreleased tensors.

Input: The computation graph cg, traversal ordering tp_order, and operator u under visiting.
Output: A set of unreleased tensors ut.

```
ut ← ∅ ;
prev_tensors ← ∅ ; // Already allocated tensors.
unvisited_ops ← ∅ ; // u will also be included.
foreach op ∈ cg do
    if IsVisited(op) then
        // MC() is the memory cost function.
        prev_tensors ← prev_tensors ∪ MC(op) ;
    else
        foreach t ∈ MC(op) do
            if t.alloc_policy == ALLOC_ON_START then
                prev_tensors ← prev_tensors ∪ { t } ;
            end
        end
        unvisited_ops ← unvisited_ops ∪ { op } ;
    end
end
foreach t ∈ prev_tensors do
    if t.release_policy == RELEASE_ON_EXIT then
        ut ← ut ∪ { t } ; // t cannot be released.
        continue;
    end
    foreach op ∈ unvisited_ops do
        if IsDependent(op, t) then
            ut ← ut ∪ { t } ; // t is alive.
            break;
        end
    end
end
ut ← ut ∪ MC(u) ; // Add tensors of u.
return ut;
```

TensorFlow, the size equals to the total size of all existing memory blocks (exponential backoff).

4.5 Resident Buffers

Resident buffers are essential GPU memory for the training and inference of DL models and are managed by the framework runtime. As shown in Table 2, DNNMem currently handles the following three categories.

CUDA Context. The CUDA context M_{ctx} is mainly determined by three factors: GPU SKU, framework type, and version. When such factors are fixed, it is constant to different DL models. DNNMem profiles values of the CUDA context under various combinations in advance for later queries. The profiling first obtains the total GPU memory consumption using NVML (NVIDIA Management Library) [32], then calculates the consumed memory by DL frameworks from runtime logs, framework APIs, or CUDA hooks, and finally computes the difference.

Internal Tensor Fragmentation. To take maximum advantage of GPU hardware, the actual size of allocated GPU memory for a tensor should meet some alignment requirements. For example, TensorFlow aligns with multiples of 256 bytes while PyTorch aligns with multiples of 512 bytes.

Allocator Reservation. Within the category of allocator reservation, released yet unreclaimed tensors, pre-allocated memory, and external tensor fragmentation can be calculated by querying the memory block manager. For the miscellaneous reservation, DNNMem currently handles the fusion buffer from the data-parallel training using Horovod [40]. DNNMem treats its size as a constant (64 MB by default) and provides a user configuration.

5 EVALUATION

5.1 Experimental Setup

We evaluate DNNMem under three popular DL frameworks: TensorFlow 1.12.0, PyTorch 1.2.0, and MXNet 1.5.0 with CUDA 9.0 and cuDNN 7.0.3. For each framework, we experiment the following 5 representative DL models shown in Table 5.

Table 5: The experimented DL models.

DL Model	Field	Dataset	# of Layers
VGG16	CV	ImageNet [13]	22
ResNet50	CV	ImageNet	50
InceptionV3	CV	ImageNet	48
LSTM	NLP	Synthetic	2
BERT (base)	NLP	GLUE [48]	12

To obtain the real consumed GPU memory of a DL model, we profiled the job using NVIDIA NVML [32]. CUDA Unified Memory [39] was disabled to avoid tensors being migrated to the main memory. We did not limit the memory usage of the cuDNN workspace.

To evaluate the effectiveness of DNNMem, we use *relative error* between the real and estimated GPU memory consumption:

$$\% \text{ error} = \frac{|\text{Est.} - \text{Real}|}{\text{Real}} \times 100$$

Smaller errors indicate better estimation accuracy.

5.2 RQ1: How effective is DNNMem in estimating GPU memory consumption of DL models?

This RQ evaluates the overall effectiveness of DNNMem in estimating GPU memory consumption. Table 6 lists the experimental results for VGG16, ResNet50, InceptionV3 (with the input image data shape [Channel:3, Height:224, Width:224] and batch size 128), and LSTM (with the hidden and input sizes 5120, 2 layers, and batch size 128) models. The results show that DNNMem is able to make satisfactory estimations. For TensorFlow, the relative errors range from 2.3% to 13.8%, with an average of 5.9%. For PyTorch, the relative errors range from 7.5% to 23.0%, with an average of 14.4%. For MXNet, the relative errors range from 0.6% to 10.0%, with an average of 3.9%.

We also compare DNNMem with Shape Inference [25, 34], a static analysis technique to infer the tensor shapes of operator inputs, outputs, and weights. Currently, the three DL frameworks do not provide stand-alone shape inference tools. DNNMem has

Table 6: GPU memory consumption (GB) of different models. "Est." is estimation. "SI" is Shape Inference. "BS128" means batch size 128. The % values denote relative errors.

Models	TensorFlow			PyTorch			MXNet		
BS128	Real	Est.	SI	Real	Est.	SI	Real	Est.	SI
VGG16	17.4	16.9 (2.8%)	7.2 (58.6%)	16.2	14.6 (9.8%)	7.2 (55.5%)	14.3	14.2 (0.6%)	7.2 (49.6%)
ResNet50	8.4	8.2 (2.3%)	5.1 (39.2%)	13.2	10.9 (17.4%)	10.1 (23.4%)	11.8	11.5 (2.5%)	10.1 (14.4%)
Inception V3	10.1	8.7 (13.8%)	7.0 (30.6%)	15.6	12 (23.0%)	11.2 (28.2%)	12.9	11.6 (10.0%)	11.2 (13.1%)
LSTM	4.1	4.3 (4.8%)	2.3 (43.9%)	7.9	8.5 (7.5%)	2.3 (70.8%)	11.9	12.2 (2.5%)	2.3 (80.6%)

already implemented our own using the framework APIs for establishing the operator memory cost functions. We query such cost functions for tensors of the initial input, weights, intermediate outputs, and final output under forward propagation, and then add them up as the GPU memory consumption estimated by Shape Inference.

On average, the relative errors of Shape Inference reach 43.0% (TensorFlow), 44.4% (PyTorch), and 39.4% (MXNet), which are much higher than those of DNNMem. The reason is that DNNMem considers hidden factors such as tensor allocation policy and cuDNN workspace.

To further evaluate DNNMem, for each framework, we experiment with the three DL CV models in Table 5 with 100 different input shapes (from [Channel: 3, Height: 224, Width: 224] to [Channel: 3, Height: 300, Width: 300]) and batch sizes (from 2 to 256). We then compute the mean relative errors (MRE) of all 100 experiments for each framework. Figure 7 summarizes the results. The mean relative errors achieved by DNNMem are 16.0% for TensorFlow, 15.4% for PyTorch, and 16.3% for MXNet. While the mean relative errors achieved by Shape Inference (SI) range from 35.9% to 49.1%. The results show the robustness and effectiveness of DNNMem.

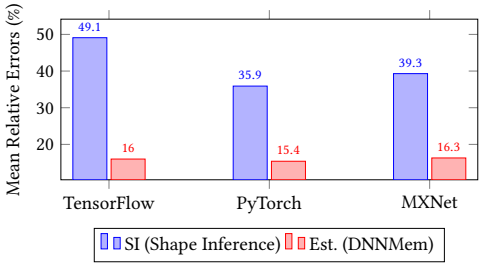

Figure 7: The effectiveness of DNNMem under different input shapes and batch sizes. The Y-axis shows the mean relative errors (%).

To evaluate the effectiveness of DNNMem in predicting the GPU OOM (out-of-memory) cases, we also increase the batch size to 512 and measure the memory consumption of three CV models under the three frameworks (total 9 experiments). Among these 9 experiments, 8 failed due to OOM. That is, the memory consumption is larger than the available memory of NVIDIA Tesla P40 (22.38 GB), which is the GPU used in this experiment. For all the OOM experiments, the memory consumption estimates made by DNNMem range from 28.7 to 46.0 GB, which are all above the available GPU memory (22.38 GB). For the remaining one experiment (TensorFlow ResNet50) that did not have the OOM failure, the estimation error achieved by DNNMem is only 3.9%. The results show that DNNMem can successfully predict OOM cases, confirming the effectiveness of DNNMem.

Table 7: GPU memory consumption (GB) of BERT (base, uncased) model with different batch sizes (BS) and sequence lengths (SL). "SI" is Shape Inference. The % values denote relative errors.

Models	TensorFlow			PyTorch			MXNet		
	Real	Est.	SI	Real	Est.	SI	Real	Est.	SI
BS32 SL32	4.2	3.4 (19.0%)	1.8 (57.1%)	3.5	2.4 (31.4%)	1.8 (48.5%)	3.7	2.9 (21.6%)	1.8 (51.3%)
BS32 SL64	8.2	5.4 (34.1%)	3.1 (62.1%)	4.7	3.8 (19.1%)	3.1 (34.0%)	4.9	4.3 (12.2%)	3.1 (36.7%)
BS128 SL64	16.2	15.4 (4.9%)	11.2 (30.8%)	12.7	12.3 (3.1%)	11.2 (11.8%)	11.9	13.1 (10.0%)	11.2 (5.8%)
BS64 SL128	16.2	15.4 (4.9%)	11.2 (30.8%)	12.6	12.3 (2.3%)	11.2 (11.1%)	13.1	13.1 (0.0%)	11.2 (14.5%)
BS100 SL128	21.2	22.7 (7.0%)	17.3 (18.3%)	18.4	18.6 (1.0%)	17.3 (5.9%)	20.5	19.6 (4.3%)	17.3 (15.6%)

Table 7 shows the experiment of BERT [14] (base) model over the GLUE (General Language Understanding Evaluation) benchmark [48], with various batch sizes and sequence lengths. DNNMem achieves average errors of 13.9% (TensorFlow), 11.3% (PyTorch), and 9.6% (MXNet) and Shape Inference achieves average errors of 39.8% (TensorFlow), 22.2% (PyTorch), and 24.7% (MXNet). The results show that DNNMem is still effective under different hyperparameters.

Table 8: Categories of GPU memory consumption (GB) of TensorFlow VGG16 model. The % values are relative errors.

Batch Size	64		128		256	
Category	Real	Est.	Real	Est.	Real	Est.
Live Tensors	4.90	3.52	8.55	7.24	15.84	14.49
Internal Fragmentation	0.14	0.02	0.27	0.04	0.13	0.04
Allocator Reservation	3.08	5.08	8.3	9.34	5.16	2.59
CUDA Context	0.37	0.37	0.37	0.37	0.37	0.37
Total	8.49	8.99 (5.88%)	17.49	16.99 (2.85%)	21.50	17.49 (18.65%)

An advantage of the analytic approach is the interpretability that DNNMem can present memory usage details, which will greatly help developers tune model configurations and framework runtime parameters. Table 8 demonstrates how GPU memory is consumed by different categories of tensors and TensorFlow runtime when training the VGG16 model. The real memory consumption of each part was obtained from TensorFlow runtime logs. "Live Tensors" refer to the Weight/In/Out/Ephemeral tensors in Table 2. DNNMem achieves low average errors of 5.88 % (Total Consumption), 17.33 % (Live Tensors), 42.42% (Allocator Reservation), and 0.0% (CUDA Context). Because the internal fragmentation has a relatively small value, the estimation can cause a much higher average error (80.04%). Nevertheless, it contributes only a very small portion of the total GPU memory consumption.

As for the time performance, the estimation time of DNNMem ranges from 0.6 to 0.7 seconds for the above experiments. DNNMem has an order of magnitude speedup compared with real execution estimation.

5.3 RQ2: How accurate are the operator memory cost functions of DNNMem?

Operators' memory cost functions play a critical role in DNNMem. This RQ is to evaluate their accuracy. Four representative operators: *Conv2d*, *AvgPool2d*, *Dropout*, and *BatchNorm* (batch normalization) were chosen for experiment. We crafted a minimal DL model for each of them to reduce distractions. For example, the *Conv2d* model only adds an additional *Linear* operator. To obtain the real memory usage, we analyzed the runtime logs of TensorFlow and MXNet. For PyTorch, we added profiling code right before and after operator construction/execution inside the framework. The shape of the input data is [BatchSize:128, Channel:3, Height:224, Width:224]. *Conv2d* has the `filter_count` of 2 and `kernel_size` of 3. For *AvgPool2d*, its `kernel_size` and `stride` are both 2.

Figure 8 shows that the estimation errors of the four operators are all less than 8%, indicating that our memory cost functions are accurate. Note that the values of TensorFlow *BatchNorm* are marked as 0 because the in-place execution is enabled by default.

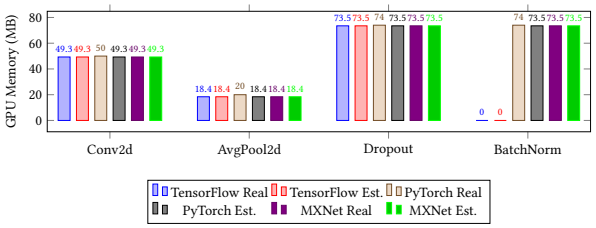

Figure 8: GPU memory consumption (MB) of DL operators.

5.4 RQ3: How effective is DNNMem in data-parallel training?

Nowadays, in industrial practice, many DL training jobs adopt data parallelism, which employs multiple GPU devices (in a single machine or distributed nodes) to increase the number of input data processed simultaneously. This RQ is to evaluate the effectiveness of DNNMem in such common scenarios. We experiment the ResNet50 model with a batch size of 64 using Horovod (a popular data-parallel training framework supporting automatic parallelization [40]). The fusion buffer has a default size of 64 MB. Note that here the TensorFlow model is provided by Horovod using Keras APIs, which is different from that in Section 5.2. We ran the multi-device experiments on a single node and ran the distributed experiments on a 3-node cluster. Each node is equipped with 4 NVIDIA K80 GPUs with 12GB memory each. The reported real GPU memory consumption is the arithmetic mean value of all training instances.

Figure 9 shows that DNNMem achieves average errors of 11.8% (TensorFlow), 13.85% (PyTorch), and 8.9% (MXNet), indicating the effectiveness of DNNMem in data-parallel training.

6 RELATED WORK

Quality attributes (*e.g.*, reliability, cost, performance, and memory consumption) are non-functional properties of software, which are vital for the success of a real-world software-intensive system. Over the years, many estimation models have been proposed to predict these attributes. Examples include defect prediction [24, 27], effort and cost estimation [28, 44], and performance prediction [16, 41].

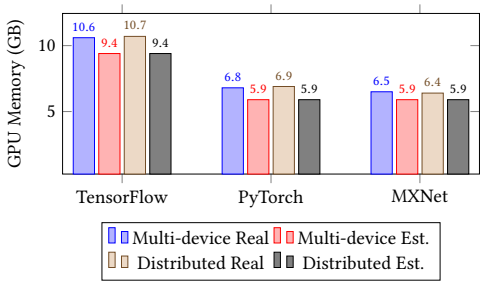

Figure 9: The effectiveness of DNNMem in data-parallel training (ResNet50).

There are also many program analysis techniques [2, 6, 7, 12, 22, 46, 47] for memory footprint analysis and estimation. For example, Verbauwhede *et al.* [47] propose to estimate the memory of DSP applications by modeling array dependencies and execution sequence as an integer linear problem solved by the ILP solver. Albert *et al.* [2] present parametric inference on the notion of object lifetime to inferring memory requirements of Java-like programs. Heo *et al.* [19] propose a resource-aware flow-sensitive analysis via online abstraction coarsening. However, as described in the paper, they cannot be directly applied to deep learning programs.

Frameworks' built-in Shape Inference [17, 25, 29, 34], and some DL performance analysis work [8] estimate GPU memory usage by summarizing weight, input, and output tensors on the computation graph under forward propagation. However, they are just a subset of the whole memory consumption. Shape Inference is incapable of analyzing the remaining yet complex memory usage by tensors under backward propagation and framework runtime (*e.g.*, memory fragmentation/reallocation/reservation, cuDNN workspace), which could observably affect the final GPU memory consumption. DNNMem adopts a novel, comprehensive, and unified analytic approach which systematically solves the challenges. We have compared DNNMem with Shape Inference in Section 5, and the results indicate that DNNMem is more effective and robust.

Real execution estimation [30] has issues of being limited by the memory capacity of testing GPUs, high execution cost, and environmental dependency, which are especially not applicable to enterprise platforms. DL compilers such as TVM [10] focus on the inference phase, cross-platform deployment, and loop level cost model. However, these techniques are beyond the scope of this paper. Researchers have also observed the need for memory cost modeling for DNN memory optimization and planning by analyzing the computation graph [26, 38, 49]. Unlike these work, DNNMem focuses on memory estimation for DL models.

7 CONCLUSION

In this paper, we have presented DNNMem, an accurate estimation tool for GPU memory consumption of deep learning models. This work is motivated by the many out-of-memory failures of DL jobs in Microsoft. DNNMem adopts an analytic approach that systematically explores many memory consumption-related factors. Our extensive experiments show that DNNMem can make satisfactory estimations of GPU memory consumption. DNNMem is also effective and robust to the choices of neural architectures, hyperparameters, and frameworks.

While we use models developed under three popular deep learning frameworks to evaluate the proposed approach, DNNMem is generalizable. We can define more memory cost functions of standard/custom operators and adapt the analytic approach to different devices and frameworks. In the future, we will experiment with the extension of DNNMem to demonstrate its generalizability.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, and Andy et.al Davis. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2010. Parametric Inference of Memory Requirements for Garbage Collected Languages. *SIGPLAN Not.* 45, 8 (June 2010), 121–130. https://doi.org/10.1145/1837855.1806671

[3] S. Amari. 1993. Backpropagation and stochastic gradient descent method. *Neurocomputing, 5(4):185 - 196* (1993).

[4] Amazon. 2019. Amazon SageMaker. https://aws.amazon.com/sagemaker.

[5] Microsoft Azure. 2019. Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning-service.

[6] Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, and Gilles Muller. 2016. Understanding the Memory Consumption of the MiBench Embedded Benchmark. In *Networked Systems*, Parosh Aziz Abdulla and Carole Delporte-Gallet (Eds.). Springer International Publishing, Cham, 71–86.

[7] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. 2008. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the 7th International Symposium on Memory Management* (Tucson, AZ, USA) *(ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 141–150.

[8] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2017. An Analysis of Deep Neural Network Models for Practical Applications. *ArXiv* abs/1605.07678 (2017).

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174

[12] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. 2016. Symbolic Execution for Memory Consumption Analysis. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems* (Santa Barbara, CA, USA) *(LCTES 2016)*. ACM, New York, NY, USA, 62–71.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR 2009*.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*.

[15] Peng Gu. 2018. Memory management for tensorflow. https://github.com/miglopst/cs263_spring2018/wiki/Memory-management-for-tensorflow

[16] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1095–1106. https://doi.org/10.1109/ICSE.2019.00113

[17] Mark Harris. 2019. TensorFlow Graph Optimizations. (2019).

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385

[19] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-Aware Program Analysis via Online Abstraction Coarsening. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 94–104. https://doi.org/10.1109/ICSE.2019.00027

[20] Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80.

[21] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, NY, USA, 510–520.

[22] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 774–784. https://doi.org/10.1145/3338906.3338936

[23] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*. http://arxiv.org/abs/1412.6980

[24] Z. Li, X. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying. 2019. On the Multiple Sources and Privacy Preservation Issues for Heterogeneous Defect Prediction. *IEEE Transactions on Software Engineering* 45, 4 (2019), 391–411.

[25] Malmaud. 2020. TensorFlow Shape Infer. https://malmaud.github.io/tfdocs/shape_inference.

[26] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training Deeper Models by GPU Memory Optimization on TensorFlow.

[27] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (1 12 2010), 375–407.

[28] K. Molokken and M. Jorgensen. 2003. A review of software surveys on software effort estimation. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* 223–230.

[29] MXNet. 2020. MXNet Memory Monger. https://github.com/dmlc/mxnet-memonger.

[30] MXNet. 2020. MXNet symbol simple bind. https://beta.mxnet.io/api/symbol/_autogen/mxnet.symbol.Symbol.\simple_bind.html.

[31] Apache MXNet. 2019. The topological sorting algorithm for computation graphs in Apache MXNet. https://github.com/apache/incubator-mxnet/blob/1.6.0/src/executor/simple_partition_pass.h

[32] NVIDIA. 2019. NVML API Reference Guide. https://docs.nvidia.com/deploy/nvml-api/index.html. (2019).

[33] Nvidia. 2020. cudnnConvolutionFwdAlgo. https://docs.nvidia.com/deeplearning/sdk/cudnn-api/index.html#cudnnConvolutionFwdAlgo_t.

[34] ONNX. 2020. ONNX Shape Inference. https://github.com/onnx/onnx/blob/v1.7.0/docs/ShapeInference.md.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, and James et al. Bradbury. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035.

[36] PyTorch. 2019. PyTorch: Control Flow + Weight Sharing. https://pytorch.org/tutorials/beginner/examples_nn/dynamic_net.html.

[37] PyTorch. 2019. The topological sorting algorithm for computation graphs in PyTorch. https://github.com/pytorch/pytorch/blob/v1.2.0/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h#L26

[38] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 18, 13 pages.

[39] Nikolay Sakharnykh. 2018. Everything you need to know about unified memory. *NVIDIA GTC* (2018).

[40] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799

[41] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 284–294.

[42] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.1556

[43] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). arXiv:1512.00567 http://arxiv.org/abs/1512.00567

[44] Hee Beng Kuan Tan, Yuan Zhao, and Hongyu Zhang. 2009. Conceptual Data Model-Based Software Size Estimation for Information Systems. *ACM Trans. Softw. Eng. Methodol.* 19, 2, Article 4 (Oct. 2009), 37 pages.

[45] T. Tieleman and G. Hinton. 2012. rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude. *COURSERA: Neural Networks for Machine Learning, 4, 26-31.* (2012).

[46] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. 2000. *Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages.* Technical Report. Indiana University.

[47] Ingrid M. Verbauwhede, Chris J. Scheers, and Jan M. Rabaey. 1994. Memory Estimation for High Level Synthesis. In *Proceedings of the 31st Annual Design Automation Conference* (San Diego, California, USA) *(DAC '94)*. Association for Computing Machinery, New York, NY, USA, 143–148.

[48] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In the Proceedings of ICLR.

[49] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18). 41–53.

[50] P. J. Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[51] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, Republic of Korea) *(ICSE '20)*. Association for Computing Machinery, NY, USA, 1159–1170.