`MODeL`: Memory Optimizations for Deep Learning

Benoit, Steiner [1] **Mostafa, Elhoushi** [2] **Jacob, Kahn** [2] **James, Hegarty** [3]

Abstract

The size of deep neural networks has grown exponentially in recent years. Unfortunately, hardware devices have not kept pace with the rapidly increasing memory requirements. To cope with this, researchers have proposed various techniques including spilling, recomputation, reduced precision training, model pruning, and so on. However, these approaches suffer from various limitations: they can increase training time, affect model accuracy, or require extensive manual modifications to the neural networks.

We present `MODeL`, an algorithm that optimizes the lifetime and memory location of the tensors used to train neural networks. Our method automatically reduces the memory usage of existing neural networks without any of the drawbacks of other techniques.

We formulate the problem as a joint integer linear program (ILP). We present several techniques to simplify the encoding of the problem, and enable our approach to scale to the size of state-of-the-art neural networks using an off-the-shelf ILP solver. We experimentally demonstrate that `MODeL` only takes seconds to allow the training of neural networks using 30% less memory on average.

`MODeL` is an open-source project available at https://github.com/facebookresearch/model_opt.

1. Introduction

Scale is a major force behind the accuracy improvements of machine-learning-based solutions (Bubeck & Sellke, 2021), and both the depth and width of deep neural networks (DNN) are expanding exponentially (Sevilla et al., 2022) (Figure 1). This inflation in size increases the memory needed to store the weights of the neural network and the intermediate results (e.g., activations and gradients) generated during the training process. Compounding the problem, researchers are training neural networks on larger inputs, such as high-resolution images (Dong et al., 2016; Tai et al., 2017), video (Feichtenhofer et al., 2019), three dimensional point-clouds (Chen et al., 2017), long natural language sequences (Vaswani et al., 2017; Child et al., 2019; Devlin et al., 2018), and using larger batch sizes to increase efficiency (Smith et al., 2018).

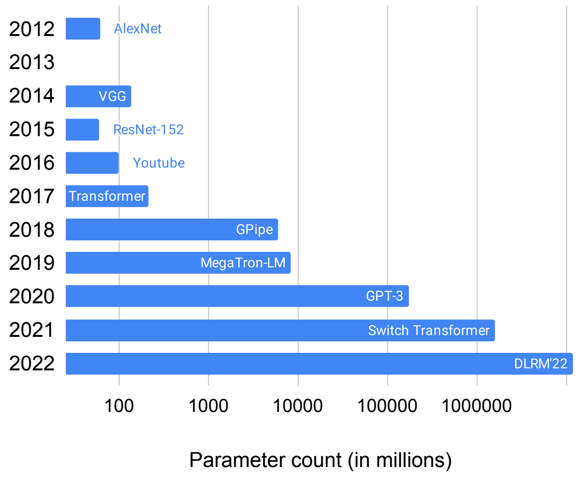


Figure 1. The number of deep neural network parameters has increased by 100,000 fold over the last 10 years, starting to grow exponentially around 2016. The x-axis is plotted on a log scale.

Unfortunately, due to the slowing of Moore's law, the memory capacity of hardware has only increased linearly over the last decade (Figure 2). Thus, the amount of memory available on the hardware used to train DNNs has not kept pace with the needs of deep learning. Furthermore, features powered by machine learning, such as automatic speech recognition (Paulik et al., 2021) or keyboard suggestions (Hard et al., 2018), are being personalized by fine tuning models on-device. This means that model training is increasingly being pushed to even more memory constrained edge devices such as smartphones. As a result, memory is increasingly becoming a bottleneck that hinders progress, and researchers frequently mention memory scarcity as a limiting factor that impacts their work (Krizhevsky et al., 2012; He et al., 2016;

[1]Anthropic, San Francisco, USA [2]Meta, FAIR, Menlo Park, USA [3]Meta, Reality Labs, Seattle, USA. Correspondence to: Benoit Steiner <benoit@anthropic.com>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

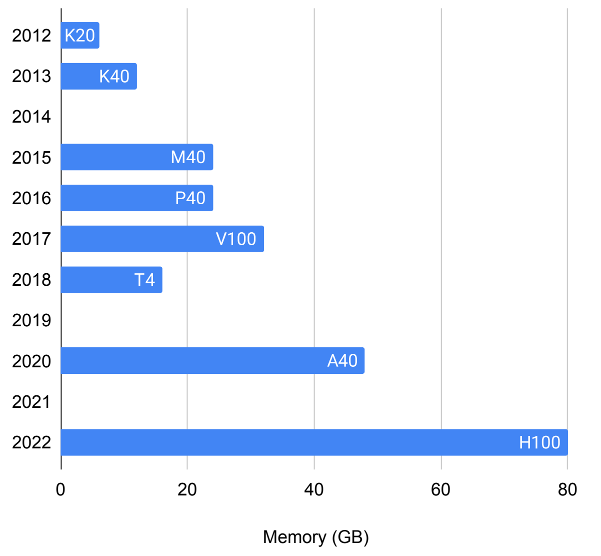


Figure 2. The memory capacity of NVidia datacenter GPUs (in gigabytes) has only increased tenfold over the last decade, which has not kept pace with the rapidly increasing size of deep neural networks. The x-axis is plotted on a linear scale.

Chen et al., 2015; Dai et al., 2019; Child et al., 2019).

Popular deep learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) do not fully utilize the limited memory available. Similar to traditional dynamic memory allocators such as tcmalloc (Google) and jemalloc (Evans), these frameworks maintain a pool of free blocks of memory at runtime. To serve memory requests, they look for a large enough memory block in the memory pool, or allocate it from the physical memory if none is available. This results in memory fragmentation when free memory blocks do not exactly match the size of an allocation request, which occurs frequently.

Furthermore, DNN frameworks do not optimize tensor lifetimes. PyTorch (Paszke et al., 2019) executes operations in the order in which they are defined in the program. TensorFlow (Abadi et al., 2016) keeps a queue of operators that are ready to run, and executes them on a first-come, first-served basis. As a result, tensors can be allocated earlier than required, or freed later than necessary, wasting valuable memory.

Our method overcomes these two limitations of existing deep learning frameworks. We model the computations performed to train a deep neural network as a dataflow graph of operations. We analyze this graph to find a topological ordering of the nodes that adjusts the lifetime of the tensors generated by these operations to minimize the peak amount of memory that needs to be allocated (Figure 3). Furthermore, we find an optimal packing of these tensors, which minimizes memory fragmentation (Figure 4). We encode these two objectives as an integer linear program (ILP) that can be solved quickly by commodity solvers, and present MODeL (Memory Optimizations for Deep Learning), our algorithm for memory-optimal training of neural networks.

In addition to significantly reducing memory usage, our solution has several key strengths. First, it does not impact the accuracy of the predictions of the neural networks. Second, it requires no modification to the neural network or the training procedure. Third, it doesn't increase training time. Fourth, it is orthogonal to and can be combined with other memory reductions techniques, such as the ones listed in section 2, to further reduce the memory needs of a neural network.

Our work makes the following novel contributions:

- We formulate the problem of finding the lifetime and memory location of tensors that minimizes the peak memory required to train neural networks as a joint integer linear program.
- We demonstrate how to leverage domain knowledge to simplify the ILP formulation, which enables off-the-shelf solvers to quickly reduce the memory usage of large DNNs.
- We study empirically the practicality and effectiveness of our solution on a wide variety of DNNs, which achieves average memory savings exceeding 30% in a median time of less than 7 seconds.
- We provide an open source implementation of MODeL at https://github.com/facebookresearch/model_opt.

2. Related Work

Various approaches, complementary to ours, have been proposed to break the "memory wall" and train larger networks. The first technique distributes the computation for a single forward-backward iteration over several hardware devices, thus making more memory available overall. However, this approach, known as model parallelism (Karakus et al., 2021), significantly increases the financial cost of training deep neural networks since it requires access to additional expensive compute accelerators and fast networks. Furthermore, partitioning a deep neural network efficiently to balance communication and computation remains an open problem still actively researched (Gholami et al., 2018; Jia et al., 2018; Mirhoseini et al., 2018).

In parallel, the research community has developed numerous solutions to reduce the memory footprint of neural networks:

- Novel neural network architectures reduce the number of parameters needed to achieve a given level of accuracy (Iandola et al., 2017; Tan & Le, 2019). Furthermore, automated search techniques known as neural architecture search (Tan et al., 2019) have been proposed to automatically design memory efficient models. The main drawbacks of these methods are that they are time consuming to deploy, and fail to match the result quality of state of the art DNNs.

- Model compression methods (Blalock et al., 2020) prune (Louizos et al., 2018; Frankle & Carbin, 2018; Molchanov et al., 2019; Elkerdawy et al., 2022; He et al., 2020) or share (Dehghani et al., 2019) weights to improve the efficiency of the model parameterization. However, the majority of these techniques require training the unpruned neural network first, and are therefore most useful for inference.
- Training using reduced precision arithmetic on 16-bit floating point or even quantized representations (Wang et al., 2018; Zhu et al., 2020; Kalamkar et al., 2019) significantly reduces memory (Fan et al., 2020; Lin et al., 2016). However, these techniques can compromise the accuracy of the neural networks, make training unstable, and require careful implementation to be deployed successfully (NVidia; Lin & Talathi, 2016).

Several efforts have looked at the problem from a systems perspective, and presented solutions to reduce pressure on the memory subsystem. These techniques encompass:

- In-memory tensor compression, which can result in minimal accuracy loss in many DNN applications (Chen et al., 2021; Jain et al., 2018). However, this comes with a runtime penalty, since the data must be compressed and uncompressed on the fly.
- Rematerialization, also known as checkpointing, discards activations in the forward pass to save memory, and recomputes those values as needed when computing the gradients. Numerous strategies to identify which activations to discard have been proposed (Jain et al., 2020; Zheng et al., 2020; Chen et al., 2016; Griewank & Walther, 2000; Shah et al., 2021). While effective at reducing memory usage, these techniques add extra computations, which increases the training time.
- Paging, aka spilling, consists of moving data between a small but high bandwidth and low latency memory pool, and a large but slow external memory. This has been demonstrated to effectively offload the data stored on a GPU device onto the host memory (Peng et al., 2020; Hildebrand et al., 2020; Meng et al., 2017), but again increases training time due to extra memory transfers.
- More recently, combining several of these techniques has been proposed to increase their effectiveness and mitigate their drawbacks (Beaumont et al., 2021; Patil et al., 2022) without fully eliminating them.

Additionally, some techniques developed primarily to increase execution speed are also beneficial for memory:

- Operator fusion can reduce memory footprint by avoiding the need to materialize large intermediate buffers and keep them around for backpropagation (Niu et al., 2021).
- Machine learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) allow some of their operators to store the data they generate in one of their input tensors, thus avoiding the need to allocate an output tensor. This is known as in-place-update, and saves memory. However, users must manually modify their neural networks to leverage this capability, and it can lead to correctness issues if used indiscriminately (Paszke et al., 2017).

Optimizing the location of tensors in memory to reduce fragmentation, also known as the dynamic storage allocation problem, is NP-hard (Garey & Johnson., 1979). This problem has been studied in the context of deep learning by other researchers (Sekiyama et al., 2018) who proposed an exact formulation to minimize the memory fragmentation of deep neural networks. However, their approach scaled poorly and only succeeded in optimizing two small neural networks in inference mode. As a result, they ultimately advocated for a heuristics based approach.

Improving the lifetime of tensors has also been studied before. Liberis *et al.* (Liberis & Lane, 2020) and Serenity (Ahn et al., 2020) looked for a memory-optimal execution schedule by enumerating the topological orders of the DNN graph and calculating their peak memory usage. To speed things up, they both proposed dynamic programming based optimizations to prune the number of orderings they needed to consider. However, the complexity of their algorithms remains prohibitive at $O(|V| * 2^{|V|})$ in both cases, and they only managed to make them work for inference on tiny graphs. Lin *et al.* (Lin et al., 2022) also mentioned reordering computations as a way to enable operator fusion and reduce the peak memory footprint while training. Unfortunately, they didn't describe the algorithm they used to find a suitable node ordering.

3. Background

3.1. Representing Neural Networks as Dataflow Graphs

Deep neural networks can be represented using dataflow graphs, as pioneered by TensorFlow (Abadi et al., 2016).

The nodes of the graph encode the computations to be performed (e.g. matrix multiplications, convolutions, activation functions), while the edges represent the data (*aka* tensor or array) that is produced by an operation and transferred to consumer nodes.

Due to the producer-consumer relation between connected nodes, edges are oriented. Each edge has exactly one source, which is the operator that generated the corresponding tensor. Since a tensor can be consumed by more than one node, edges can have multiple sinks.

Operators can have multiple incoming (*aka* fanin) edges. Typically, one of these incoming edges will be the tensor generated by the previous layer, and another one will be a weight tensor. Similarly, operators can have multiple outgoing (*aka* fanout) edges: while most operations generate a single output tensor, some may create two or more. Operators with no fanout edges are used to model the final outputs of the neural network. Operators without fanin edges can model random number generators, constants, weights, or initial inputs to the neural network.

In the remainder of this paper, we assume that the graphs are acyclic. In practice, this is not a significant limitation since recurrent neural networks such as LSTM (Hochreiter & Schmidhuber, 1997) have been eclipsed by transformers (Vaswani et al., 2017). Furthermore, their loops can be unrolled to avoid the problem altogether.

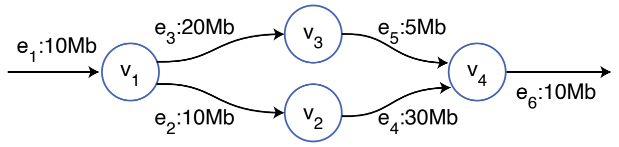
3.2. Optimizing Tensor Lifetimes

For an operator to run, all its input tensors must be resident in memory, and its output tensors must have been allocated so that they can be written to while the node executes. Additionally, to avoid recomputing tensors, once a tensor is generated it must be preserved in memory until all its consumers have been run.

We define the resident set $RS(s)$ at a given step s in the execution of a neural network as the set of tensors that need to be kept in memory at that point in time. It comprises the tensors in the fanin and fanout of the operator that is scheduled for execution at step s, as well as all the other tensors that were previously generated but need to be kept in memory to be able to run subsequent operators. The peak resident set is the largest resident set over the execution of the network.

The order in which nodes are executed impacts the lifetime of the tensors, and therefore the peak working set. Figure 3 illustrates a simple example in which changing the operator ordering noticeably improves memory usage.

Among all possible node orderings, those prioritizing the execution of nodes that free large amounts of data while generating little output data themselves, are likely to be more efficient. However, as demonstrated in prior works (Bernstein et al., 1989; Bruno & Sethi, 1976), finding an optimal scheduling for a generic DAG is an NP-complete problem, which cannot be solved with a simple greedy approach.



Resident Sets:

Order #1, Peak=60Mb			Order #2, Peak=45Mb		
v_1	e_1, e_2, e_3	40Mb	v_1	e_1, e_2, e_3	40Mb
v_2	e_2, e_3, e_4	60Mb	v_3	e_2, e_3, e_5	35Mb
v_3	e_3, e_4, e_5	55Mb	v_2	e_2, e_4, e_5	45Mb
v_4	e_4, e_5, e_6	45Mb	v_4	e_4, e_5, e_6	45Mb

Figure 3. Node execution orders can impact peak memory usage. Edges are annotated with the size of their corresponding tensors, and the two feasible node orders are annotated with the set of tensors resident in memory at each step. Running v_3 before v_2 is significantly more memory efficient.

3.3. Optimizing Tensor Locations in Memory

Similar to malloc-style memory allocators, the tensor allocation schemes used by typical deep learning frameworks operate online and suffers from fragmentation. Indeed, free memory is often segregated into small blocks and interspersed by memory allocated to live tensors. As a result, a significant fraction of the total memory is effectively unusable because it is divided into pieces that are not large enough to fit a tensor. Figure 4 illustrates this phenomenon and demonstrates how planning the location of each tensor ahead of time can significantly reduce the overall peak memory usage.

4. Formulation

We propose to take advantage of the predictability of neural network computations to proactively optimize the lifetime and location of tensors in memory.

We formulate the problem of optimizing the ordering of computations (which determines the tensor lifetimes) and the location of tensors in memory (which determines the amount of memory fragmentation) of generic data-flow graphs, including those used in neural network training. We encode the problem as an integer linear program (Wikipedia, 2023) and use an off-the-shelf ILP solver to find a solution that minimizes the peak memory required to run the dataflow graph.

We solve the ILP problem ahead of time, before the training

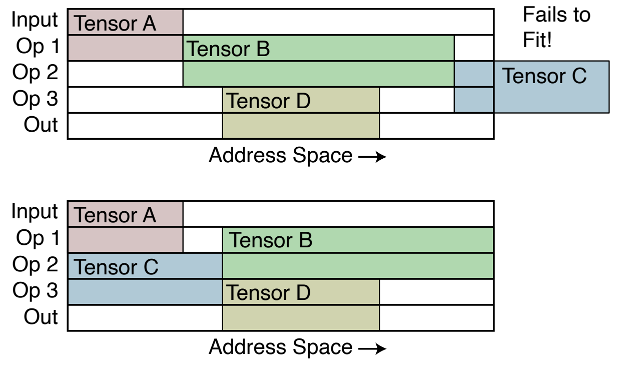


Figure 4. Memory fragmentation can significantly increase the memory needed to store tensors. A greedy allocator (top) would not leave any room between tensors A and B, thus making it impossible to reuse the space left once tnsor A is freed to store tensor C. MODeL (bottom) leaves a gap between tensor A and B to enable the reuse of the memory freed by tensor A and fits all the tensors in less memory.

process starts. This results in a small one-time initial cost, which is negligible compared to the time it takes to train a neural network (see section 5.3).

4.1. DNN representation

As mentioned in section 3.1, we model a neural network as a directed acyclic graph G = (V, E) with n nodes $V = v_1, ..., v_n$ that represent the operators and the neural network, and m edges $E = e_1, ..., e_m$ that encode the tensors exchanged by operators. The size in bytes of the tensor represented by edge e_i is denoted as S_i. The source vertex of edge e is denoted $src(e)$. The set of sink vertices of edge e is denoted $snks(e)$.

The set of edges in the fanout of a node v is denoted $fo(v)$, while the set of edges in its fanin is represented as $fi(v)$. We will also denote $fi(e)$ the set of edges in the fanin of the source vertex of e. We represent by $sib(e)$ the siblings to an edge e, that is the collection of edges that are driven by the same source vertex.

We model the execution of a neural network as a sequence of discrete steps $S = s_1, ..., s_n$. At least one operator is executed at each step, and therefore, we need at most n steps to schedule a graph of n operators.

4.2. Encoding Tensor Lifetimes

We track which tensors are allocated and which tensors are preserved in memory at each execution step. To do this, we use two sets of binary variables:

- A variable labeled $C_{e,\,s} \in \{0, 1\}$ indicates whether or not the tensor e should be created (i.e. allocated) at step s by running its source vertex.
- A variable named $P_{e,\,s} \in \{0, 1\}$ reflects whether tensor e needs to be preserved in memory at step s or whether it can be freed.

We leverage a set of linear constraints to ensure that the sequence of tensor creations and preservations reflects a valid execution sequence of the neural network corresponding to a feasible topological ordering of the DAG.

First, a tensor e can either be created or preserved at each step s, but not both (equation 1). Note that it's possible for both $C_{e,\,s}$ and $P_{e,\,s}$ to be false, which indicates that the tensor does not reside in memory at this point in time.

$$\forall e \in E,\, \forall s \in S \quad C_{e,\,s} + P_{e,\,s} \leq 1 \tag{1}$$

Second, a tensor e can be preserved in memory at step s if and only if it was created or preserved at the previous step (equation 2).

$$\forall e \in E,\, \forall s \in S \quad P_{e,\,s} \leq P_{e,\,s-1} + C_{e,\,s-1} \tag{2}$$

Third, to ensure that the solver does not simply avoid running any of the operators, we force every tensor e to be created once through equation 3.

$$\forall e \in E \quad \sum_{s \in S} C_{e,\,s} = 1 \tag{3}$$

Fourth, a tensor e can only be created by running its source operator v. In order to do so, all the tensors in the fanin of v must be present in memory (equation 4).

$$\forall e \in E,\, \forall s \in S,\;\; \forall f \in fi(e) \quad C_{e,\,s} \leq P_{f,\,s} \tag{4}$$

Last but not least, we also need to make sure that operators with multiple outputs create their output tensors at the same time. We achieve this by tying the values of the $C_{f,s}$ variables for all the siblings f to a tensor e in equation 5.

$$\forall e \in E,\, \forall s \in S,\;\; \forall f \in sib(e) \quad C_{f,\,s} = C_{e,\,s} \tag{5}$$

The combination of constraints 1 through 5 ensures that all the feasible solutions to the ILP correspond to valid schedules. They guarantee that the creation step of each tensor corresponds to a topologically feasible ordering of the vertices of the graph. Moreover, they force the preservation in memory of each tensor from the time it is generated until the last step in which it is consumed.

4.3. Encoding Tensor Locations

To let our solver also optimize the placement of tensors in memory, we assign an integer variable $A_e \in [0, M]$ to each tensor e that encodes its base address. Here, $M = \sum_e S_e$, which corresponds to the worst case scenario where all the tensors reside concurrently in memory.

We also introduce two binary variables $a_{i,j} \in \{0, 1\}$ and $b_{i,j} \in \{0, 1\}$ for each pair of tensors i and j. We constrain them through equation 6 in such a way that either $a_{i,j}$ or $b_{i,j}$ is equal to 1 if both tensors reside in memory concurrently at any point in time, but can be 0 otherwise.

$$\begin{aligned}\forall (i,j) \in E^2 \quad & a_{i,j} + b_{i,j} \leq 1 \\ & a_{i,j} + b_{i,j} \geq live_{i,t} + live_{j,t} - 1 \\ where\ live_{i,t} &= C_{i,t} + P_{i,t} \\ and\ live_{j,t} &= C_{j,t} + P_{j,t}\end{aligned} \tag{6}$$

We use these variables to prevent the overlap of tensors that reside in memory at the same time in equations 7a and 7b.

$$\forall (i,j) \in E^2 \quad A_i + S_i - A_j \leq (1 - a_{i,j}) * M \tag{7a}$$

$$\forall (i,j) \in E^2 \quad A_i - A_j - S_j \geq (b_{i,j} - 1) * M \tag{7b}$$

If $a_{i,j}$ takes the value 1, equation 7a degenerates into $A_i + S_i \leq A_j$. This forces tensor i to reside below tensor j in memory. Similarly, equation 7b degenerates into $A_i \geq A_j + S_j$ when $b_{i,j}$ takes the value 1, which forces tensor i to be placed above tensor j. On the other hand, if $a_{i,j}$ and $b_{i,j}$ take the value 0, equations 7a and 7b hold for any value of A_i and A_j in the range $[0, M]$. In other words, they don't impose further restrictions on the location of e_i and e_j.

Put altogether, constraints 6, 7a, and 7b ensure that tensors can share the same memory space if and only if their lifetimes do not overlap.

4.4. Minimizing Peak Memory Usage

We track the peak memory usage by introducing a variable $peak\_mem$ that we constrain as follow:

$$\forall e \in E \quad A_e + S_e \leq peak\_mem \tag{8}$$

We find the schedule of operators and memory location of tensors that optimizes the memory usage of the neural network by feeding program 9 to an ILP solver.

$$\begin{aligned} & \arg\min_{C,P,A} peak\_mem \\ & subject\ to\ (1), (2), (3), (4), (5), \\ & (6), (7a), (7b), (8)\end{aligned} \tag{9}$$

4.5. Decoding the ILP Result

Given a feasible solution to our ILP, we generate an optimized execution sequence of operations $ES = (v_1, ..., v_n)$ for the neural network using function 1.

Function 1 GenerateExecutionSequence(C)

```
▷ Converts the output of the ILP into an optimized
▷ execution sequence of operations $seq$.
$seq$ = []
for $s$ in $S$ do
    for $e$ in $E$ do
        if $C_{e,s}$ = 1 and $src(e)$ not in seq then
            add $src(e)$ to $seq$
        end if
    end for
end for
return $seq$
```

Tensors are stored in a shared preallocated buffer B sized to accommodate the peak memory usage. The value of each A_e variable represents the offset location of tensor e in B.

We can map memory allocation requests to addresses over multiple iterations of the training loop as follow. We'll assume that each operator generates a single output tensor for the sake of simplicity, but our approach generalizes to handle operators with multiple outputs. The k_{th} memory allocation request corresponds to the tensor generated by the operator located at position $k \bmod |V|$ in the execution sequence ES. This tensor e is to be located at address $A_B + A_e$, where A_B is the base address of buffer B. Memory deallocation requests are no-ops.

5. Experiments

We measured the impact of MODeL on the memory usage of DNN training. We tried to answer the following questions:

- How effective is our algorithm at reducing peak memory usage?
- How practical are our algorithms? Can they be applied to large neural networks in a reasonable amount of time?
- What are the respective contributions of our two strategies of node reordering and address generation to the overall memory reduction?

5.1. Experimental Setup

We implemented MODeL on top of PyTorch version 1.11 (Paszke et al., 2019) with torchtext 0.12 and torchvision 0.12. We leveraged torch.FX to convert neural networks into executable sequences of operator calls, and reconstructed the computation graphs from the operator arguments. We encoded and solved the memory optimizations problem (equation 9) using Gurobi version 9.1.1 (Gurobi Optimization, LLC, 2022). We translated the Gurobi results into optimized execution sequences and memory locations as described in section 4.5.

We leveraged several techniques to optimize the implementation of our approach. We describe the most impactful optimizations in appendix A.

We ran all our experiments on a workstation featuring a Intel Xeon Gold 6138 CPU running at 2.0 GHz and a NVidia A100 GPU. We show how to integrate MODeL in a regular training flow in appendix B.

5.2. Methodology

We evaluated MODeL on a comprehensive set of neural networks. We included the ResNet (He et al., 2016) and Transformer (Vaswani et al., 2017) models since they are ubiquitous and used in many downstream tasks: the former introduced the concept of residual connection, and the later popularized the attention mechanism. We also included neural networks designed for specific tasks, such as computer vision (AlexNet (Krizhevsky et al., 2012), VGG (Simonyan & Zisserman, 2015), video understanding (ResNet3D (Tran et al., 2018)), and large language models (BERT (Devlin et al., 2018), XLM-R (Conneau et al., 2019)).

In addition to these models that were designed to run on datacenter hardware, we also evaluated our approach on MobileNet (Howard et al., 2017). This neural network was tailored to run in resource constrained environments such as edge devices. Additionally, we trained the neural networks at batch size 1 and 32. Batch size 1 is commonly used when training a model on devices with limited memory capacity, while batch size 32 is used often when running in datacenters.

To be representative of the evolution of DNN designs over time, we made sure our models cover almost a decade of machine learning research, starting with AlexNet (Krizhevsky et al., 2012) which was published back in 2012 and ending with VIT (Dosovitskiy et al., 2020) which was released in 2020. We also tested our approach on MNASNet (Tan et al., 2019), a model designed by a computer using an automated process called neural architecture search (Elsken et al., 2019).

To validate the scalability of our solution, we tested it on neural networks as small as Alexnet (Krizhevsky et al., 2012) (118 operators) and as large as BERT (2116 operators).

5.3. Overall Memory Improvement

Optimizing for both operator ordering and the memory location of tensors using equation 9 results in a reduction in peak memory usage ranging from 8 to 45% at batch size 1, and 12 to 68% at batch size 32. The average saving was 31.4% for batch size 1 and 32.8% for batch size 32 (Figure 5).

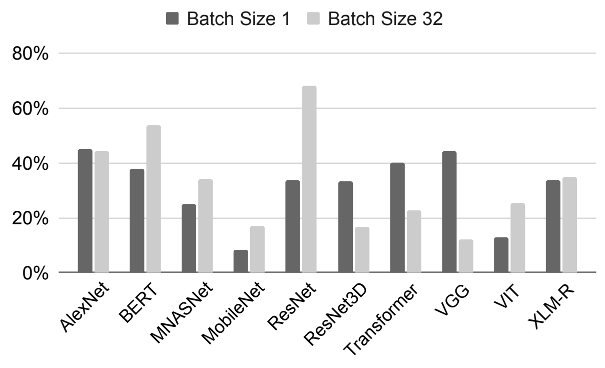


Figure 5. Total reduction in peak memory usage (in %) during training at various batch sizes compared to PyTorch.

The memory optimization process takes an average of 7.4 ± 0.7 seconds. In the worst case, our algorithm needs 18.1 seconds to run, and the best case is 100 milliseconds (Figure 6). This process is run only once before training the model. It introduces a negligible overhead to the total training time yet significantly reduces the peak memory usage.

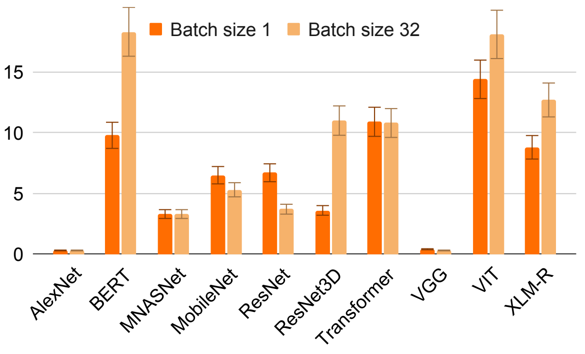


Figure 6. Memory optimization times (in seconds) for training graphs at batch sizes 1 and 32.

5.4. Impact of Address Generation

We define the fragmentation of a memory allocator as the difference between the memory the allocator needs to reserve from the hardware MR and the size of the resident set RS. We measure it when MR reaches its peak value using the ratio $(MR - RS)/MR$.

We measured the ability of our memory optimizer to reduce memory fragmentation in two scenarios: first, when our optimizer is free to reorder operators, and second when it is forced to honor the PyTorch operator ordering. The second scenario is implemented by constraining the values of the $C_{e,\,s}$ variables to be consistent with the PyTorch node ordering. In both cases, we found that our address generator was able to completely eliminate memory fragmentation on all the models of our benchmark suite. By contrast, PyTorch suffered from an average fragmentation of 7.8% at batch size 1, and 21.3% at batch size 32 (Figure 7). The PyTorch memory allocator uses a different strategy for small and large objects, which could explain why fragmentation is significantly worse for the larger batch size. However, it is unclear whether it could be modified to better handle large tensors without introducing other drawbacks.

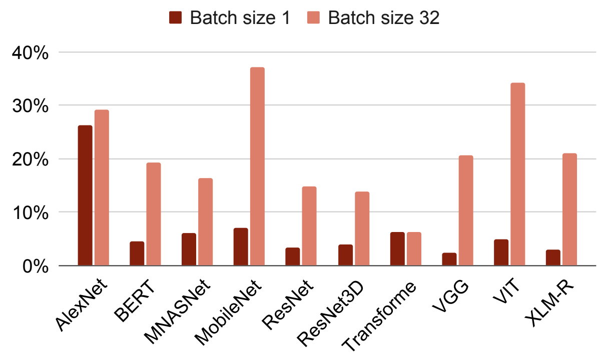


Figure 7. PyTorch memory fragmentation (in %) during training at various batch sizes. Our method fully eliminates fragmentation.

5.5. Impact of Operator Reordering

To evaluate the impact of our tensor lifetime optimization on the overall result, we compared the ideal peak memory necessary to run various neural networks when using the PyTorch node ordering and the node ordering determined by our algorithm. For these measurements, we eliminated the impact of memory fragmentation by recording the peak memory PyTorch operators need to request from the system to run these models under both node orderings instead of the memory actually used to run the models.

We find that optimizing the order in which operators are run reduces peak memory usage by up to 38% compared to PyTorch (Figure 8). On average, our solution achieves a reduction of 23.9% at batch size 1 and 11.7% at batch size 32.

The activations generated during the forward pass are preserved in memory for the backward pass of the training. As a result, MODeL has limited ability to decrease the memory usage of the forward pass. On the other hand, the order of the computation and application of the gradients with respect to

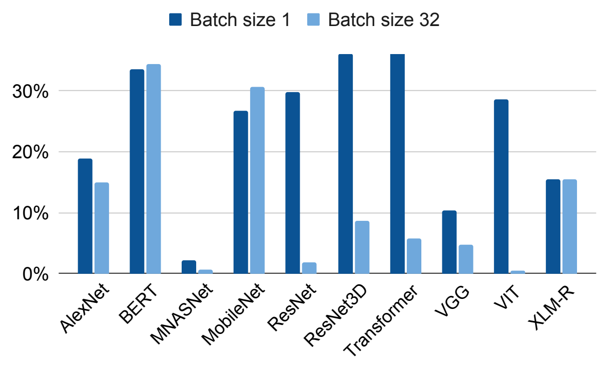


Figure 8. Reduction (in %) in ideal peak memory usage compared to PyTorch as a result of our node reordering. Ideal memory usage assumes that there is no fragmentation.

the weights offers a great deal of flexibility, which MODeL leverages to decrease the memory usage of the backward pass. However, the gradients with respect to the weights are roughly smaller than the activations by a factor of batch size. Therefore, at large batch sizes, a larger percentage of the total memory is used to store activations, while at smaller batch sizes these gradients represent a larger fraction of the total. As a result operator reordering tends to be more effective at small batch sizes.

6. Limitations

Our approach suffers from three main limitations. First, the neural network we want to optimize must be representable using a dataflow graph. This is the case for all the major neural architectures, so we do not believe that this is a significant drawback in practice. Second, the sizes of all the tensors must be known ahead of time. In the case where sizes are variable (for example, in the case of a language model operating on sentences of variable length) one can optimize for the worst case scenario (e.g the longest sentence). Third, our formulation assumes that tensors are stored in a single contiguous memory space. It can be extended to support multiple memory spaces (for example, when multiple devices are used to train a large neural network), but this is beyond the scope of this paper.

7. Conclusion

The limited memory capacity of the hardware used by deep learning practitioners is one of the main challenges to train state-of-the art neural networks. This ”memory wall” limits the size of the neural networks that can be trained, and ultimately impacts the quality of their predictions. Furthermore, as memory needs increase much faster than memory capacity, we expect this memory bottleneck to worsen over time.

To alleviate memory scarcity, we proposed to optimize both the lifetime and location of tensors in memory and we presented an ILP formulation of the problem.

We tested our solution, MODeL, on a wide variety of neural networks. We demonstrated experimentally that it can locate tensors optimally in memory, thus eliminating the problem of memory fragmentation. Furthermore, we showed that it further decreases the peak memory usage of deep neural networks by optimizing the lifetime of the tensors, and, by combining these 2 techniques, MODeL reduced peak memory usage by more than 30% on average.

We also emphasized the practicality of MODeL. We established empirically that it scales well and can handle large DNNs. We showed that it finds optimal memory plans in just a few seconds.

8. Acknowledgements

We would like to thank Ana Klimovic and Foteini Strati, whose insightful comments and feedback helped improve the paper. We are also grateful to the paper reviewers for their suggestions.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Ahn, B. H., Lee, J., Lin, J. M., Cheng, H.-P., Hou, J., and Esmaeilzadeh, H. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices, 2020.

Baruch, Z. Scheduling algorithms for high-level synthesis. *ACAM Scientific Journal*, 5(1-2):48–57, 1996.

Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *NeurIPS 2021 - Thirty-fifth Conference on Neural Information Processing Systems*, Virtual-only Conference, France, December 2021. URL https://hal.inria.fr/hal-03359793.

Bernstein, D., Rodeh, M., and Gertner, I. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Computers*, 38:1308–1313, 1989.

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. What is the state of neural network pruning? In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 129–146, 2020. URL https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

Bruno, J. and Sethi, R. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, jul 1976. ISSN 0004-5411. doi: 10.1145/321958.321971. URL https://doi.org/10.1145/321958.321971.

Bubeck, S. and Sellke, M. A universal law of robustness via isoperimetry. In *NeurIPS 2021*, December 2021. URL https://www.microsoft.com/en-us/research/publication/a-universal-law-of-robustness-via-isoperimetry/.

Chen, J., Zheng, L., Yao, Z., Wang, D., Stoica, I., Mahoney, M., and Gonzalez, J. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 1803–1813. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/chen21z.html.

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K. P., and Yuille, A. L. Semantic image segmentation with deep convolutional nets and fully connected crfs. *CoRR*, abs/1412.7062, 2015.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *ArXiv*, abs/1604.06174, 2016.

Chen, X., Ma, H., Wan, J., Li, B., and Xia, T. Multi-view 3d object detection network for autonomous driving. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6526–6534, 2017.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *ArXiv*, abs/1904.10509, 2019.

Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., Grave, E., Ott, M., Zettlemoyer, L., and Stoyanov, V. Unsupervised cross-lingual representation learning at scale. *CoRR*, abs/1911.02116, 2019. URL http://arxiv.org/abs/1911.02116.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL https://arxiv.org/abs/1901.02860.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Łukasz Kaiser. Universal transformers, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. URL https://arxiv.org/abs/1810.04805.

Dong, C., Loy, C. C., He, K., and Tang, X. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38:295–307, 2016.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale, 2020. URL https://arxiv.org/abs/2010.11929.

Elkerdawy, S., Elhoushi, M., Zhang, H., and Ray, N. Fire together wire together: A dynamic pruning approach with self-supervised mask prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL http://jmlr.org/papers/v20/18-598.html.

Evans, J. jemalloc. https://github.com/jemalloc/jemalloc.

Fan, A., Stock, P., Graham, B., Grave, E., Gribonval, R., Jegou, H., and Joulin, A. Training with quantization noise for extreme model compression. *arXiv preprint arXiv:2004.07320*, 2020.

Feichtenhofer, C., Fan, H., Malik, J., and He, K. Slowfast networks for video recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2018. URL http://arxiv.org/abs/1803.03635. cite arxiv:1803.03635Comment: ICLR camera ready.

Garey, M. R. and Johnson., D. S. Computers and intractability: A guide to the theory of np-completeness. *Journal of Symbolic Logic*, 1979.

Gholami, A., Azad, A., Jin, P. H., Keutzer, K., and Buluç, A. Integrated model, batch, and domain parallelism in training neural networks. *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018.

Google. Tcmalloc. https://github.com/google/tcmalloc.

Griewank, A. and Walther, A. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26:19–45, 2000.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL https://www.gurobi.com.

Hard, A., Rao, K., Mathews, R., Beaufays, F., Augenstein, S., Eichner, H., Kiddon, C., and Ramage, D. Federated learning for mobile keyboard prediction. *ArXiv*, abs/1811.03604, 2018.

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

He, Y., Ding, Y., Liu, P., Zhu, L., Zhang, H., and Yang, Y. Learning filter pruning criteria for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

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

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5MB model size, 2017. URL https://openreview.net/forum?id=S1xh5sYgx.

Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pp. 776–789. IEEE Press, 2018. ISBN 9781538659847. doi: 10.1109/ISCA.2018.00070. URL https://doi.org/10.1109/ISCA.2018.00070.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 497–511, 2020. URL https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in accelerating convolutional neural networks. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2274–2283. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/jia18a.html.

Kalamkar, D. D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., and Dubey, P. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019. URL http://arxiv.org/abs/1905.12322.

Karakus, C., Huilgol, R., Wu, F., Subramanian, A., Daniel, C., Çavdar, D., Xu, T., Chen, H., Rahnama, A., and Quintela, L. Amazon sagemaker model parallelism: A general and flexible framework for large model training. *CoRR*, abs/2111.05972, 2021. URL https://arxiv.org/abs/2111.05972.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012.

Liberis, E. and Lane, N. D. Neural networks on microcontrollers: saving memory at inference via operator reordering, 2020.

Lin, D., Talathi, S., and Annapureddy, S. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pp. 2849–2858. PMLR, 2016.

Lin, D. D. and Talathi, S. S. Overcoming challenges in fixed point training of deep convolutional networks. *ArXiv*, abs/1607.02241, 2016.

Lin, J., Zhu, L., Chen, W.-M., Wang, W.-C., Gan, C., and Han, S. On-device training under 256kb memory, 2022. URL https://arxiv.org/abs/2206.15472.

Louizos, C., Welling, M., and Kingma, D. P. Learning sparse neural networks through l_0 regularization, 2018.

Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y. Training deeper models by gpu memory optimization on tensorflow. In *NIPS 2017 Workshop on ML Systems*, 2017. URL http://learningsys.org/nips17/assets/papers/paper_18.pdf.

Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. A hierarchical model for device placement. In *ICLR*, 2018.

Molchanov, P., Mallya, A., Tyree, S., Frosio, I., and Kautz, J. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. 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*, PLDI 2021, pp. 883–898, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454083. URL https://doi.org/10.1145/3453483.3454083.

NVidia. Mixed precision training. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html. [Online; accessed 13-Oct-2022].

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff*, 2017. URL https://openreview.net/forum?id=BJJsrmfCZ.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Patil, S. G., Jain, P., Dutta, P., Stoica, I., and Gonzalez, J. POET: Training neural networks on tiny devices with integrated rematerialization and paging. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 17573–17583. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/patil22b.html.

Paulik, M., Seigel, M., Mason, H., Telaar, D., Kluivers, J., van Dalen, R., Lau, C. W., Carlson, L., Granqvist, F., Vandevelde, C., Agarwal, S., Freudiger, J., Byde, A., Bhowmick, A., Kapoor, G., Beaumont, S., Cahill, A., Hughes, D., Javidbakht, O., Dong, F., Rishi, R., and Hung, S. Federated evaluation and tuning for on-device personalization: System design and applications, 2021. URL https://arxiv.org/abs/2102.08503.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. 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*, ASPLOS '20, pp. 891–905, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378505. URL https://doi.org/10.1145/3373376.3378505.

Sekiyama, T., Imamichi, T., Imai, H., and Raymond, R. Profile-guided memory optimization for deep neural networks, 2018. URL https://arxiv.org/abs/1804.10001.

Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., and Villalobos, P. Compute trends across three eras of machine learning. *2022 International Joint Conference on Neural Networks (IJCNN)*, Jul 2022. doi: 10.1109/ijcnn55064.2022.9891914. URL http://dx.doi.org/10.1109/IJCNN55064.2022.9891914.

Shah, A., Wu, C.-Y., Mohan, J., Chidambaram, V., and Kraehenbuehl, P. Memory optimization for deep networks. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=bnY0jm4l59.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

Smith, S. L., Kindermans, P.-J., and Le, Q. V. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=B1Yy1BxCZ.

Tai, Y., Yang, J., and Liu, X. Image super-resolution via deep recursive residual network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *ArXiv*, abs/1905.11946, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile, 2019.

Tran, D., Wang, H., Torresani, L., Ray, J., LeCun, Y., and Paluri, M. A closer look at spatiotemporal convolutions for action recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6450–6459, 2018. doi: 10.1109/CVPR.2018.00675.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf.

Wikipedia. Integer programming. https://en.wikipedia.org/wiki/Integer_programming, 2023. [Online; accessed 15-Mar-2023].

Zheng, B., Vijaykumar, N., and Pekhimenko, G. Echo: Compiler-based gpu memory footprint reduction for lstm rnn training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1089–1102, 2020. doi: 10.1109/ISCA45697.2020.00092.

Zhu, F., Gong, R., Yu, F., Liu, X., Wang, Y., Li, Z., Yang, X., and Yan, J. Towards unified int8 training for convolutional neural network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

A. Scaling to Large Neural Networks

Our formulation requires $2 \times |E| \times |V|)$ binary variables since we have one C and one P variable per tensor per step, as well as $|E|$ integer variables to track tensor addresses. Additionally, we create $\mathcal{O}(|V| \times |E|)$ constraints to encode tensor precedence and life cycle requirements, and $\mathcal{O}(|V| \times |E|^2)$ constraints to ensure that tensors never overlap in memory.

We develop the following techniques to reduce the complexity of the ILP formulation and enable our approach to scale well. This permits MODeL to optimize the memory usage of neural networks with complex tensor computation graphs comprised of thousands of vertices and edges.

A.1. Bounding Lifetime Ranges

All of the input tensors of a node must reside in memory for it to run at a given step. This means that all the operators in the immediate fanin of the node must have been run at least one step prior. As a result, we can identify the earliest step $ASAP(v)$ ("as soon as possible") during which a node v can run. $ASAP(v)$ is the longest distance from v to an input of the neural network, which is computed in linear time using a simple depth first search traversal of the graph (Baruch, 1996). Using the same approach, we can also identify the latest step $ALAP(v)$ ("as late as possible") at which a node v can run, which is the longest distance from v to an output of the neural network.

A node v can only run within the span $[ASAP(v), ALAP(v)]$. Since tensors are created when their source node is run, a variable $C_{e,s}$ will always be false outside the span of their source node (Equation 10).

$$\begin{gathered} SPAN(v) = [ASAP(v), ALAP(v)] \\ \forall e \in E,\ \ \forall s \notin SPAN(src(e)) \quad C_{e,s} = 0 \end{gathered} \tag{10}$$

Furthermore, a tensor only needs to be preserved in memory until all its sink operators have run. This enables us to define the Maximum Useful Lifetime (MUL) range of a tensor, and set the variable $P_{e,s}$ for a tensor e to false outside of this range (Equation 11).

$$\begin{gathered} MUL(e) = [ASAP(src(e)), \max_{f \in snks(e)} ALAP(f)] \\ \forall e \in E,\ \ \forall s \notin MUL(e) \quad P_{e,s} = 0 \end{gathered} \tag{11}$$

Additionally, tensors must be preserved in memory from the time they are created until their last sink node has run. Therefore, $P_{e,s}$ must always be true from the last step at which e can be created until the earliest step at which its last sink can run (Equation 12).

$$\begin{gathered} PRES(e) = [ALAP(src(e) + 1, \max_{f \in snks(e)} ASAP(f)] \\ \forall e \in E,\ \ \forall s \in PRES(e) \quad P_{e,s} = 1 \end{gathered} \tag{12}$$

This enables us to reduce the number of steps to track for each tensor. In the best case scenario, where a neural network is a linear sequence of operators, the span of each node v is reduced to a single step, and we can derive the values of all the $C_{e,s}$ and $P_{e,s}$ purely from the structure of the graph. However, in the opposite extreme case where a neural network consists exclusively of operators that can run in parallel, we cannot infer any of the values of the $C_{e,s}$ and $P_{e,s}$ variables. The structure of real neural networks lies somewhere between these two extremes.

A.2. Leveraging Precedence Constraints

We simplify our memory placement formulation by avoiding the need to create the variables and constraints from equations 6, 7a and 7b whenever we can determine that two tensors can never reside in memory at the same time. We exploit two sufficient conditions to achieve this.

First, we leverage the Maximum Useful Lifetime ranges from our ASAP/ALAP analysis. If the MUL ranges of two tensors do not overlap, they will never be present concurrently in memory.

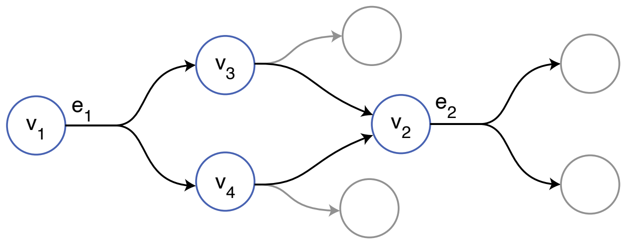


Figure 9. Edge precedence: $e_1 \prec_{prec} e_2$ since the sinks v_3 and v_4 of e_1 are both in the transitive fanin of the source node of e_2, and e_1 and e_2 have no vertex in common.

We complement this first condition with a precedence analysis. If a vertex v_2 is reachable from another vertex v_1 (i.e. if v_1 is in the transitive fanin of v_2), the corresponding operator v_1 must be run before operator v_2. Therefore, if all the sink vertices of an edge e_1 are in the transitive fanin of the source vertex of an edge e_2, e_1 and e_2 can only be present in memory if there is a vertex v such that e_1 is one of the fanout edges of v and e_2 is one of its fanin edges (Figure 9). We call this condition $\prec_{prec}$, and if either condition $e_1 \prec_{prec} e_2$ or $e_2 \prec_{prec} e_1$ holds e_1 and e_2 can never reside together in memory.

We use a simple depth-first search (Function 2) to determine whether a vertex v_2 is reachable from a vertex v_1. We leverage memoization to ensure that answering the query for a pair (v_1, v_2) yields to constant time queries for all future queries (v, v_2) that involve a vertex v on a path from v_1 to v_2.

Function 2 IsInTransitiveFanin(v1, v2, cache)

```
▷ Returns true iff v2 can be reached from v1.
if $(v1, v2)$ in $cache$ then
  return$cache[(v1, v2)]$
end if
for $f$ in $fi(v2)$ do
  if $src(f) = v1$ then
    $cache[(v1, v2)] \leftarrow true$
    return true
  end if
  if $IsInTransitiveFanin(v1, src(f))$ then
    $cache[(v1, v2)] \leftarrow true$
    return true
  end if
end for
$cache[(v1, v2)] \leftarrow false$
return false
```

B. Usage

MODeL can be integrated in a PyTorch training script with minimal effort. We demonstrate how to do this on a simple example.

```
import model_opt
import torch, torchvision

def accuracy(output, target):
    # Computes the number of top-1 predictions that match the labels.
    _, pred = torch.topk(output, 1)
    correct = pred.eq(target)
    return torch.sum(correct)

model = torchvision.models.resnet50().cuda()
sample_input = torch.rand(1, 3, 224, 224).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
loss_fn = torch.nn.CrossEntropyLoss()

model_trainer = model_opt.optimize(model, sample_input,
                                   optimizer=optimizer, loss_fn=loss_fn)

train_loader = torch.utils.data.DataLoader(...)
correct = 0
for example, target in train_loader:
    # model_trainer encapsulates the forward pass, the backward pass,
    # and the weight update step
    output = model_trainer(example)
    correct += accuracy(output, target)
```