An Efficient Training Framework for Reversible Neural Architectures

Zixuan Jiang[1][0000−0001−6180−6487], Keren Zhu[1][0000−0003−2698−141X], Mingjie Liu[1][0000−0002−3488−9763], Jiaqi Gu[1][0000−0001−8535−7698], and David Z. Pan[1][0000−0002−5705−2501]

The University of Texas at Austin, Austin Texas 78712, USA
{zixuan, keren.zhu, jay_liu, jqgu}@utexas.edu, dpan@ece.utexas.edu
https://www.cerc.utexas.edu/utda/

Abstract. As machine learning models and dataset escalate in scales rapidly, the huge memory footprint impedes efficient training. Reversible operators can reduce memory consumption by discarding intermediate feature maps in forward computations and recover them via their inverse functions in the backward propagation. They save memory at the cost of computation overhead. However, current implementations of reversible layers mainly focus on saving memory usage with computation overhead neglected. In this work, we formulate the decision problem for reversible operators with training time as the objective function and memory usage as the constraint. By solving this problem, we can maximize the training throughput for reversible neural architectures. Our proposed framework fully automates this decision process, empowering researchers to develop and train reversible neural networks more efficiently.

Keywords: reversible neural networks, efficient training, machine learning framework

1 Introduction

The backpropagation [20] mechanism is widely used in training neural networks. However, since intermediate results need to be saved for backward computations, the backpropagation requires considerable memory footprint. As neural networks become larger and deeper, the increasing memory footprint is forcing the usage of smaller mini-batch sizes. In extreme cases, deep networks have to be trained with a mini-batch size of 1 [25]. The issue of memory consumption impedes the explorations of desirable deep learning models.

Researchers have proposed several methods to address the challenge of inflating memory footprint [21]. Chen et al. [6] propose gradient checkpoint mechanism to store partial intermediate results. The discarded activations will be recovered through recomputations in the backward pass. The memory swapping method [19, 24] moves intermediate activations to other devices to reduce the memory footprint of the current device. The extra memory transfer imposes

overhead on training efficiency. Reversible operators [7] allow recovering the intermediate feature maps in backward pass through the corresponding inverse functions. All these three methods reduce the memory footprint at the cost of extra computation or memory transfer. They do not affect the model accuracy as the training process is numerically unchanged.

Specifically, reversible neural architectures have been successfully adopted in computer vision research, e.g., the reversible U-net for volumetric image segmentation [3], and the reversible architecture for 3D high-resolution medical image processing [2]. The lower memory footprint allows deeper models to be trained, inducing more predictive capability and higher accuracy.

Figure 1 shows two extremes in neural network training. Standard backpropagation achieves the extreme of computation efficiency at the expense of the highest memory footprint, such that it does not contain any redundant computations. On the other extreme, the fully reversible strategy has the lowest memory footprint with imposing the greatest computation overhead. However, The design space between two extremes is less studied. Existing research regarding reversible neural networks mainly focuses on saving memory consumption. All the reversible layers are executed in the memory-efficient mode. The computation overhead of their inverse functions is overlooked.

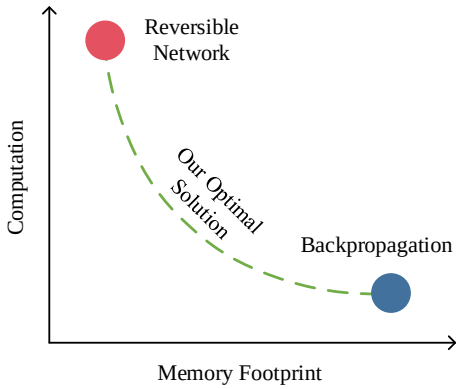


Fig. 1: Two extremes when training neural networks. The lower right extreme stands for the standard backpropagation method, which does not contain any redundant computations. The upper left extreme can achieve the lowest memory footprint by fully leveraging the reversibility of the neural network.

In this paper, we explore the design space by considering the trade-off between computation and memory footprint. We derive the mathematical formulation of the decision problem for reversible neural architectures. We formulate the training time as the objective function with memory usage as an optimization constraint. By showing that it is a standard 0/1 knapsack problem in essence,

we use a dynamic programming algorithm to find the optimal solution. We also discuss the relationship between mini-batch size and training throughput.

Our contributions are highlighted as follows.

- **New Perspective.** We explore the design space for reversible neural architectures from a novel perspective of joint optimization.
- **Optimality.** Our framework guarantees to obtain the maximum training throughput for reversible neural architectures under given memory constraints.
- **Automation.** Our framework provides a fully automated solution, enabling more efficient development and training for reversible neural networks.

2 Background

In this section, we discuss the background of reversible neural architectures and the scheduling framework for the training process.

2.1 Reversible Neural Architectures

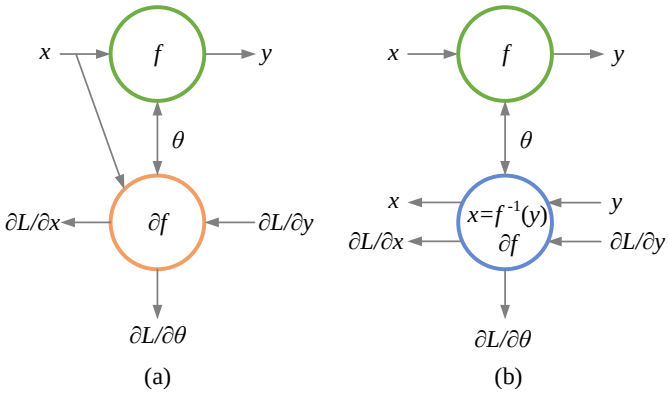


Fig. 2: (a) non-reversible and (b) reversible neural architectures. For a non-reversible layer, we often need to save its original input x for backward computations. For a reversible layer, the original input x can be calculated via its inverse function $x = f^{-1}(y)$.

Figure 2a demonstrates a conventional non-reversible neural architectures. The layer $y = f(x)$ is non-reversible if and only if there is no inverse computation $x = f^{-1}(y)$ for the original function f. For a non-reversible layer, we often need to store its original input x during forward computation so that we can compute gradients during backpropagation. As an example, for a linear layer

$y = f(x) = \theta^T x$, where θ represents the weight vector, its backward computation $\partial y / \partial \theta = x$ depends on the original input x.

Traditional neural networks are mostly based on these non-reversible layers. The memory consumed by the feature maps dominates the total memory utilization, especially in deep neural networks [19]. Therefore, the memory footprint can decrease significantly by discarding those feature maps.

Figure 2b illustrates a reversible operator. When using the reversible layer $y = f(x)$, it is possible recover x in the backward computation by calling its inverse function $x = f^{-1}(y)$. Therefore the memory consumption can be saved by discarding the intermediate feature map x.

Some of the commonly used operators in neural networks are implicitly reversible, such as convolution layers with a stride of 1 [12], and fully connected layers with invertible weight matrix. Inplace Activated Batch Normalization (ABN) [4] leverages the reversibility of the batch normalization [8] and some activation functions (such as leaky ReLU). Neural ordinary differential equations [5] can achieve constant memory usage through reversibility in backpropagation.

Researchers also propose many variations of explicit reversible neural architectures [9]. The reversible residual architecture [7] does computations on a pair of inputs (x_1, x_2) as shown in Equation 1.

$$y_1 = x_1 + F(x_2), y_2 = x_2 + G(y_1) \tag{1}$$

It is reversible since the inputs can be recovered from output pairs as demonstrated in Equation 2.

$$x_2 = y_2 - G(y_1), x_1 = y_1 - F(x_2) \tag{2}$$

This technique can be combined with traditional recurrent neural networks to get reversible RNNs [16]. Kitaev et al. apply the above architecture to the Transformer [22] and obtain Reformer [13] as shown in Equations 3 and 4.

$$y_1 = x_1 + \text{Attention}(x_2), y_2 = x_2 + \text{FeedForward}(y_1) \tag{3}$$

$$x_2 = y_2 - \text{FeedForward}(y_1), x_1 = y_1 - \text{Attention}(x_2) \tag{4}$$

Although the computation overhead is considered and discussed, these prior studies mainly focus on memory footprint reduction. They do not explore the space between the two extremes illustrated in Figure 1.

2.2 Scheduling for Training

For most developers, the primary concern regarding the training process is how to maximize the training throughput given existing machines, especially GPUs. Specifically, there is a need for a framework to automate the training process to fully utilize the computation capability and memory capacity of specific machines.

Frameworks for the scheduling problem with gradient checkpoints are great examples. The scheduling problem seeks the minimum computation overhead

with a memory footprint constraint. Researchers propose many algorithms to find optimal solutions for gradient checkpoint selection. Kusumoto et al. provide a dynamic programming algorithm from the perspective of computation graphs [14]. Jain et al. formulate the scheduling problem as a mixed integer linear program and solve it via standard solvers [10]. However, a similar problem for reversible neural architectures does not get much attention. We formulate and solve this problem in this work.

There are also work focused on the scheduling for distributed training. Jia et al. optimize how each layer is parallelized in distributed and parallel training [11]. However, they do not consider the reversibility. Our framework can be used directly in every single machine in the distributed training scenario.

3 Method

In this section, we first describe two modes for reversible neural architectures. We denote them `M-Mode` and `C-Mode`, respectively. We then formulate the decision problem, and propose an algorithm and our framework. We also discuss the problem when mini-batch sizes are not fixed.

3.1 Memory Centric and Computation Centric Modes

Each reversible layer $y = f(x)$ can be computed in two modes during the training process. First, we can leverage its reversibility. We denote it `M-Mode`, which represents **m**emory centric **mode**. Precisely, we discard the activation x in forward computations, then recover it in the backward pass. This mode saves the memory consumed by x at the cost of inverse computation of $x = f^{-1}(y)$. Another mode is treating the reversible layer as a conventional non-reversible layer, which is denoted `C-Mode` representing **c**omputation centric **mode**. In this mode, we save the feature map x in the forward pass, then use it directly in the backward computation. This mode does not involve redundant computations but requires an extra memory footprint. Table 1 summarizes these two modes.

Table 1: Comparisons of two modes.

mode	forward	backward	computation cost	memory cost
`M-Mode`	discard x	recover x from y	$x = f^{-1}(y)$	0
`C-Mode`	save x	use x directly	0	size of x

3.2 Formulation

Let f be a neural network with $(k+n)$ layers, among which there are n reversible layers $\{f_i\}_{i=1}^{n}$. For each of these n reversible layers, we can decide to do forward and backward computation following one of the modes above. Let $x \in \{0,1\}^n$ be the decision variable. $x_i = 0(1)$ means that the reversible layer f_i follows

the `M-Mode` (`C-Mode`). Thus, for n reversible layers, the 2^n choices constitute the whole solution space.

The two extremes in Figure 1 can be written as $x = \mathbf{0}$ and $x = \mathbf{1}$. $x = \mathbf{0}$ represents that we discard all the intermediate results to achieve the lowest memory footprint. We treat it as `baseline-M`. We denote the other extreme without redundant computations ($x = \mathbf{1}$) as `baseline-C`. Currently, most of the implementations of reversible neural networks use `baseline-M` directly.

Let $t_{f1}, t_{b1}(t_{f2}, t_{b2}) \in \mathbb{R}^n_{++}$ be the execution time vector of forward and backward pass in the `M-Mode` (`C-Mode`) respectively. Compared with the `C-Mode`, the extra execution time consumed by the `M-Mode` is $t_e = (t_{f1} + t_{b1}) - (t_{f2} + t_{b2})$. The total execution time of forward and backward computation of all these reversible layers $\{f_i\}_{i=1}^n$ are

$$(\mathbf{1} - x)^T(t_{f1} + t_{b1}) + x^T(t_{f2} + t_{b2}) = \mathbf{1}^T(t_{f1} + t_{b1}) - t_e^T x$$

Similarly, let $m \in \mathbb{Z}^n_{++}$ be the extra memory footprint of `C-Mode` compared with `M-Mode`, i.e., the size of corresponding intermediate activations. The total extra memory footprint of these feature maps is $m^T x$.

Finally, the time centric optimization problem can be written as Problem 5.

$$\begin{aligned} \min_{x} \quad & \mathbf{1}^T(t_{f1} + t_{b1}) - t_e^T x \\ \text{s.t.} \quad & m^T x + m_o \leq M \\ & x_i \in \{0, 1\}, i = 1, ..., n \end{aligned} \tag{5}$$

where M is the memory capacity of the machine, m_o represents the memory allocated for other tensors (such as feature maps of non-reversible layers, and neural network parameters) when we achieve peak memory in a training iteration. Users can also specify the memory capacity M explicitly.

For other parts of the training process, such as the optimizer, the computation of non-reversible layers, their execution time is constant and independent of our decisions. Therefore, we can minimize the training time by minimizing the total wall-clock time of all these reversible layers.

3.3 Algorithm and Framework

Problem 5 can be rewritten as Problem 6.

$$\begin{aligned} \max_{x} \quad & t_e^T x \\ \text{s.t.} \quad & m^T x \leq M - m_o \\ & x_i \in \{0, 1\}, i = 1, ..., n \end{aligned} \tag{6}$$

Problem 6 can be interpreted as follows. We take the `baseline-M` ($x = \mathbf{0}$) as the reference. The object function $t_e^T x$ is the execution time reduction when we apply the decision x. The remaining memory capacity for these reversible layers is $M - m_o$.

Problem 6 is a standard **0/1 knapsack problem** in essence [17]. Note that the memory-related variables and parameters m, M, m_o are all positive integers since all of them are in the unit of bytes. Therefore, it can be solved by dynamic programming, as shown in Algorithm 1.

Algorithm 1 Dynamic programming algorithm for 0/1 knapsack problem

Input: $t_e, m, M - m_o, n$. {Indices of vectors t_e and m strat from 1.}
Define saved$[n, M - m_o]$ and initialize all entries as -1, which means the entry is undefined. {The entry saved$[i, j]$ records the maximum saved time under the condition that we consider first i items with total memory limit of j.}
foo(i, j) {This recursive function calculates saved$[i, j]$.}
  if $i == 0$ **or** $j \leq 0$ **then**
    return 0 {No time saved under this condition}
  end if
  if saved$[i-1, j] == -1$ **then**
    saved$[i-1, j] =$ **foo**$(i-1, j)$
  end if
  if $m[i] > j$ **then**
    saved$[i, j] =$ saved$[i-1, j]$
  else
    if saved$[i-1, j-m[i]] == -1$ **then**
      saved$[i-1, j-m[i]] =$ **foo**$(i-1, j-m[i])$
    end if
    saved$[i, j] = \max\{$saved$[i-1, j],$ saved$[i-1, j-m[i]] + t_e[i]\}$
  end if
  return saved$[i, j]$
end foo
saved$[n, M - m_o] =$ **foo**$(n, M - m_o)$
Initialize decision variables $x = \mathbf{0}$ {Do backtracking to find the optimal solution.}
$j = M - m_o$
for $i = n, n-1, ..., 1$ **do**
  if saved$[i, j] \neq$ saved$[i-1, j]$ **then**
    $x[i] = 1$
    $j = j - m[i]$
  end if
end for
return saved$[n, M - m_o], x$ {Return optimal values and solutions.}

Based on the algorithm, we propose a framework to automate the decision process. Figure 3 shows the four stages of our framework. Initially, we verify the reversibility of each operator. The correctness of the original and inverse functions will be verified. In the second stage, we will obtain parameters t_e and m from realistic measurements. Our framework is hardware-aware since we use realistic profiling data from specific machines. Then we use Algorithm 1 to get the optimal solution. Finally, we can train the network with maximum throughput. The dynamic programming algorithm will only be executed once

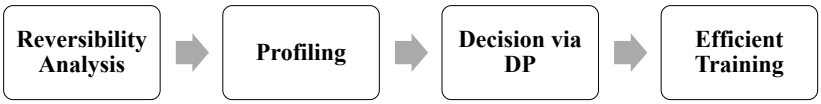


Fig. 3: Four stages in our framework

to obtain the optimal schedule. After that, this schedule can be used in all the training iterations. Thus, the added complexity is negligible compared with the training process.

3.4 Various Mini-batch Size

The above discussions are based on the assumption that the mini-batch size is fixed. When we have many choices on the mini-batch size (denoted b), the optimization problem will be more complicated.

We assume that for each layer, its execution time is linear to the batch size, whether it is reversible or not. Namely, the execution time satisfies that $t(b) = t^{(0)} + bt^{(1)}$. The total execution time of all the non-reversible layers is $t_n^{(0)} + bt_n^{(1)}$. The total execution time of all the reversible layers is

$$\mathbf{1}^T(t_{f1} + t_{b1}) - t_e^T x = \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)}) + b\mathbf{1}^T(t_{f1}^{(1)} + t_{b1}^{(1)}) - {t_e^{(0)}}^T x - b{t_e^{(1)}}^T x$$

The execution time of the optimizer, scheduler, and control are independent of mini-batch size, denoted by t_o. The execution time per sample is

$$t_n^{(1)} + \mathbf{1}^T(t_{f1}^{(1)} + t_{b1}^{(1)}) - {t_e^{(1)}}^T x + \frac{t_o + t_n^{(0)} + \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)}) - {t_e^{(0)}}^T x}{b}$$

The memory footprint is also linear to the mini-batch size. The size of network parameters is independent of the mini-batch size. The size of the feature maps of the non-reversible layers is proportional to the mini-batch size. Thus, the memory constraint can be rewritten as $bm^T x + m_o^{(0)} + bm_o^{(1)} \leq M$.

The optimization problem is now

$$\begin{aligned} \min_{x,b} \quad & t_n^{(1)} + \mathbf{1}^T(t_{f1}^{(1)} + t_{b1}^{(1)}) - {t_e^{(1)}}^T x + \frac{t_o + t_n^{(0)} + \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)}) - {t_e^{(0)}}^T x}{b} \\ \text{s.t.} \quad & bm^T x + m_o^{(0)} + bm_o^{(1)} \leq M \\ & x_i \in \{0,1\}, i = 1, ..., n \\ & b \in [b_l, b_u], b \in \mathbb{Z} \end{aligned} \tag{7}$$

where b_l, b_u are lower and upper bounds of the mini-batch size.

Rewrite the problem as Problem 8.

$$\begin{aligned} \max_{x,b} \quad & f(x,b) = {t_e^{(1)}}^T x - \frac{C - {t_e^{(0)}}^T x}{b} \\ \text{s.t.} \quad & bm^T x + m_o^{(0)} + bm_o^{(1)} \leq M \\ & x_i \in \{0,1\}, i = 1, ..., n \\ & b \in [b_l, b_u], b \in \mathbb{Z} \end{aligned} \tag{8}$$

where $C = t_o + t_n^{(0)} + \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)})$ is a constant.

Problem 8 is a non-linear integer programming optimization problem, which is hard to get the optimal solution. A simple method is to sweep the mini-batch size in the range of $[b_l, b_u]$ with our framework. Empirically the Problem 6 is fast to solve using Algorithm 1. Thus, it is affordable to apply the simple method of sweeping the mini-batch size. We further discuss various mini-batch size in in Section 4.6. We leave Problem 8 as an open problem for future research.

4 Experiments

In this section, we provide the experimental settings initially. Then we discuss the details of profiling. We analyze three reversible neural architectures: RevNet-104, ResNeXt-101 with inplace ABN, and Reformer. We further discuss the results in terms of various mini-batch sizes.

4.1 Settings

We adapt source codes from MemCNN[1] [15], Inplace ABN[2] [4], and Reformer[3] [13]. We follow their original settings and hyperparameters except that we can decide what modes each reversible layer will use.

Unless otherwise stated, we use PyTorch[18] 1.4.0. The training process runs on a Linux server with Intel Core i9-7900X CPU and 1 NVIDIA TITAN Xp GPU, whose memory capacity is 12,196 MiB. All the tensor operations are on the GPU. We report the mean of 100 training iterations.

4.2 Profiling

To ensure hardware-awareness, our framework needs to do profiling on the execution time and memory allocation to obtain t_e, m, m_o based on realistic measurement. It is easy to collect memory-related terms m, m_o since the memory footprint is stable throughout a whole training process.

For the execution time t_e, the most accurate way to obtain it is running the model in two modes respectively and collect all the four corresponding vectors

[1] https://github.com/silvandeleemput/memcnn
[2] https://github.com/mapillary/inplace_abn
[3] https://github.com/lucidrains/reformer-pytorch

(t_{f1}, t_{b1}, t_{f2}, t_{b2}). We can also directly compare these two modes and conclude their difference. For the feature maps in the `C-Mode`, it takes extra time for the memory writes in the forward computation, and memory read in the backward pass. In the `M-Mode`, there is overhead in reading y from memory and the inverse computation.

It is complicated to analyze the memory behaviour, and the analysis is beyond discussions of this paper. Fortunately, we observe that $t_{f1} \approx t_{f2} \approx t_{b1} - t_{b2}$. For instance, the average execution time of the RevNet-104 [7] with mini-batch size of 64 on ImageNet is $t_{f1} = 10.425$ms, $t_{f2} = 10.404$ms, $t_{b1} = 29.276$ms, and $t_{b2} = 18.865$ms. This observation is prevalent in current machine learning frameworks, since memory accesses are hidden by computations [18, 1]. Thus, we can only take computation into account when analyzing the difference in execution time. In short, $t_e = (t_{f1}+t_{b1})-(t_{f2}+t_{b2}) \approx t_{f1} \approx t_{f2}$ The assumption is verified for all the following experiments. We use $t_e = t_{f1}$ in the optimization problem directly.

4.3 RevNet

We apply our framework on RevNet-104 [7] for image classification on ImageNet. By sweeping the mini-batch sizes, we can obtain various memory budgets and computation overhead. Figure 4 illustrates our decision for different mini-batch sizes. When the mini-batch size is smaller than 65, the GPU memory capacity is large enough to contain all the intermediate activations. Thus, the optimal decision is saving all of them to achieve maximum training throughput. Starting from a mini-batch size of 65, we have to use the `M-Mode` in partial reversible layers due to the limited memory budget. Our dynamic programming solver will obtain the optimal decision for each setting. If the mini-batch size is larger than 117, we will encounter the issue of out of memory even if we use `baseline-M`, the most memory-efficient decision. As shown in Figure 4, the optimal decision is non-trivial across different mini-batch sizes.

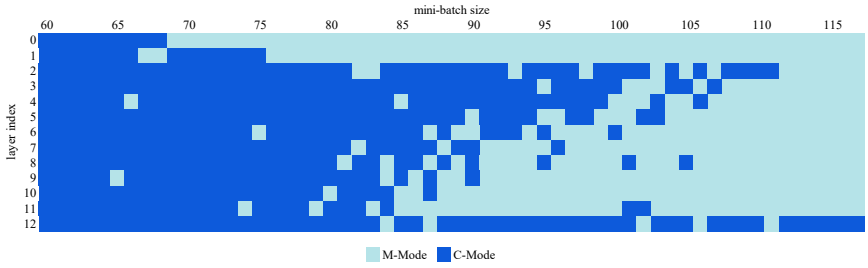
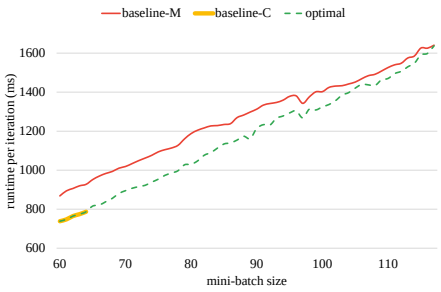
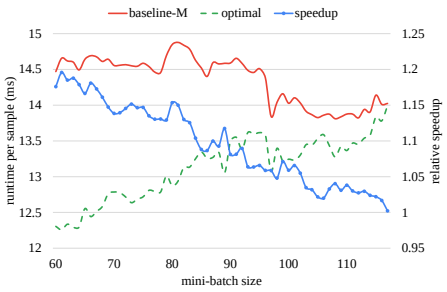


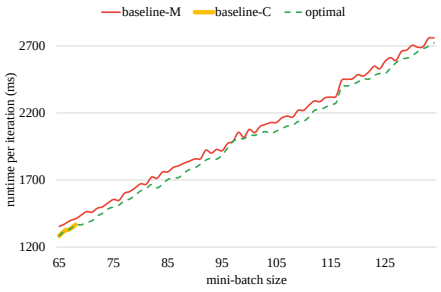
Fig. 4: The heat map of the optimal solutions throughout different mini-batch sizes on RevNet-104 with 13 reversible layers. The horizontal and vertical axes represent the mini-batch size and the layer index, respectively.



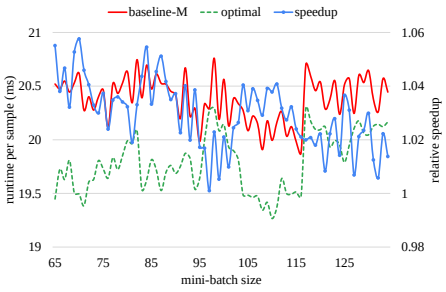
(a) Training time per iteration of RevNet-104



(b) Training time per sample and relative speedup of RevNet-104



(c) Training time per iteration of ResNeXt-101 with Inplace ABN



(d) Training time per sample and relative speedup of ResNeXt-101 with Inplace ABN

Fig. 5: Training time and speedup comparison of RevNet-104 and ResNeXt-101 with Inplace ABN on ImageNet. Training time per iteration is the time of one complete iteration (forward, backward, and optimizer updating). Training time per sample is the multiplicative inverse of training throughput. The curves of `baseline-C` are truncated due to device memory limitation.

Figure 5a shows the training time per iteration of `baseline-M`, `baseline-C` and our optimal solution. The solid red line and the green dashed line represents the `baseline-M` and optimal settings provided by our framework. The `baseline-C` is highlighted in the lower left corner, since it is limited by the device's memory capacity and cannot contain a large batch size. Our optimal solution overlaps with the `baseline-C` when `baseline-C` is feasible, i.e., mini-batch size smaller than 65. When `baseline-C` is not available, our framework approach the `baseline-M` gradually. The reason is that as the mini-batch size grows, the harsh memory constraint pushes us forward to the extreme of memory efficiency. The gap between the two curves (`baseline-M` and optimal) demonstrates the absolute time saved by applying our method.

Figure 5b compares the training time per sample. We can use this metric to compare the training throughput (which is the multiplicative inverse of the training time per sample) for different mini-batch sizes. Before applying our framework, the training speed increases as the mini-batch size grows for two reasons. First, we leverage the parallelism across batches. Second, the execution time of the optimizer, scheduler, and control is independent of the mini-batch size. This part of execution is amortized by the large mini-batch size. After using our framework, the trend is different. The training throughput decreases as the mini-batch size grows, because the computation overhead of inverse functions is much larger than the benefit from large mini-batch size. We also show the relative speedup of our optimal execution time compared with `baseline-M`. We can achieve up to 1.15× speedup for this benchmark.

4.4 Inplace ABN

We follow the settings in the paper of Inplace ABN [4] and use our framework to train ResNeXt-101 [23] for image classification on ImageNet. Figures 5c and 5d compares the training time per iteration across different mini-batch sizes. The results are similar to those of RevNet-104 except the relative speedup.

The computation overhead of the Inplace ABN is relatively low compared with RevNet-104 in the previous subsection. The execution time of `baseline-C` is only $0.8-2\%$ smaller than that of `baseline-M`. Therefore, the relative speedup using our method is not as significant as the experiments on RevNet-104. The reason is that the maximum training throughput of our framework is bounded by `baseline-C`. However, the advantage of our method is that we can find the optimal point across two baselines.

4.5 Reformer

We also do experiments on the enwik8 task with Reformer. Specifically, there are 8 heads in our 12-layer model. The maximum sequence length is 4,096, and the number of tokens is 256. For each iteration, we call the optimizer to update the trainable parameters after accumulating gradients for 4 steps. Table 2 shows the training time in different modes.

Table 2: Results of Reformer on enwik8 task. TPI and TPS are abbreviations for training time per iteration and training time per sample. OOM stands for out of memory. All the execution time is in the unit of seconds.

mini-batch	baseline-C	baseline-M	optimal	baseline-C	baseline-M	optimal	speedup
size	TPI	TPI	TPI	TPS	TPS	TPS	
1	0.951	1.321	0.949	0.951	1.321	0.949	1.392
2	1.738	2.533	1.738	0.869	1.266	0.869	1.457
3	OOM	3.603	2.752	OOM	1.201	0.917	1.310
4	OOM	4.792	4.175	OOM	1.198	1.044	1.148
5	OOM	6.020	5.236	OOM	1.204	1.047	1.150
6	OOM	7.210	6.692	OOM	1.202	1.115	1.077
7	OOM	8.420	7.670	OOM	1.203	1.096	1.098
8	OOM	9.490	9.044	OOM	1.186	1.130	1.049
9	OOM	10.603	10.123	OOM	1.178	1.125	1.047
10	OOM	11.873	11.295	OOM	1.187	1.129	1.051

Due to the large memory footprint, the `baseline-C` can only run with a mini-batch size of 2. The reversibility enables us to train the model with a mini-batch size up to 10. Our framework provides a smooth transition from `baseline-C` to `baseline-M`. We achieve 1.3× relative speedup when the mini-batch size is 3.

4.6 Various mini-batch sizes

For this subsection, we discuss the optimal mini-batch size from the perspective of training throughput. In the above experiments, the lowest execution time per sample (TPS) is approximately obtained at the largest mini-batch size when `baseline-C` is feasible. For example, the Reformer get the lowest TPS 0.869s at the mini-batch size of 2. The reason is that the computation overhead of inverse functions is much larger than the benefit from large mini-batch size. In other words, we cannot accelerate the training process via reversible neural architectures. From the perspective of Problem 8, the TPS $f(x, b) = {t_e^{(1)}}^T x - \frac{C - {t_e^{(0)}}^T x}{b}$ is dominated by the first term ${t_e^{(1)}}^T x$.

5 Conclusions

In this paper, we present the framework to execute reversible neural architectures in the most efficient modes. We formulate the decision problem for reversible operators. The training time is the objective function with memory usage as a constraint. By solving this problem, we can maximize the training speed for any reversible neural architectures. Our framework automates this decision process, empowering researchers to develop and train reversible networks more efficiently.

For future directions, we may integrate gradient checkpoints and reversible neural architectures to enlarge the search space, since gradient checkpoints allow

non-reversible layers to follow M-Mode by doing recomputation. The optimal mini-batch size in terms of training throughput is another critical issue.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), software available from tensorflow.org
2. Blumberg, S.B., Tanno, R., Kokkinos, I., Alexander, D.C.: Deeper image quality transfer: Training low-memory neural networks for 3d images. Lecture Notes in Computer Science p. 118–125 (2018)
3. Brügger, R., Baumgartner, C.F., Konukoglu, E.: A partially reversible u-net for memory-efficient volumetric image segmentation. Medical Image Computing and Computer Assisted Intervention – MICCAI 2019 p. 429–437 (2019)
4. Bulò, S.R., Porzi, L., Kontschieder, P.: In-place activated batchnorm for memory-optimized training of dnns. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 5639–5647 (June 2018). https://doi.org/10.1109/CVPR.2018.00591
5. Chen, T.Q., Rubanova, Y., Bettencourt, J., Duvenaud, D.K.: Neural ordinary differential equations. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 6571–6583. Curran Associates, Inc. (2018)
6. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost (2016)
7. Gomez, A.N., Ren, M., Urtasun, R., Grosse, R.B.: The reversible residual network: Backpropagation without storing activations. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. p. 2211–2221. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)
8. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37. p. 448–456. ICML'15, JMLR.org (2015)
9. Jacobsen, J.H., Smeulders, A.W., Oyallon, E.: i-revnet: Deep invertible networks. In: International Conference on Learning Representations (2018)
10. Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., Stoica, I.: Breaking the memory wall with optimal tensor rematerialization. In: Proceedings of Machine Learning and Systems 2020, pp. 497–511 (2020)
11. Jia, Z., Lin, S., Qi, C.R., Aiken, A.: Exploring hidden dimensions in accelerating convolutional neural networks. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 2274–2283. PMLR, Stockholmsmässan, Stockholm Sweden (10–15 Jul 2018)
12. Kingma, D.P., Dhariwal, P.: Glow: Generative flow with invertible 1x1 convolutions. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 10215–10224. Curran Associates, Inc. (2018)
13. Kitaev, N., Łukasz Kaiser, Levskaya, A.: Reformer: The efficient transformer (2020)

14. Kusumoto, M., Inoue, T., Watanabe, G., Akiba, T., Koyama, M.: A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. In: Advances in Neural Information Processing Systems 32, pp. 1161–1170. Curran Associates, Inc. (2019)
15. Leemput, S.C.v., Teuwen, J., Ginneken, B.v., Manniesing, R.: Memcnn: A python/pytorch package for creating memory-efficient invertible neural networks. Journal of Open Source Software **4**(39), 1576 (2019). https://doi.org/10.21105/joss.01576
16. MacKay, M., Vicol, P., Ba, J., Grosse, R.: Reversible recurrent neural networks. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. p. 9043–9054. NIPS'18, Curran Associates Inc., Red Hook, NY, USA (2018)
17. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc., USA (1990)
18. 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., 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)
19. Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., Keckler, S.W.: Vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-49, IEEE Press (2016)
20. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Neurocomputing: Foundations of research. Nature pp. 696–699 (1988)
21. Sohoni, N.S., Aberger, C.R., Leszczynski, M., Zhang, J., Ré, C.: Low-memory neural network training: A technical report (2019)
22. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30, pp. 5998–6008. Curran Associates, Inc. (2017)
23. Xie, S., Girshick, R., Dollar, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (Jul 2017). https://doi.org/10.1109/cvpr.2017.634
24. Zhang, J., Yeung, S.H., Shu, Y., He, B., Wang, W.: Efficient memory management for gpu-based deep learning systems (2019)
25. Zhu, J., Park, T., Isola, P., Efros, A.A.: Unpaired image-to-image translation using cycle-consistent adversarial networks. In: 2017 IEEE International Conference on Computer Vision (ICCV). pp. 2242–2251 (Oct 2017). https://doi.org/10.1109/ICCV.2017.244