Training Deep Nets with Sublinear Memory Cost

Tianqi Chen [1], **Bing Xu** [2], **Chiyuan Zhang** [3], and **Carlos Guestrin** [1]

[1] Unveristy of Washington [2] Dato. Inc [3] Massachusetts Institute of Technology

Abstract

We propose a systematic approach to reduce the memory consumption of deep neural network training. Specifically, we design an algorithm that costs $O(\sqrt{n})$ memory to train a n layer network, with only the computational cost of an extra forward pass per mini-batch. As many of the state-of-the-art models hit the upper bound of the GPU memory, our algorithm allows deeper and more complex models to be explored, and helps advance the innovations in deep learning research. We focus on reducing the memory cost to store the intermediate feature maps and gradients during training. Computation graph analysis is used for automatic in-place operation and memory sharing optimizations. We show that it is possible to trade computation for memory giving a more memory efficient training algorithm with a little extra computation cost. In the extreme case, our analysis also shows that the memory consumption can be reduced to $O(\log n)$ with as little as $O(n \log n)$ extra cost for forward computation. Our experiments show that we can reduce the memory cost of a 1,000-layer deep residual network from 48G to 7G on ImageNet problems. Similarly, significant memory cost reduction is observed in training complex recurrent neural networks on very long sequences.

1 Introduction

In this paper, we propose a systematic approach to reduce the memory consumption of deep neural network training. We mainly focus on reducing the memory cost to store intermediate results (feature maps) and gradients, as the size of the parameters are relatively small comparing to the size of the intermediate feature maps in many common deep architectures. We use a computation graph analysis to do automatic in-place operation and memory sharing optimizations. More importantly, we propose a novel method to trade computation for memory. As a result, we give a practical algorithm that cost $O(\sqrt{n})$ memory for feature maps to train a n layer network with only double the forward pass computational cost. Interestingly, we also show that in the extreme case, it is possible to use as little as $O(\log n)$ memory for the features maps to train a n layer network.

We have recently witnessed the success of deep neural networks in many domains [8], such as computer vision, speech recognition, natural language processing and reinforcement learning. Many of the success are brought by innovations in new architectures of deep neural networks. Convolutional neural networks [15, 14, 13, 10] model the spatial patterns and give the state of art results in computer vision tasks. Recurrent neural networks, such as long short-term memory [12], show inspiring results in sequence modeling and structure prediction. One common trend in those new models is to use deeper architectures [18, 14, 13, 10] to capture the complex patterns in a large amount of training data. Since the cost of storing feature maps and their gradients scales linearly with the depth of network, our capability of exploring deeper models is limited by the device (usually a GPU) memory. For example, we already run out of memories in one of the current state-of-art models as described in [11]. In the long run, an ideal machine learning system should be able to continuously learn from an increasing amount of training data. Since the optimal model size and complexity often grows with more training data, it is very important to have memory-efficient training algorithms.

Reducing memory consumption not only allows us to train bigger models. It also enables larger batch size for better device utilization and stablity of batchwise operators such as batch normalization [13]. For memory limited devices, it helps improve memory locality and potentially leads to better memory access patterns. It also enables us to switch from model parallelism to data parallelism for training deep convolutional neural networks, which can be beneficial in certain circumstances. Our solution enables us to train deeper convolutional neural networks, as well as recurrent neural networks with longer unrolling steps. We provide guidelines for deep learning frameworks to incorporate the memory optimization techniques proposed in this paper. We will also make our implementation of memory optimization algorithm publicly available.

2 Related Works

We can trace the idea of computational graph and liveness analysis back to the literatures of compiler optimizations [3]. Analogy between optimizing a computer program and optimizing a deep neural network computational graph can be found. For example, memory allocation in deep networks is similar to register allocation in a compiler. The formal analysis of computational graph allows us save memory in a principled way. Theano [5, 4] is a pioneering framework to bring the computation graph to deep learning, which is joined by recently introduced frameworks such as CNTK [2], Tensorflow [1] and MXNet [6]. Theano and Tensorflow use reference count based recycling and runtime garbage collection to manage memory during training, while MXNet uses a static memory allocation strategy prior to the actual computation. However, most of the existing framework focus on graph analysis to optimize computation after the gradient graph is constructed, but do not discuss the computation and memory trade-off.

The trade-off between memory and computation has been a long standing topic in systems research. Although not widely known, the idea of dropping intermediate results is also known as gradient checkpointing technique in automatic differentiation literature [9]. We bring this idea to neural network gradient graph construction for general deep neural networks. Through the discussion with our colleagues [19], we know that the idea of dropping computation has been applied in some limited specific use-cases. In this paper, we propose a general methodology that works for general deep neural networks, including both convolutional and recurrent neural networks. Our results show that it is possible to train a general deep neural network with sublinear memory cost. More importantly, we propose an automatic planning algorithm to provide a good memory plan for real use-cases. The proposed gradient graph optimization algorithm can be readily *combined with all the existing memory optimizations* in the computational graph to further reduce the memory consumption of deep learning frameworks.

There are other ways to train big models, such as swapping of CPU/GPU memory and use of model parallel training [7, 16]. These are orthogonal approaches and can be used together with our algorithm to train even bigger models with fewer resources. Moreover, our algorithm does not need additional communication over PCI-E and can save the bandwidth for model/data parallel training.

3 Memory Optimization with Computation Graph

We start by reviewing the concept of computation graph and the memory optimization techniques. Some of these techniques are already used by existing frameworks such as Theano [5, 4], Tensorflow [1] and MXNet [6]. A computation graph consists of operational nodes and edges that represent the dependencies between the operations. Fig. 1 gives an example of the computation graph of a two-layer fully connected neural network. Here we use coarse grained forward and backward operations to make the graph simpler. We further simplify the graph by hiding the weight nodes and gradients of the weights. A computation graph used in practice can be more complicated and contains mixture of fine/coarse grained operations. The analysis presented in this paper can be directly used in those more general cases.

Once the network configuration (forward graph) is given, we can construct the corresponding backward pathway for gradient calculation. A backward pathway can be constructed by traversing

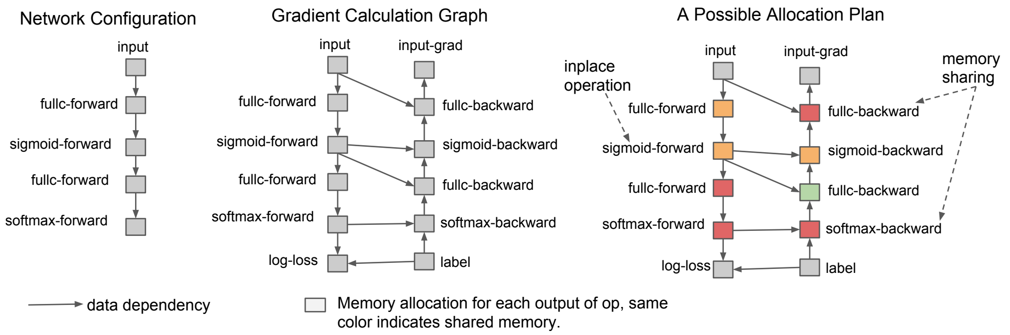


Figure 1: Computation graph and possible memory allocation plan of a two layer fully connected neural network training procedure. Each node represents an operation and each edge represents a dependency between the operations. The nodes with the same color share the memory to store output or back-propagated gradient in each operator. To make the graph more clearly, we omit the weights and their output gradient nodes from the graph and assume that the gradient of weights are also calculated during backward operations. We also annotate two places where the in-place and sharing strategies are used.

the configuration in reverse topological order, and apply the backward operators as in normal back-propagation algorithm. The backward pathway in Fig. 1 represents the gradient calculation steps *explicitly*, so that the gradient calculation step in training is simplified to just a forward pass on the entire computation graph (including the gradient calculation pathway). Explicit gradient path also offers some other benefits (e.g. being able to calculate higher order gradients), which is beyond our scope and will not be covered in this paper.

When training a deep convolutional/recurrent network, a great proportion of the memory is usually used to store the intermediate outputs and gradients. Each of these intermediate results corresponds to a node in the graph. A smart allocation algorithm is able to assign the least amount of memory to these nodes by sharing memory when possible. Fig. 1 shows a possible allocation plan of the example two-layer neural network. Two types of memory optimizations can be used

- *Inplace operation*: Directly store the output values to memory of a input value.
- *Memory sharing*: Memory used by intermediate results that are no longer needed can be recycled and used in another node.

Allocation plan in Fig. 1 contains examples of both cases. The first sigmoid transformation is carried out using inplace operation to save memory, which is then reused by its backward operation. The storage of the softmax gradient is shared with the gradient by the first fully connected layer. Ad hoc application of these optimizations can leads to errors. For example, if the input of an operation is still needed by another operation, applying inplace operation on the input will lead to a wrong result.

We can only share memory between the nodes whose lifetime do not overlap. There are multiple ways to solve this problem. One option is to construct the conflicting graph of with each variable as node and edges between variables with overlapping lifespan and then run a graph-coloring algorithm. This will cost $O(n^2)$ computation time. We adopt a simpler heuristic with only $O(n)$ time. The algorithm is demonstrated in Fig. 2. It traverses the graph in topological order, and uses a counter to indicate the liveness of each record. An inplace operation can happen when there is no other pending operations that depend on its input. Memory sharing happens when a recycled tag is used by another node. This can also serve as a dynamic runtime algorithm that traverses the graph, and use a garbage collector to recycle the outdated memory. We use this as a static memory allocation algorithm, to allocate the memory to each node before the execution starts, in order to avoid the overhead of garbage collection during runtime.

Guidelines for Deep Learning Frameworks As we can see from the algorithm demonstration graph in Fig. 2. The data dependency causes longer lifespan of each output and increases the memory

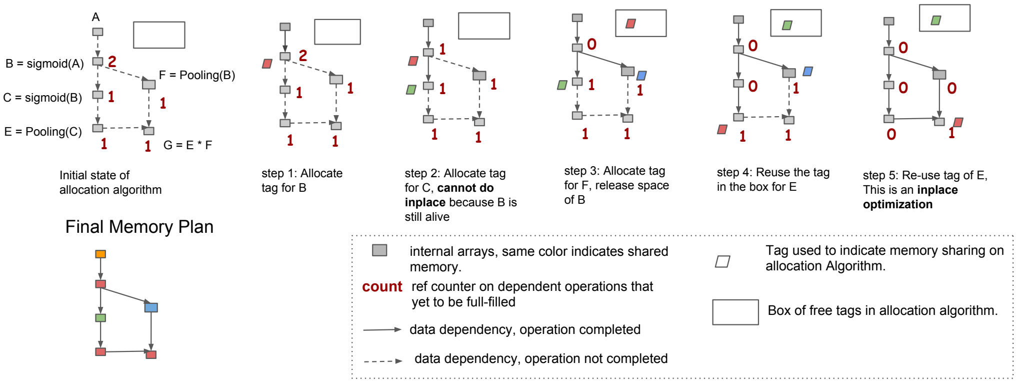

A
B = sigmoid(A)
C = sigmoid(B)
E = Pooling(C)
F = Pooling(B)
G = E * F
2
1
1
1
1
Initial state of allocation algorithm
2
1
1
1
1
step 1: Allocate tag for B
1
1
1
1
1
step 2: Allocate tag for C, **cannot do inplace** because B is still alive
0
1
1
1
1
step 3: Allocate tag for F, release space of B
0
0
1
1
1
step 4: Reuse the tag in the box for E
0
0
0
0
1
step 5: Re-use tag of E, This is an **inplace optimization**
Final Memory Plan
internal arrays, same color indicates shared memory.
count
ref counter on dependent operations that yet to be full-filled
data dependency, operation completed
data dependency, operation not completed
Tag used to indicate memory sharing on allocation Algorithm.
Box of free tags in allocation algorithm.

Figure 2: Memory allocation algorithm on computation graph. Each node associated with a liveness counter to count on operations to be full-filled. A temporal tag is used to indicate memory sharing. Inplace operation can be carried out when the current operations is the only one left (input of counter equals 1). The tag of a node can be recycled when the node's counter goes to zero.

consumption of big network. It is important for deep learning frameworks to

- Declare the dependency requirements of gradient operators in minimum manner.
- Apply liveness analysis on the dependency information and enable memory sharing.

It is important to declare minimum dependencies. For example, the allocation plan in Fig. 1 won't be possible if `sigmoid-backward` also depend on the output of the first `fullc-forward`. The dependency analysis can usually reduce the memory footprint of deep network prediction of a n layer network from $O(n)$ to nearly $O(1)$ because sharing can be done between each intermediate results. The technique also helps to reduce the memory footprint of training, although only up to a constant factor.

4 Trade Computation for Memory

4.1 General Methodology

The techniques introduced in Sec. 3 can reduce the memory footprint for both training and prediction of deep neural networks. However, due to the fact that most gradient operators will depend on the intermediate results of the forward pass, we still need $O(n)$ memory for intermediate results to train a n layer convolutional network or a recurrent neural networks with a sequence of length n. In order to further reduce the memory, we propose to *drop some of the intermediate results*, and recover them from an extra forward computation when needed.

More specifically, during the backpropagation phase, we can re-compute the dropped intermediate results by running forward from the closest recorded results. To present the idea more clearly, we show a simplified algorithm for a linear chain feed-forward neural network in Alg. 1. Specifically, the neural network is divided into several segments. The algorithm only remembers the output of each segment and drops all the intermediate results within each segment. The dropped results are recomputed at the segment level during back-propagation. As a result, we only need to pay the memory cost to store the outputs of each segment plus the maximum memory cost to do backpropagation on each segment.

Alg. 1 can also be generalized to common computation graphs as long as we can divide the graph into segments. However, there are two drawbacks on directly applying Alg. 1: 1) users have to manually divide the graph and write customized training loop; 2) we cannot benefit from other memory optimizations presented in Sec 3. We solve this problem by introducing a general gradient graph construction algorithm that uses essentially the same idea. The algorithm is given in Alg. 2. In this algorithm, the user specify a function $m : \mathcal{V} \rightarrow \mathbb{N}$ on the nodes of a computation graph

Algorithm 1: Backpropagation with Data Dropping in a Linear Chain Network

```
$v \leftarrow input$
for $k = 1$ to $length(segments)$ do
    $temp[k] \leftarrow v$
    for $i = segments[k].begin$ to $segments[k].end - 1$ do
        $v \leftarrow layer[i].forward(v)$
    end
end
$g \leftarrow gradient(v, label)$
for $k = length(segments)$ to $1$ do
    $v \leftarrow temp[k]$
    $localtemp \leftarrow$ empty hashtable
    for $i = segments[k].begin$ to $segments[k].end - 1$ do
        $localtemp[i] \leftarrow v$
        $v \leftarrow layer[i].forward(v)$
    end
    for $i = segments[k].end - 1$ to $segments[k].begin$ do
        $g \leftarrow layer[i].backward(g, localtemp[i])$
    end
end
```

to indicate how many times a result can be recomputed. We call m the mirror count function as the re-computation is essentially duplicating (mirroring) the nodes. When all the mirror counts are set to 0, the algorithm degenerates to normal gradient graph. To specify re-computation pattern in Alg. 2, the user only needs to set the $m(v) = 1$ for nodes within each segment and $m(v) = 0$ for the output node of each segment. The mirror count can also be larger than 1, which leads to a recursive generalization to be discussed in Sec 4.4. Fig. 3 shows an example of memory optimized gradient graph. Importantly, Alg. 2 also outputs a traversal order for the computation, so the memory usage can be optimized. Moreover, this traversal order can help introduce control flow dependencies for frameworks that depend on runtime allocation.

4.2 Drop the Results of Low Cost Operations

One quick application of the general methodology is to drop the results of low cost operations and keep the results that are time consuming to compute. This is usually useful in a `Conv-BatchNorm-Activation` pipeline in convolutional neural networks. We can always keep the result of convolution, but drop the result of the batch normalization, activation function and pooling. In practice this will translate to a memory saving with little computation overhead, as the computation for both batch normalization and activation functions are cheap.

4.3 An $O(\sqrt{n})$ Memory Cost Algorithm

Alg. 2 provides a general way to trade computation for memory. It remains to ask which intermediate result we should keep and which ones to re-compute. Assume we divide the n network into k segments the memory cost to train this network is given as follows.

$$\text{cost-total} = \max_{i=1,\ldots,k} \text{cost-of-segment}(i) + O(k) = O\left(\frac{n}{k}\right) + O(k) \quad (1)$$

The first part of the equation is the memory cost to run back-propagation on each of the segment. Given that the segment is equally divided, this translates into $O(n/k)$ cost. The second part of equation is the cost to store the intermediate outputs between segments. Setting $k = \sqrt{n}$, we get the cost of $O(2\sqrt{n})$. This algorithm *only requires an additional forward pass* during training, but

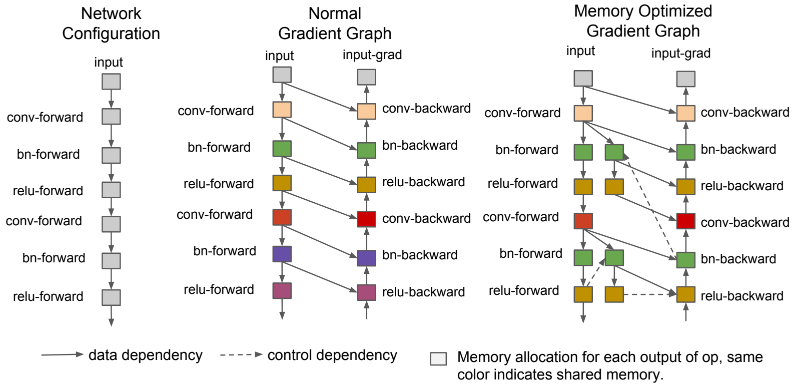


Figure 3: Memory optimized gradient graph generation example. The forward path is *mirrored* to represent the re-computation happened at gradient calculation. User specifies the mirror factor to control whether a result should be dropped or kept.

```
Algorithm 2: Memory Optimized Gradient Graph Construction
Input: $G = (V, \text{pred})$, input computation graph, the $\text{pred}[v]$ gives the predecessors array of
    node $v$.
Input: gradient(succ_grads, output, inputs), symbolic gradient function that creates a gradient
    node given successor gradients and output and inputs
Input: $m : V \to \mathbb{N}^+$, $m(v)$ gives how many time node $v$ should be duplicated, $m(v) = 0$
    means do no drop output of node $v$.
$a[v] \leftarrow v$ for $v \in V$
for $k = 1$ to $\max_{v \in V} m(v)$ do
    for $v$ in topological-order$(V)$ do
        if $k \leq m(v)$ then
            $a[v] \leftarrow$ new node, same operator as v
            $\text{pred}[a[v]] \leftarrow \cup_{u \in \text{pred}[v]} \{a[u]\}$
        end
    end
end
$V' \leftarrow$ topological-order$(V)$
for $v$ in reverse-topological-order$(V)$ do
    $g[v] \leftarrow gradient([g[v]$ for v in $successor(v)], a[v], [a[v]$ for v in $pred[v]])$
    $V' \leftarrow append(V'$, topological-order$(acenstors(g[v])) - V')$
end
Output: $G' = (V', \text{pred})$ the new graph, the order in $V'$ gives the logical execution order.
```

reduces the memory cost to be *sub-linear*. Since the backward operation is nearly twice as time consuming as the forward one, it only slows down the computation by a small amount.

In the most general case, the memory cost of each layer is not the same, so we cannot simply set $k = \sqrt{n}$. However, the trade-off between the intermediate outputs and the cost of each stage still holds. In this case, we use Alg. 3 to do a greedy allocation with a given budget for the memory cost within each segment as a single parameter B. Varying B gives us various allocation plans that either assign more memory to the intermediate outputs, or to computation within each stage. When we do static memory allocation, we can get the *exact memory cost* given each allocation plan. We can use this information to do a heuristic search over B to find optimal memory plan that balances the cost of the two. The details of the searching step is presented in the supplementary material. We find this approach works well in practice. We can also generalize this algorithm by considering the cost to run each operation to try to keep time consuming operations when possible.

```
Algorithm 3: Memory Planning with Budget
  Input: $G = (V, \text{pred})$, input computation graph.
  Input: $C \subset V$, candidate stage splitting points, we will search splitting points over $v \subset C$
  Input: $B$, approximate memory budget. We can search over $B$ to optimize the memory
         allocation.
  $temp \leftarrow 0, x \leftarrow 0, y \leftarrow 0$
  for $v$ in topological-order$(V)$ do
      $temp \leftarrow temp + \text{size-of-output}(v)$
      if $v \in C$ and $temp > B$ then
          $x \leftarrow x + \text{size-of-output}(v)$, $y \leftarrow max(y, temp)$
          $m(v) = 0$, $temp \leftarrow 0$
      else
          $m(v) = 1$
      end
  end
  Output: $x$ approximate cost to store inter-stage feature maps
  Output: $y$ approximate memory cost for each sub stage
  Output: $m$ the mirror plan to feed to Alg. 2
```

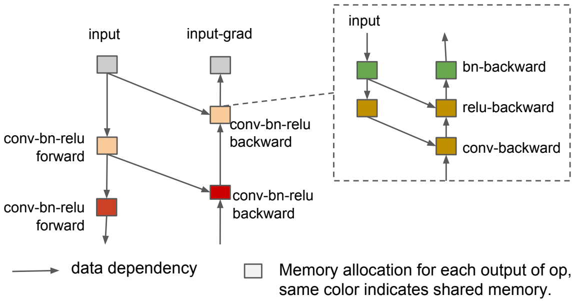


Figure 4: Recursion view of the memory optimized allocations. The segment can be viewed as a single operator that combines all the operators within the segment. Inside each operator, a sub-graph as executed to calculate the gradient.

4.4 More General View: Recursion and Subroutine

In this section, we provide an alternative view of the memory optimization scheme described above. Specifically, we can view each segment as a bulk operator that combines all the operations inside the segment together. The idea is illustrated in Fig. 4. The combined operator calculates the gradient by executing over the sub-graph that describes its internal computation. This view allows us to treat a series of operations as subroutines. The optimization within the sub-graph does not affect the external world. As a result, we can recursively apply our memory optimization scheme to each sub-graph.

Pay Even Less Memory with Recursion Let $g(n)$ to be the memory cost to do forward and backward pass on a n layer neural network. Assume that we store k intermediate results in the graph and apply the same strategy recursively when doing forward and backward pass on the sub-path. We have the following recursion formula.

$$g(n) = k + g\left(n/(k+1)\right) \tag{2}$$

Solving this recursion formula gives us

$$g(n) = k \log_{k+1}(n) \tag{3}$$

As a special case, if we set $k = 1$, we get $g(n) = \log_2 n$. This is interesting conclusion as all the existing implementations takes $O(n)$ memory in feature map to train a n layer neural network. This will require $O(\log_2 n)$ cost forward pass cost, so may not be used commonly. But it demonstrates how we can trade memory even further by using recursion.

4.5 Guideline for Deep Learning Frameworks

In this section, we have shown that it is possible to trade computation for memory and combine it with the system optimizations proposed in Sec 3. It is helpful for deep learning frameworks to

- Enable option to drop result of low cost operations.
- Provide planning algorithms to give efficient memory plan.
- Enable user to set the mirror attribute in the computation graph for memory optimization.

While the last option is not strictly necessary, providing such interface enables user to hack their own memory optimizers and encourages future researches on the related directions. Under this spirit, we support the customization of graph mirror plan and will make the source code publicly available.

5 Experiments

5.1 Experiment Setup

We evaluate the memory cost of storing intermediate feature maps using the methods described in this paper. We our method on top of MXNet [6], which statically allocate all the intermediate feature maps before computation. This enables us to report the *exact memory cost* spend on feature maps. Note that the memory cost of parameters and temporal memory (e.g. required by convolution) are not part of the memory cost report. We also record the runtime total memory cost by running training steps on a Titan X GPU. Note that all the memory optimizations proposed in this paper gives equivalent weight gradient for training and can always be safely applied. We compare the following memory allocation algorithms

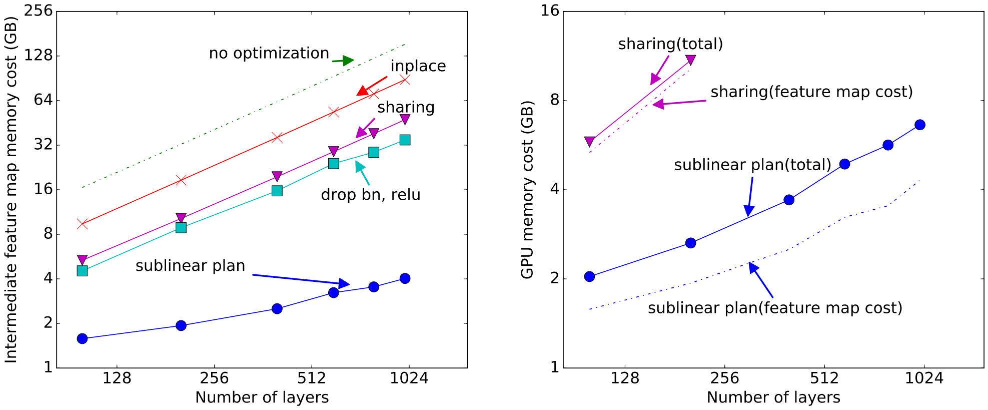
- *no optimization*, directly allocate memory to each node in the graph without any optimization.
- *inplace*, enable inplace optimization when possible.
- *sharing*, enable inplace optimization as well as sharing. This represents all the system optimizations presented at Sec. 3.
- *drop bn-relu*, apply all system optimizations, drop result of batch norm and relu, this is only shown in convolutional net benchmark.
- *sublinear plan*, apply all system optimizations, use plan search with Alg 3 to trade computation with memory.

5.2 Deep Convolutional Network

We first evaluate the proposed method on convolutional neural network for image classification. We use deep residual network architecture [11] (ResNet), which gives the state of art result on this task. Specifically, we use 32 batch size and set input image shape as $(3, 224, 224)$. We generate different depth configuration of ResNet [1] by increasing the depth of each residual stage.

We show the results in Fig. 5. We can find that the system optimizations introduced in Sec. 3 can help to reduce the memory cost by factor of two to three. However, the memory cost after optimization still exhibits a linear trend with respect to number of layers. Even with all the system optimizations, it is only possible to train a 200 layer ResNet with the best GPU we can get. On the other hand, the proposed algorithm gives a sub-linear trend in terms of number of layers. By trade computation with memory, we can train a 1000 layer ResNet using less than 7GB of GPU memory.

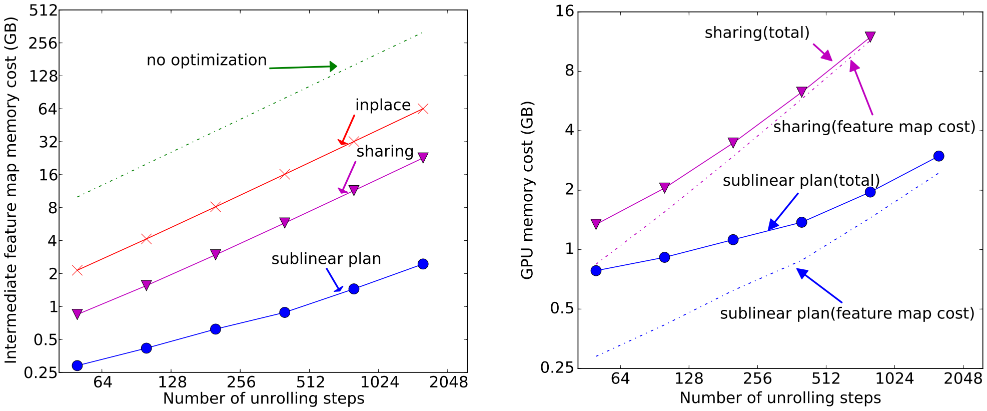
[1] We count a conv-bn-relu as one layer



(a) Feature map memory cost estimation

(b) Runtime total memory cost

Figure 5: The memory cost of different allocation strategies on deep residual net configurations. The feature map memory cost is generated from static memory allocation plan. We also use nvidia-smi to measure the total memory cost during runtime (the missing points are due to out of memory). The figures are in log-scale, so $y = \alpha x^{\beta}$ will translate to $\log(y) = \beta \log(x) + \log \alpha$. We can find that the graph based allocation strategy indeed help to reduce the memory cost by a factor of two to three. More importantly, the sub-linear planning algorithm indeed gives sub-linear memory trend with respect to the workload. The real runtime result also confirms that we can use our method to greatly reduce memory cost deep net training.



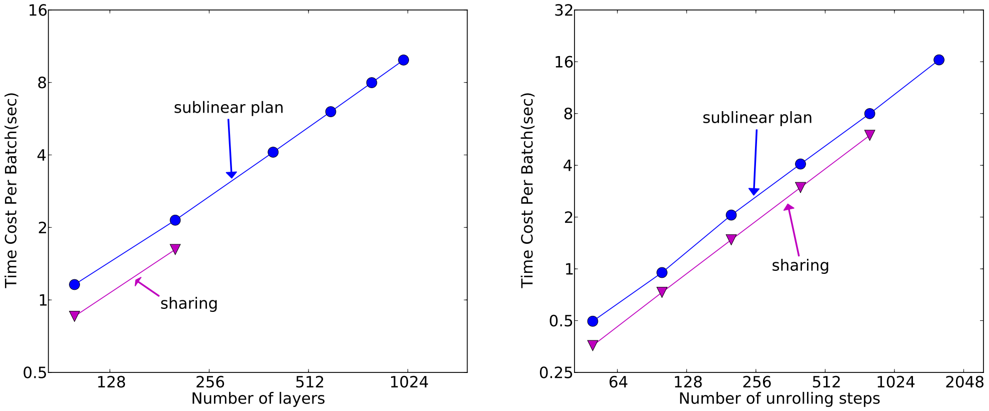
(a) Feature map memory cost estimation

(b) Runtime total memory cost

Figure 6: The memory cost of different memory allocation strategies on LSTM configurations. System optimization gives a lot of memory saving on the LSTM graph, which contains a lot of fine grained operations. The sub-linear plan can give more than 4x reduction over the optimized plan that do not trade computation with memory.

5.3 LSTM for Long Sequences

We also evaluate the algorithms on a LSTM under a long sequence unrolling setting. We unrolled a four layer LSTM with 1024 hidden states equals 64 over time. The batch size is set to 64. The input of each timestamp is a continuous 50 dimension vector and the output is softmax over 5000 class. This is a typical setting for speech recognition[17], but our result can also be generalized to other recurrent networks. Using a long unrolling step can potentially help recurrent model to learn long



(a) ResNet

(b) LSTM

Figure 7: The runtime speed of different allocation strategy on the two settings. The speed is measured by a running 20 batches on a Titan X GPU. We can see that using sub-linear memory plan incurs roughly 30% of additional runtime cost compared to linear memory allocation. The general trend of speed vs workload remains linear for both strategies.

term dependencies over time. We show the results in Fig. 6. We can find that inplace helps a lot here. This is because inplace optimization in our experiment enables direct addition of weight gradient to a single memory cell, preventing allocate space for gradient at each timestamp. The sub-linear plan gives more than 4x reduction over the optimized memory plan.

5.4 Impact on Training Speed

We also measure the runtime cost of each strategy. The speed is benchmarked on a single Titan X GPU. The results are shown in Fig. 7. Because of the double forward cost in gradient calculation, the sublinear allocation strategy costs 30% additional runtime compared to the normal strategy. By paying the small price, we are now able to train a much wider range of deep learning models.

6 Conclusion

In this paper, we proposed a systematic approach to reduce the memory consumption of the intermediate feature maps when training deep neural networks. Computation graph liveness analysis is used to enable memory sharing between feature maps. We also showed that we can trade the computation with the memory. By combining the techniques, we can train a n layer deep neural network with only $O(\sqrt{n})$ memory cost, by paying nothing more than one extra forward computation per mini-batch.

Acknowledgement

We thank the helpful feedbacks from the MXNet community and developers. We thank Ian Goodfellow and Yu Zhang on helpful discussions on computation memory tradeoffs. We would like to thank David Warde-Farley for pointing out the relation to gradient checkpointing. We would like to thank Nvidia for the hardware support. This work was supported in part by ONR (PECASE) N000141010672, NSF IIS 1258741 and the TerraSwarm Research Center sponsored by MARCO and DARPA. Chiyuan Zhang acknowledges the support of a Nuance Foundation Grant.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Yongqiang Wang, Huaming Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, August 2014.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, , and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems (LearningSys'15)*, 2015.

[7] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[8] Ian Goodfellow, Yoshua Bengio, , and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[9] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, March 2000.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32th International Conference on Machine Learning (ICML'15)*, 2015.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.

[15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In S. Haykin and B. Kosko, editors, *Intelligent Signal Processing*, pages 306–351. IEEE Press, 2001.

[16] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. Virtualizing deep neural networks for memory-efficient neural network design. *arXiv preprint arXiv:1602.08124*, 2016.

[17] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 338–342, 2014.

[18] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. *arXiv preprint arXiv:1507.06228*, 2015.

[19] Yu Zhang, Guoguo Chen, Dong Yu, Kaisheng Yao, Sanjeev Khudanpur, and James Glass. Highway long short-term memory rnns for distant speech recognition. *arXiv preprint arXiv:1510.08983*, 2015.

A Search over Budget B

Alg. 3 allows us to generate an optimized memory plan given a single parameter B. This algorithm relies on approximate memory estimation for faster speed. After we get the plan, we can use the static allocation algorithm to calculate the exact memory cost. We can then do a grid search over B to find a good memory plan.

To get the setting of the grid, we first run the allocation algorithm with $B = 0$, then run the allocation algorithm again with $B = \sqrt{xy}$. Here x and y are the outputs from Alg. 3 in the first run. Here x is the approximate cost to store inter-stage feature maps and y is the approximate cost to run each stage. $B = \sqrt{xy}$ an estimation of each stage's memory cost. This can already give a good memory plan. We then set grid around $B = \sqrt{xy}$ to further refine the solution.

In practice, we find that using a size 6 grid on $[B/\sqrt{2}, \sqrt{2}B]$ can already give good memory plans in the experiments. We implemented the allocation algorithm in python without any attempt to optimize for speed. Our code costs a few seconds to get the plans needed in the experiments.