GPU Memory Management for Deep Neural Networks Using Deep Q-Network

Anonymous authors
Paper under double-blind review

Abstract

Deep neural networks use deeper and broader structures to achieve better performance and consequently, use increasingly more GPU memory as well. However, limited GPU memory restricts many potential designs of neural networks. In this paper, we propose a reinforcement learning based variable swapping and recomputation algorithm to reduce the memory cost, without sacrificing the accuracy of models. Variable swapping can transfer variables between CPU and GPU memory to reduce variables stored in GPU memory. Recomputation can trade time for space by removing some feature maps during forward propagation. Forward functions are executed once again to get the feature maps before reuse. However, how to automatically decide which variables to be swapped or recomputed remains a challenging problem. To address this issue, we propose to use a deep Q-network(DQN) to make plans. By combining variable swapping and recomputation, our results outperform several well-known benchmarks.

1 Introduction

Limited GPU memory restricts model performance due to two different reasons. Firstly, there is a trend that deep neural networks (DNNs) use deeper and more GPU memory-intensive structures(Wang et al., 2018), and have continuously made improvement in various computer vision areas such as image classification, object detection, and semantic segmentation(He et al., 2016a; Simonyan & Zisserman, 2014; Krizhevsky et al., 2012; Ronneberger et al., 2015; Goodfellow et al., 2016; Szegedy et al., 2015). Likewise, empirical results show that deeper networks can achieve higher accuracy (He et al., 2016b; Urban et al., 2016). Deeper network means higher consumption of GPU memory. Secondly, He et al. (2019) shows that bigger input batch size can speed up the training process and achieve higher accuracy. However, a bigger input batch size requires more GPU memory to store intermediate variables. We want more GPU memory to get better performance.

The rationale to utilize CPU memory by offloading, and later prefetching variables from it is twofold. Firstly, the size of the CPU memory is usually bigger than that of GPU memory. If we do not use variable swapping, all the tensors will stay in GPU memory. Figure 1 shows the details of variable swapping. Secondly, due to the availability of the GPU direct memory access (DMA) engines, which can overlap data transfers with kernel execution. More specifically, a GPU engine is an independent unit which can operate or be scheduled in parallel with other engines. DMA engines control data transfers, and kernel engines can execute different layer functions of DNNs. Hence, in the ideal case, we can completely overlap DNNs training with variable swapping. Therefore, variable swapping is efficient.

Regarding recomputation, some feature maps are not stored in GPU memory in forward propagation, but the feature maps are gotten by running forward functions in backpropagation, as shown in Figure 2. Why do we combine swapping with recomputation? Because recomputation uses GPU computing engines to reduce memory usage, and variable swapping uses DMA engines to save memory. Different engines can run parallelly. If we execute recomputation during data transfers, we will not waste computing engines or DMA engines.

It is hard to decide which variables should be swapped or recomputed. Different DNNs have different structures. Networks have thousands of variables during training, so it is intractable to enumerate the search space exhaustively. Some existing works use heuristic search algorithms for recompu-

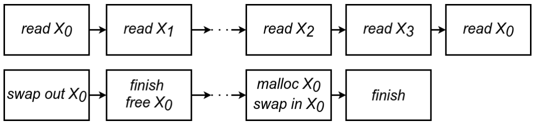


Figure 1: The upper graph shows GPU operations in a standard neural network in a time sequence. The lower one shows how to add variable swapping operations. The nodes in the same column represent they occur at the same time. We copy X_0 into CPU memory while $reading\ X_0$. After data transfer and reading, we $free\ X_0$ from GPU memory. Before using X_0 again, we allocate space for X_0 and transfer it back to GPU memory.

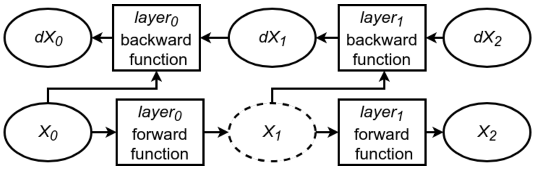

dX_0
$layer_0$ backward function
dX_1
$layer_1$ backward function
dX_2
X_0
$layer_0$ forward function
X_1
$layer_1$ forward function
X_2

Figure 2: If we do not store X_1 in the memory in the forward propagation, we need to execute the $layer_0$ forward function again to get X_1 for the $layer_1$ backward function.

tation or swapping with limited information from computational graphs. For example, they do not consider the time cost of recomputing different layers or swapping different variables. Additionally, they do not make plans for recomputation during swapping in order to increase GPU utilization. Our work utilizes more information from computational graphs than theirs and makes plans automatically for users.

The contribution of our paper is that we propose a DQN algorithm to make plans for swapping and recomputation to reduce memory usage of DNNs. Users only need to set memory usage limits and do not require background knowledge on DNNs. Additionally, the variable swapping and recomputation will not decrease the accuracy of networks.

2 Related Work

Variable swapping is widely used in DNNs for GPU memory management. Rhu et al. (2016) uses a greedy algorithm for swapping, which may be myopic. Le et al. (2018) uses a heuristic algorithm to decide on which variables to be offloaded. However, users are required to decide the number of variables to be swapped. Besides, they cannot devise plans automatically given different memory limits. Wang et al. (2018) uses a least recently used (LRU) algorithm, which does not take advantage of the iterative nature of neural networks. Our method makes use of more information from computation graphs and provides plans automatically.

Recomputation can trade space for time. Chen et al. (2016) proposes recomputation, in-place operation, and memory sharing. They mentioned a grid search method for recomputation when given a memory limit. However, the time cost of recomputation does not become a concern to them. Meng et al. (2017) designs a strategy only for recomputing attention structure. Gomez et al. (2017); MacKay et al. (2018) propose reversible networks. However, they cannot make different plans given different memory limits. Wang et al. (2018) selects some specific low-cost layers for recomputation. However, they do not utilize other types of layers. Moreover, recomputing some layers introduces many allocation operations during backward propagation, which can influence the memory load of DNNs. Our method considers the above challenges by using DQN to make plans for recomputation.

3 Problem Definition

Our major problem is to minimize the computation overhead with a GPU memory limit.

Let $\mathbb{O} = (o_0, o_1, o_2, ..., o_n)$ be a sequence of GPU operations in a training iteration, where o_i denotes the i_{th} GPU operation. GPU operations include four types: $malloc$, $free$, $read$, and $write$. Let m_i be the GPU memory usage from o_0 to o_i. $t_{\mathbb{O}}$ is the the overall execution time of $\mathbb{O}$. We have several

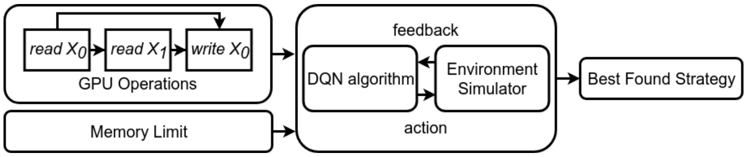


Figure 3: The inputs are a sequence of GPU operations and a memory limit from a user. The algorithm gets the inputs and interacts with an environment simulator. After a few optimization steps, the algorithm gives a best-found strategy which minimizes the training time and subjects to the memory limit.

choices between offloading (swapping out) a variable from GPU memory, prefetching (swapping in) a variable, removing a variable (the first phase of recomputation) from GPU memory, recomputing a variable (the second phase of recomputation), and doing nothing. In each GPU operation o_i, we have a choice to make sure that $max_{i\in\{0,\ldots,n\}}m_i$ is less than the memory limit and use as little $t_{\mathbb{O}}$ as possible. Figure 3 shows the overall algorithm.

4 Reinforcement Learning

4.1 Overview

We focus on optimizing DNNs and some machine learning algorithms such as K-means with iterative nature. We first train DNNs or machine learning algorithms for a few iterations and collect outputs of GPU operations. By utilizing the sequence of GPU operations and memory limit provided by users, we can use DQN to find an optimal plan to swap and recompute. Finally, we train the DNNs or machine learning algorithms following the plan. Our algorithm is an offline version since we want to exploit the iterative nature of DNNs to get more information.

In each GPU operation, we need to choose an action. Actions include three types: swapping, recomputation, and doing nothing. Which action should we choose? We can use Q-learning to solve the problem. We cannot enumerate all states and find their corresponding Q-values since even a small network has hundreds of candidate variables. Hence we use deep Q-network, which learns to approximate Q-values from states by a neural network and chooses actions following the Q-values. We need to swap and recompute to make the memory usage not exceed the memory limit set by the user. The reward is related to the performance overhead of swapping or recomputation.

4.2 Deep Q-network

Let us introduce GPU operations. Operations include four types such as *malloc*, *write*, *read*, and *free*. The beginning time of each operation is known to us. Each operation contains a variable and the address and the size of the variable. We can view each GPU operation as a node in the graph. If two rows are continuous or use the same variable, we will add an edge to these two nodes. The weight of the edge is the time lag between the two nodes.

We need to create agent states for DQN. Here are four types of information that we should record: the variable which is being executed, variables which can be swapped or recomputed, the attributes of the variables, various structures of DNNs. The first two types of information can change while the agent state changes. However, the last two will not change when actions are applied to the agent. We map the structures of DNNs as well as other information into vectors as agent states. We will introduce a representation of the state of each node, and then combine all node states into agent states (graph states).

4.3 Common Formulation

Before continuing, let us list some of the necessary notations.

- $\boldsymbol{s}_v$ is the state of node v, where $v \in \mathbb{V}$. $\mathbb{V}$ is the set of nodes. $\mathbb{S}$ includes all node states.

- $w(u, v)$ is the weighted edge between node u and v, and its value is the time lag between u and v. Each node represents a GPU operation.
- $u \in \mathcal{N}(v)$ means that there is an edge between node u and v, or $u = v$.
- The parameter matrix $\boldsymbol{W}$ can be learned.
- $\mathbb{H}^0$ and $\mathbb{H}^1$ include all variables which can be offloaded and recomputed in the current state respectively.
- $[,]$ joins a sequence of matrices along the first dimension.

4.4 State

4.4.1 Node State

As shown in the following Equation 1. $\boldsymbol{s}_v^t$ means the state of node v, where $v \in \mathbb{V}$. Each node state $\boldsymbol{s}_v^t$ contains its at most $t - 1$ hop neighbor information, node features, and edges information between itself and its neighbors. $\mathbb{S}^t$ includes all node states in t iteration: $\boldsymbol{s}_0^t$, $\boldsymbol{s}_1^t$, $\boldsymbol{s}_3^t$... We first initialize node set $\mathbb{S}^0$ to zero and then use the Equation 1 to update $\mathbb{S}^1$ by $\mathbb{S}^0$. Now, $\boldsymbol{s}_v^1$ only has the information of node v. We update $\mathbb{S}^2$, $\mathbb{S}^3$, and $\mathbb{S}^4$ until $\mathbb{S}^T$ in sequence. The number of iterations T for each node is usually small, such as $T = 4$, which is inspired by Dai et al. (2017).

$$\boldsymbol{s}_v^{t+1} = ReLU(\boldsymbol{W}_1\boldsymbol{x}_v + \boldsymbol{W}_2\Sigma_{u\in\mathcal{N}(v)}\boldsymbol{s}_u^t + \boldsymbol{W}_3\Sigma_{u\in\mathcal{N}(v)}ReLU(\boldsymbol{W}_4 w(u,v))) \tag{1}$$

where $\boldsymbol{W}_1 \in R^{p\times 6}$, $\boldsymbol{W}_2, \boldsymbol{W}_3 \in R^{p\times p}$, $\boldsymbol{W}_4 \in R^{p\times 1}$, $\boldsymbol{s}_v^t \in R^{p\times 1}$, and $\boldsymbol{x}_v \in R^{6\times 1}$. $\boldsymbol{x}_v$ includes six features: the size of the variable operated in node v, the duration of the variable transferring between GPU memory and CPU memory, the duration of the variable recomputing, how soon the variable will be revisited, the action type of the node (Section 4.5), whether it is valid to execute the action in node v.

One reason for adding neighbor nodes and edges is that adding operation is invariant to the different order over neighbors. The other reason is that we can use the same length of the node vectors for different DNN structures.

4.4.2 Graph State

The graph state concatenates the summation over each node state with the state of the node which is under execution.

$$\boldsymbol{g} = [\boldsymbol{W}_5\Sigma_{u\in\mathbb{V}}\boldsymbol{s}_u^T, \boldsymbol{W}_6\boldsymbol{s}_c^T] \tag{2}$$

where $\boldsymbol{W}_5, \boldsymbol{W}_6 \in R^{p\times p}$. $\boldsymbol{s}_c$ is a node state which indicates that node c is under execution.

Because we add all the state of nodes together, the graph feature is not related to the number of nodes. An advantage of using such graph feature representation is that we can train a DQN on a small graph and fine-tune on a large graph.

4.5 Actions

The actions a include three types: offloading a variable in the set $\mathbb{H}^0$ into CPU memory, removing a variable from the set $\mathbb{H}^1$ during forward propagation, doing nothing (Figure 4). Note, variable swapping includes two phases: swapping out a variable and swapping in a variable. Recomputation also includes two phases: removing a variable in forward propagation and recomputing functions to get the removed variable during backpropagation. Actions only include the first phase for variable swapping and recomputation, so we call them the swap-out action and the removing action.

There is no swap-in action or the second phase recomputation action since when the variable is in CPU memory, the optimal swap-in or the second phase recomputation timing is fixed, no need for including into actions. As to prefetching, We first prefetch the earliest reused variable which is not in the GPU memory and then the second earliest reused variable. If swapping in a variable does not cause any excess of memory usage until the end of the backpropagation, we will begin to prefetch

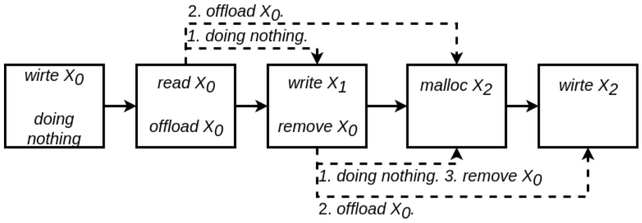


Figure 4: Solid and dashed lines represent time relation between nodes and agent transition (Section 4.6) respectively. Each node represents exactly one GPU operation and will be executed in a time sequence. Each node represents at most one action, and not all of the actions will be executed. In each node, we can choose one action from several candidate actions. For example, we can choose to do nothing, remove X_0, or offload X_0 in the 3_{th} node.

the variable. If a GPU operation requires a variable, we need to suspend GPU operations before finishing prefetching the variable. We use a similar procedure for recomputation.

4.6 Transition

For the state transition, when we apply an action to an agent, $\{\mathbb{H}^0, \mathbb{H}^1\}$ and the node which is under execution will change. However, actions will not influence the relationships between nodes and attributes of nodes. For example, As shown in Figure 4, the agent is in the 3_{th} node. It takes one second to finish each GPU operation. Offloading X_0 takes 1.5 seconds. If we choose an action which is doing nothing or removing X_0, the agent will be in the 4_{th} node. If the action is offloading X_0, the agent will be in the 5_{th} node since we always round up the agent to the next node. We need an extra check. The current GPU memory usage should be less or equal than the memory limit. If we choose to offload X_0, and the current memory usage is equal to the memory limit, the agent will be in the 4_{th} node instead of in the 5_{th} node. We will not $malloc$ new variables before the GPU memory has enough space. The forward overhead for offloading X_0 is 0.5 seconds since GPU operation pause for 0.5 seconds to wait for offloading. We have known the prefetching order, which is described in the Section 4.5 so that we can calculate the backward overhead in the same way.

4.7 Rewards

If we never pause GPU operations for variable swapping to make the memory usage less than the memory limit, and there is no recomputation, the reward will be zero. As shown in Figure 4, if we choose to remove X_0, the reward will be negative time for recomputing forward layer functions to get X_0 in the backward propagation. If we choose to offload X_0, the reward will be negative overhead of the forward and backward propagation which is caused by offloading and prefetching X_0.

In order to get the right state transitions and rewards, we need to know the exact time for each GPU operation. However, sometimes, we cannot fit a huge model to GPU memory. We came up with an idea to solve the problem. During training, we *free* all big intermediate results. Before we reuse the intermediate results, we *malloc* new intermediate results for backpropagation. Because we use memory pool(NVIDIA, 2019), its *free* and *malloc* are fast, so we need neglectable extra time for *free* and *malloc*. Hence, the training time will be roughly correct. It should be noted that we cannot get the right derivative of weights by such way. However, we can get the roughly correct time for each GPU operation.

4.8 Environment

When we apply an action to the agent, the agent will transition to the next state from the current state; at the same time, we get a reward. A simulator provides the next state and a reward following the criteria that we defined in the action, transition, and reward sections (Section 4.5). We use the simulator to simulate the training environment while updating DQN. We have such following two assumptions: The first one is that the recomputing time can be estimated(Jia et al., 2018). The second one is that variable swapping can run parallel with layer functions entirely.

Algorithm 1 Update DQN

```
Experience replay dataset: $\mathbb{D} = \{\}$.
for episode $= 1$ to $l$ do
    ${}^{it}\boldsymbol{g} = {}^{0}\boldsymbol{g}$, ${}^{0}\boldsymbol{g}$ is the state of the beginning of the neural network iteration.
    Action set: $\mathbb{A} = \{\}$
    while ${}^{it}\boldsymbol{g}$ are not the terminal state do
        Generate a random variable $e$ between 0.0 and 1.0
        if $e > \frac{episode}{l}$ then
            $a$ = a random action where $a$ is the index of node state.
        else
            $a = argmax_{a'} \hat{Q}({}^{it}\boldsymbol{g}, \boldsymbol{s}_{a'}^{T}; \boldsymbol{W})$
        end if
        Feed simulator ${}^{it}\boldsymbol{g}, {}^{it}\boldsymbol{s}_{a}^{T}$ and it provides ${}^{it}r, {}^{it+1}\boldsymbol{g}$.
        Add tuple $({}^{it}\boldsymbol{g}, {}^{it}\boldsymbol{s}_{a}^{T}, {}^{it}r, {}^{it+1}\boldsymbol{g})$ to $\mathbb{D}$
        Update $\boldsymbol{W}$ over Equation 4 for $\mathbb{B}$ by SGD
        ${}^{it}\boldsymbol{g} = {}^{it+1}\boldsymbol{g}$, where batch $\mathbb{B}$ is sampled from $\mathbb{D}$
        Add $a$ to $\mathbb{A}$
    end while
end for
return $\mathbb{A}$
```

4.9 Q VALUES

It is easy to convert graph states to Q values. We concatenate the graph state and an action node state to a vector and then map the vector to a value. The action node represents not only a GPU operation but also an action(Figure 4).

$$\hat{Q}(\boldsymbol{g}, \boldsymbol{s}_{a}^{T}; \boldsymbol{W}) = \boldsymbol{W}_{7} ReLU([\boldsymbol{g}, \boldsymbol{W}_{8}\boldsymbol{s}_{a}^{T}]) \tag{3}$$

where $\boldsymbol{W}_7 \in R^{1\times 3p}$ and $\boldsymbol{W}_8 \in R^{p\times p}$. $\boldsymbol{s}_{a}^{T}$ is an action node. As shown in Figure 4, we can begin to copy X_0 into CPU memory while reading X_0, but we need to remove X_0 from GPU memory after reading X_0. It is noteworthy that we cannot offload X_0 after removing X_0, and vice versa. We usually use the first node to represent doing nothing action. We use a heuristic method to decide which node can also represent an action, and we guarantee that no node represents more than one action.

4.10 FITTED Q-ITERATION

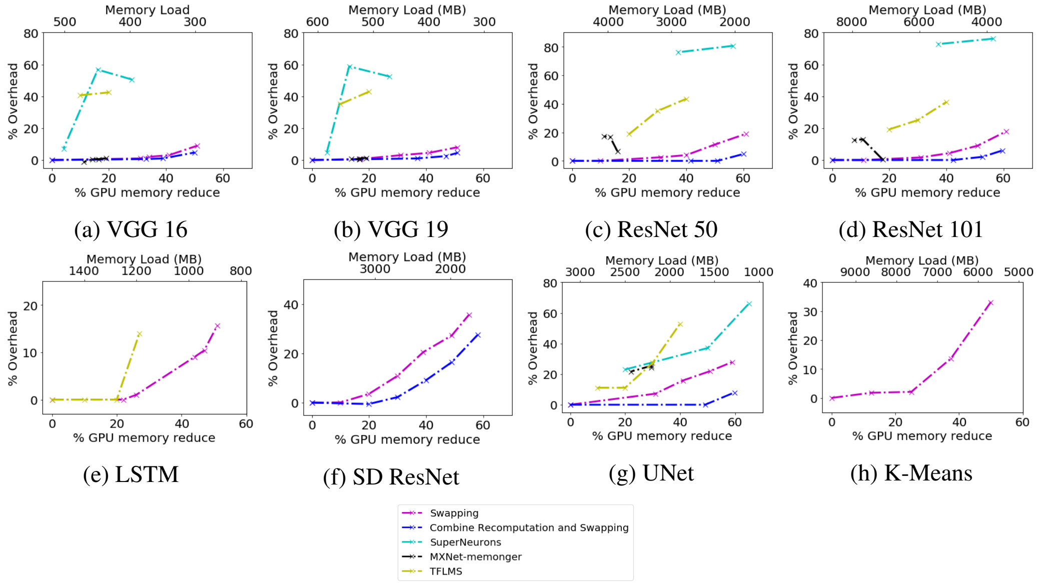
We train an end-to-end DQN by the following loss function.

$$loss = (y - \hat{Q}({}^{it}\boldsymbol{g}, {}^{it}\boldsymbol{s}_{a}^{T}; \boldsymbol{W}))^2 \tag{4}$$

$$y = \gamma max_{\boldsymbol{s}_{a'}^{T}} \hat{Q}({}^{it+1}\boldsymbol{g}, \boldsymbol{s}_{a'}^{T}; \boldsymbol{W}) + r({}^{it}\boldsymbol{g}, {}^{it}\boldsymbol{s}_{a}^{T}) \tag{5}$$

where γ is a decay factor. y is a constant value, which means that the gradient will not flow through y. $r({}^{it}\boldsymbol{g}, {}^{it}\boldsymbol{s}_{a}^{T})$ is the reward for agent state ${}^{it}\boldsymbol{g}$ and action ${}^{it}\boldsymbol{s}_{a}^{T}$. Terminal state $\hat{Q}_t$ is zero. If we no longer need to remove or offload any variables until the end of the current iteration, and $max_{i\in\{0,\ldots,n\}} m_i$ is less than the memory limit, the state will be the terminal state. As shown in algorithm 1, we do not update the Equation 4 by the single currently experienced sample. Instead, fitted Q-iteration updates the weights with mini-batches sampled from experience replay dataset. We use the ϵ greedy method to choose an action a from $\{\mathbb{H}^0, \mathbb{H}^1\}$.

Finally, we train the DNN following the plan that is generated by DQN. We execute each GPU operation in a time sequence. If the current node is an action node, and the action is in action set $\mathbb{A}$, we will execute the action following Section 4.6. As for prefetching and the second phase of recomputation, we following the method introduced in Section 4.5.



(a) VGG 16

(b) VGG 19

(c) ResNet 50

(d) ResNet 101

(e) LSTM

(f) SD ResNet

(g) UNet

(h) K-Means

Figure 5: Different overheads for different memory reductions. The y-axis represents performance overhead. The lower x-axis shows memory reduction, and the upper x-axis shows corresponding memory usage.

5 Experiment

In this part, we evaluate the performance of variable swapping and variable swapping combined with recomputation. We test our method on various architectures such as ResNet, VGG, UNet, K-means(Src-D, 2019), and LSTM, whose structures do not change during training. We also extend our method on a dynamic computation graph, e.g., deep networks with stochastic depth(Huang et al., 2016), whose structures can change during training. Our images dataset is CIFAR-10 and Karpathy's char-RNN dataset(Karpathy, 2019). Our k-means dataset is generated randomly by NVIDIA code. Additionally, we train ResNet and VGG for two different depths for better analyzation.

Our experiments are conducted on a workstation, which equipped CPU Intel Xeon E5 and NVIDIA GeForce GTX 1080 Ti with 11 GB RAM. The CPU memory is 64GB. Our motherboard has PCI Express x16 for data communication between GPU and CPU. Our system is Ubuntu 16.04, with CUDA 9.0 and cuDNN 7.0. We use the fastest cuDNN algorithm, which requires extra workspaces to store intermediate results. Our method is tested on deep learning framework Singa (Wang et al., 2015; Ooi et al., 2015).

5.1 Compare with Other Baselines

We compare our method with other baselines. They are MXNet-memonger(Chen et al., 2016), SuperNeurons(Wang et al., 2018), and TFLMS(Le et al., 2018). MXNet-memonger trades computation to exchange memory, but the performance depends on recomputing layers that we choose. SuperNeuraons have proposed recomputation and variable swapping. TFLMS only uses variable swapping.

Figure 5 shows different computation overheads versus different memory reductions. For MXNet-memonger, we can obtain different data points in our graphs as changing recomputation layers. As for SuperNeurons, we choose to use recomputation mode, swapping mode, and swapping combined with recomputation mode to get three different data points. Their program does not have recomputation mode for ResNet, so we only report two data points in ResNet for their baseline. As for TFLMS, we choose the different number of swapping variables to control memory usage. Our method takes less extra computation time and saves more GPU memory, which shows that our results are better than MXNet-memonger, SuperNeurons, and TFLMS.

SuperNeuron uses the least recently used (LRU) algorithm for variable swapping. They view GPU memory as a cache and use a classical cache replacement algorithm. However, they do not make use of the iterative nature of the DNN. TFLMS only uses variable swapping, and they need to set the number of swap variables manually. SuperNeuron and MXNet-memonger choose specific layers for recomputation by expert knowledge. Our method makes use of more information from computation graphs and provides plans automatically for users.

SuperNeuron also combines variable swapping with recomputation. However, they do not treat variable swapping and recomputation separately. When we run their program, we find that their program GPU utilization during network training is much lower than ours. They waste some GPU resource for saving memory, which can be avoided. The GPU utilization of our method is higher than theirs.

Our work can be used in more general architectures. Only TFLMS and our method can work on LSTM function *CuDNNLSTM*. We cannot run the other two methods on such architecture. Additionally, among these four works, only our method supports ResNet with stochastic depth and K-Means.

Compared with other baselines, our algorithm has the following advantages. First of all, we can set a wide range of memory limit easily. Secondly, our method can work well on an extensive range of iterative nature machine learning algorithms. Last, our method provides plans automatically for users, and users do not need expert knowledge.

5.2 Compare for Different Architectures

Let us analyze our method for different architectures. For ResNet and VGG, they have similar architectures and get similar results. Regarding UNet, its structure is different from that of ResNet and VGG. For example, the first feature map is required to be used at the end phase of the forward propagation and the second feature map need to be used at the second last phase of the forward pass and so on. If we offload the first feature map, we need to prefetch it before the last phase of the forward pass, which means we need to swap it in again in memory usage growing phase. If we do not offload such variables, GPU data transfer engines will be idle for some time or have fewer candidate variables to be offloaded. Thus results on UNet is worse than ResNet and VGG. Concerning LSTM, it does not have convolutional layers. Convolutional layers execute slower than other layers. If we need to use longer time to do kernel operation in GPU, we will have a longer time for data transfers since kernel operation, and data transfers are executed in different GPU engines. In consequence, the overhead of LSTM is longer than that of ResNet and VGG. As to SD ResNet, it has dynamic structures. The architecture of the network can change during training. Our method is not designed for such structures, so the result is worse than others.

6 Conclusions

In this paper, we propose a DQN to devise plans for variable swapping and recomputation to reduce memory usage. Our work can work well with different memory limits. Our method provides plans automatically for users. They only need to set a memory limit and do not require background knowledge on DNN or machine learning algorithm. Our method can work well for different network structures such as ResNet, VGG, K-means, SD ResNet, and LSTM. Besides, the variable swapping and recomputation do not decrease the accuracy of networks.

Acknowledgments

We appreciate that Xuehu Yu, Yuanzhi Yue shared exciting ideas with us for this work. We thank Junzhe Zhang for sharing his GPU memory management C++ code with us. We were so grateful that Shicong Lin, James, Dongwen Lin, Yidan Sun, Pingcheng Ruan, Yisen Ng, Xutao Sun, and Dongjun Zhai help us to remove ambiguity for the paper.

REFERENCES

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in Neural Information Processing Systems*, pp. 2214–2224, 2017.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016b.

Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 558–567, 2019.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pp. 646–661. Springer, 2016.

Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018.

Andrej Karpathy. http://cs.stanford.edu/people/karpathy/char-rnn/, 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037*, 2018.

Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger Grosse. Reversible recurrent neural networks. In *Neural Information Processing Systems (NeurIPS)*, 2018.

Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

NVIDIA. https://github.com/nvidia/cnmem, 2019.

Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony KH Tung, Yuan Wang, et al. Singa: A distributed deep learning platform. In *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 685–688. ACM, 2015.

Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 18. IEEE Press, 2016.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Src-D. src-d/kmcuda, Feb 2019. URL `https://github.com/src-d/kmcuda`.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Gregor Urban, Krzysztof J Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Rich Caruana, Abdelrahman Mohamed, Matthai Philipose, and Matt Richardson. Do deep convolutional nets really need to be deep and convolutional? *arXiv preprint arXiv:1603.05691*, 2016.

Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *ACM SIGPLAN Notices*, volume 53, pp. 41–53. ACM, 2018.

Wei Wang, Gang Chen, Anh Tien Tuan Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. Singa: Putting deep learning in the hands of multimedia users. In *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 25–34. ACM, 2015.