A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation

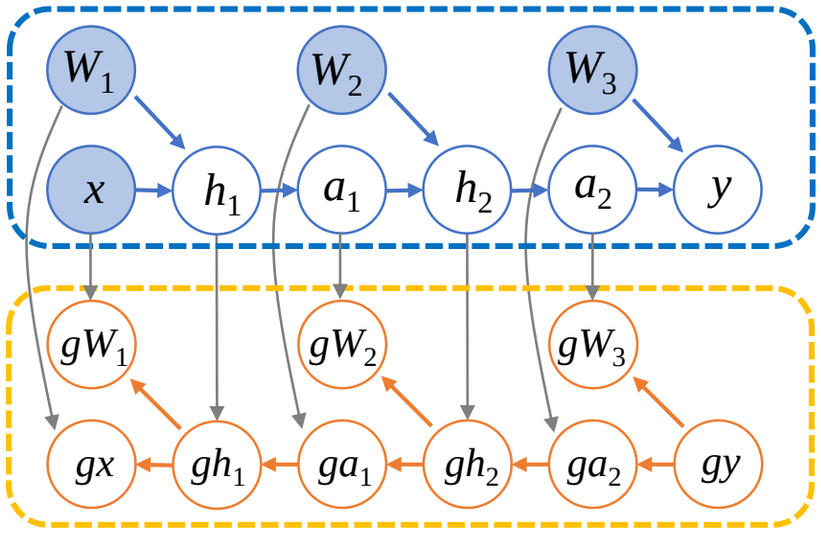
Mitsuru Kusumoto*, Takuya Inoue**,
Gentaro Watanabe*, Takuya Akiba*, Masanori Koyama*

*:Preferred Networks
**: The University of Tokyo
The first two authors are equal contribution.

Training DNNs is memory consuming

- Modern deep neural networks tend to be large.
- Input to the networks can be large. (e.g., high resolution images)

When training DNNs, we usually need to store all the intermediate variables in both forward and backward computation.



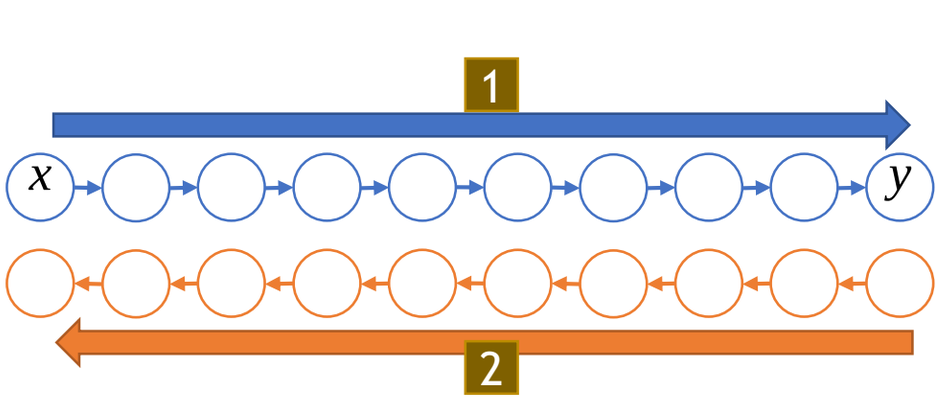
The lack of ample free memory enforces researchers to use small batch-size. This may cause low accuracy because the quality of batch normalization may degrade.

Memory reduction in backpropagation is an important issue.

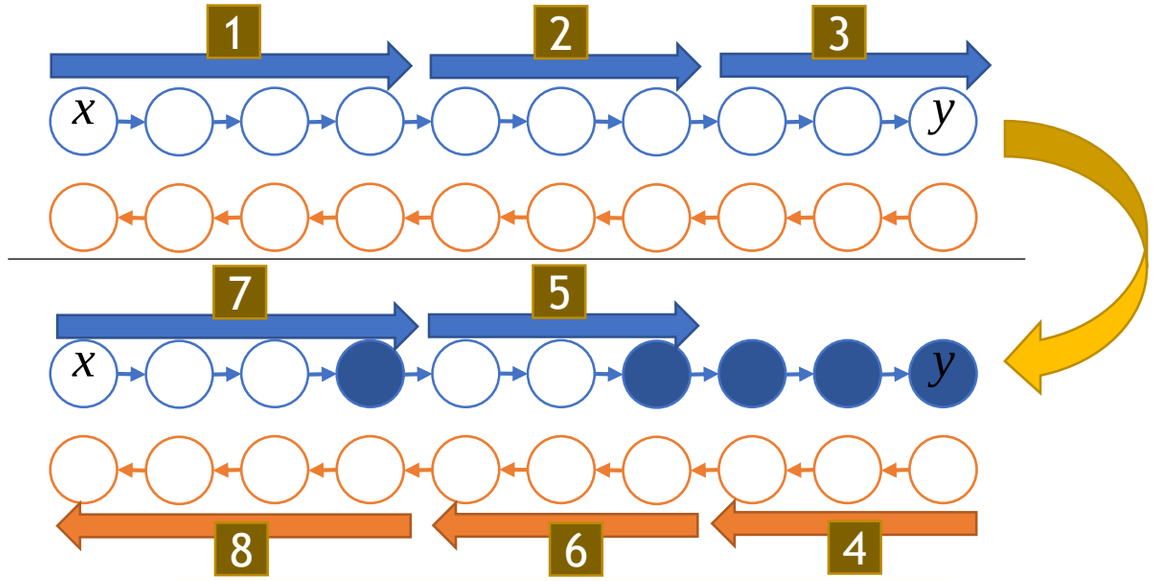
Recomputation can save memory

A recomputation method (a.k.a. check-pointing) is a smart memory manipulation that can reduce the memory consumption.

- Unlike other memory reduction methods, recomputation methods do not alter the outcome of the computation or compromise the accuracy.
- Trade computation time to memory consumption - deliberately discard intermediate results and recompute them on a need basis.



Vanilla execution



Execution with recomputation

The efficacy of recomputation method depends on its schedule – *what to forget and what to cache in what order*.

Our contribution

Recomputation methods have been introduced in deep learning community [1,2]. However, their applications were limited to a specific family of networks.

- In our study, we propose a novel and efficient recomputation that can be applied to theoretically all types of neural nets.
- To this end, we introduce general recomputation problem using the language of graph theory and provide an efficient dynamic programming (DP) solution.
- In experiments, we show that our method can reduce the peak memory consumption by 36%~81% on various benchmark networks.

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

[2] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In Advances in Neural Information Processing Systems (NIPS), pages 4125–4133, 2016.

Problem formulation (1/2)

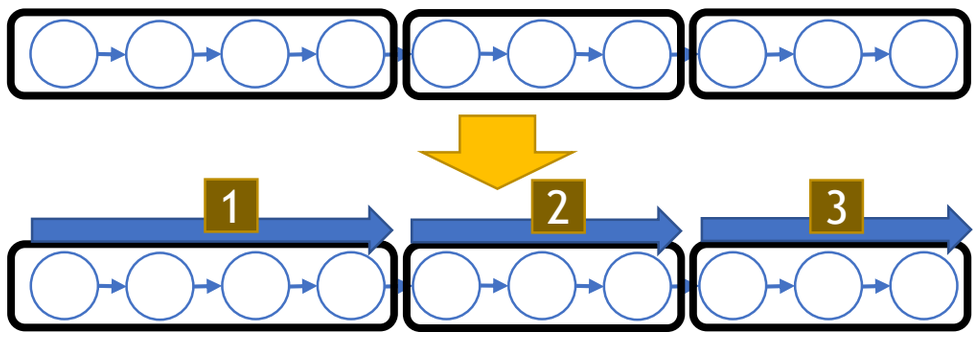
Problem setting

- We are given a network structure and available memory budget.
- Each node represents a tensor.
- Each node has its memory consumption and computational time.
- We want to find a recomputation strategy with small computational overhead that satisfies memory constraint.

Since backward part will be automatically determined from forward part, we will focus on only forward part.

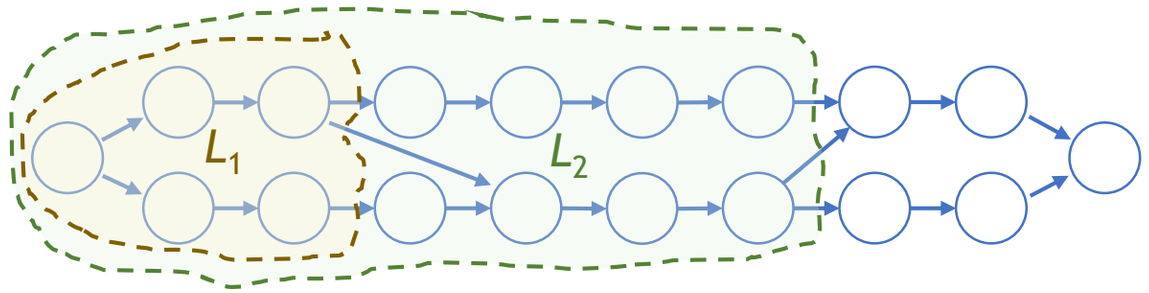
Problem formulation (2/2)

In path-like network



Partition of the path determines its “canonical” recomputation strategy.

In network with general structure



Sequence of *lower sets* determines its canonical strategy.

Here, we define a node set L is a lower set if there is no edge from $(V\text{-}L)$ to L.

Given a lower set sequence $(L_1, \ldots, L_k)$, the computational overhead and peak memory consumption of its canonical strategy can be analyzed. We denote them by $T(L_1, \ldots, L_k)$ and $M(L_1, \ldots, L_k)$.

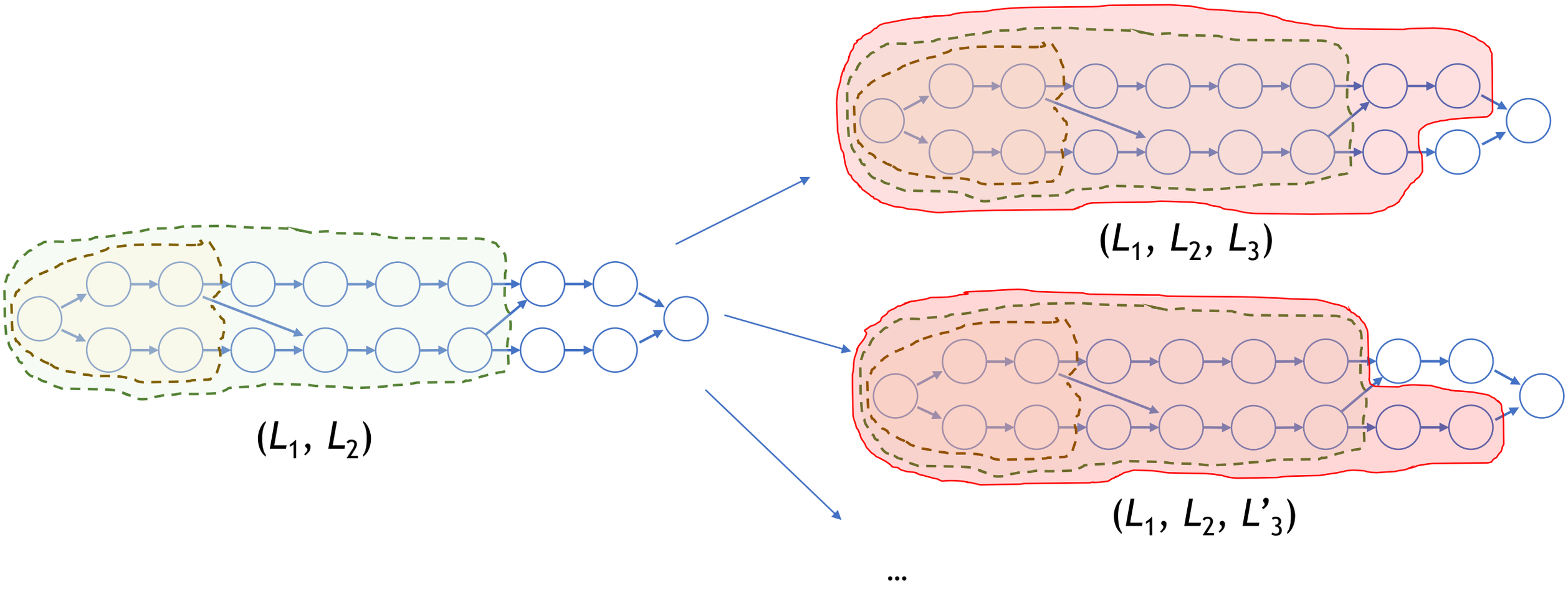
General Recomputation Problem

Let us find a sequence $(L_1, \ldots, L_k)$ that minimizes overhead $T(L_1, \ldots, L_k)$ under memory constraint $M(L_1, \ldots, L_k) \leq$ (budget).

Our algorithm (1/3)

Naive approach

Because number of possible sequences is finite (though astronomically large), we can design brute force algorithm to enumerate all sequences.

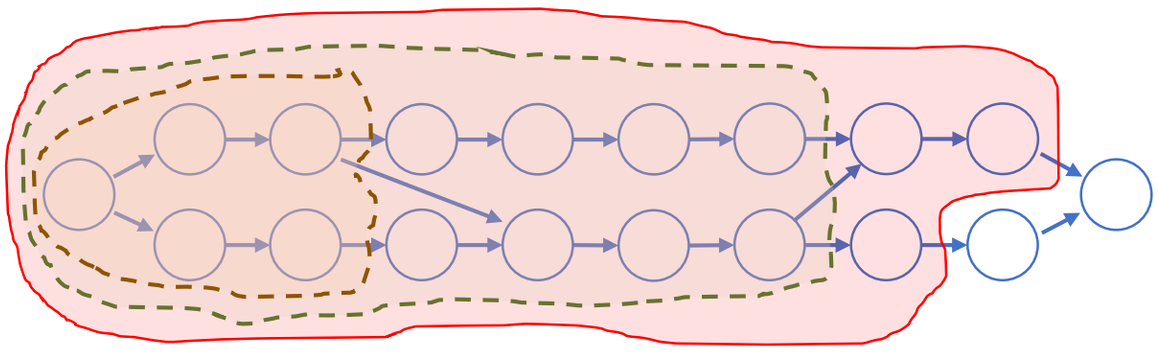


Our algorithm (2/3)

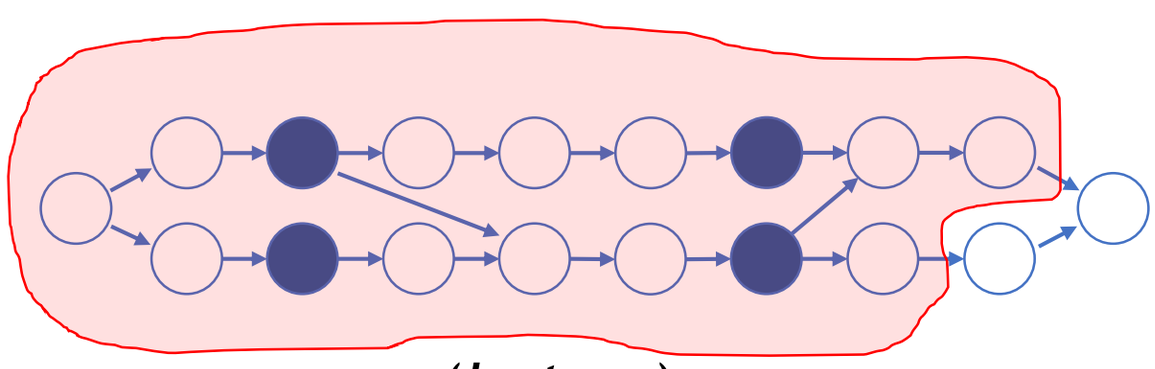
Efficient algorithm During the construction of sequences in brute force algorithm, we only need to retain a limited information rather than the sequence itself.

Instead of (L_1, …, L_i), we retain a triplet (L, t, m). This yields efficient DP algorithm.

Runtime will be O(#(lower sets)^2 * (total overheads))



(L_1, L_2, L_3)



(L, t, m)

L := L_3 (last element)
t := Computational overhead so far
(discarded node will be reocmputed)
m := Memory consumption so far

Our algorithm (3/3)

Approximate algorithm In some complex network, the number of lower sets can be very large.

We experimentally found that using a good small subset of lower sets as keys in DP table yields good performance. (optimality will not be guaranteed)

Runtime will be *O*(#(vertices)^2 * (total overheads)).
This runs very fast.

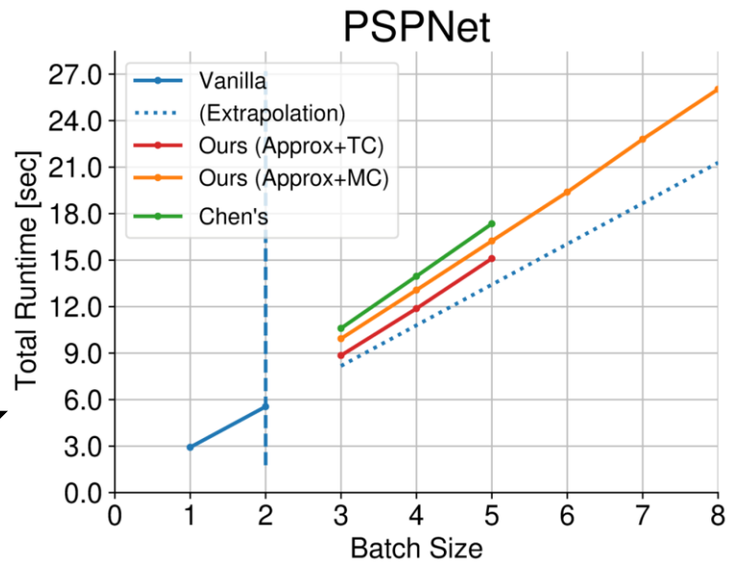
Memory centric strategy We can apply liveness analysis to our canonical strategy. We experimentally found that liveness analysis tends to work well when the node-set is partitioned *coarsely*.

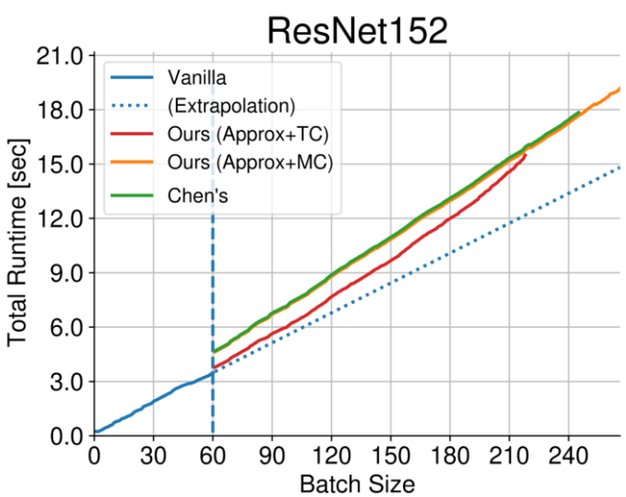
We discovered that we can realize this coarse partition intentionally by using a strategy with long computational overhead.

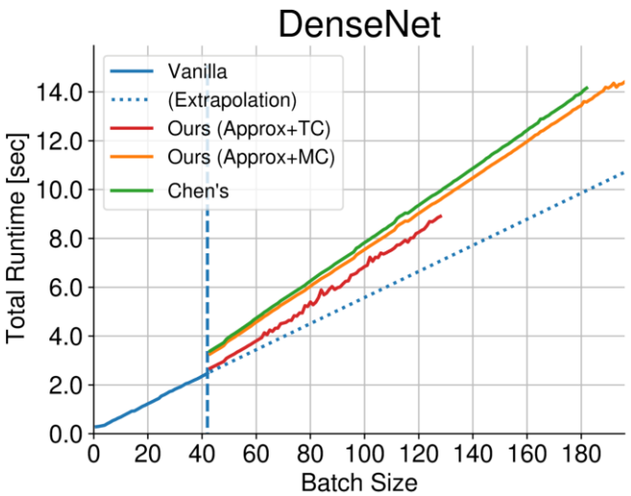
Results

Computation time vs. memory consumption

Lower is better







In presence of ample memory, our method seeks a strategy with small computational overhead.

Minimum memory consumption

Memory consumption is shown.

(-xx%) is a reduction from vanilla execution.

Network	Ours (approx.)	Ours (exact)	Chen's [1]	Vanilla
PSPNet	**2.7 GB (-71%)**	2.8 GB (-70%)	4.0 GB (-58%)	9.4 GB
U-Net	5.0 GB (-45%)	**4.7 GB (-48%)**	7.4 GB (-18%)	9.1 GB
ResNet50	**3.4 GB (-62%)**	**3.4 GB (-62%)**	3.7 GB (-59%)	8.9 GB
ResNet152	**2.3 GB (-75%)**	**2.3 GB (-75%)**	2.4 GB (-74%)	9.2 GB
VGG19	**4.5 GB (-36%)**	**4.5 GB (-36%)**	4.7 GB (-34%)	7.0 GB
DenseNet	**1.6 GB (-81%)**	1.7 GB (-80%)	1.8 GB (-79%)	8.5 GB
GoogLeNet	**5.2 GB (-39%)**	**5.2 GB (-39%)**	6.5 GB (-24%)	8.5 GB

NOTE: For some networks, the approximate DP yielded slightly better results than the exact DP because the effect of the liveness analysis is not taken into the account in the definition of optimality in the general recomputation problem.