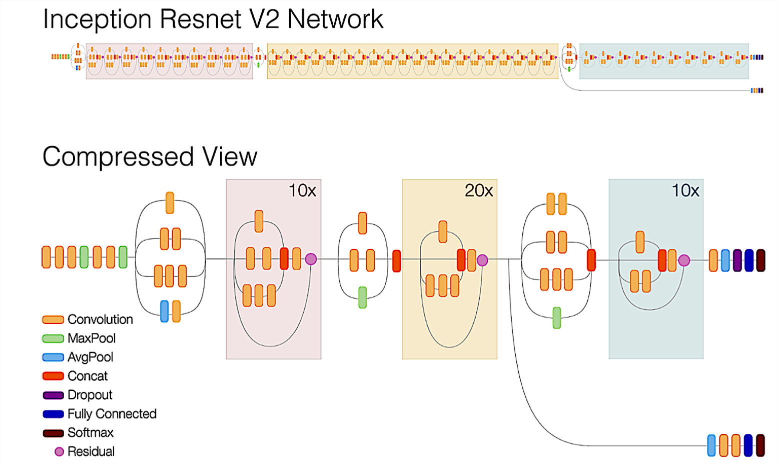
Memory Optimization with Rematerialization when Training DNNs

Inria Skoltech Workshop

Alena Shilova,
with Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Alexis Joly
July 7 2021

DL training phase: computational DAG

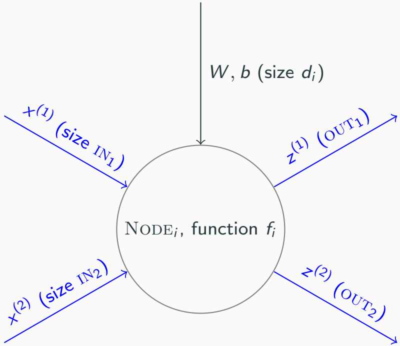


DI training phase: computational DAG

for instance,

$f_i = RELU(Wx + b)$

Node of the DAG (*layer*)



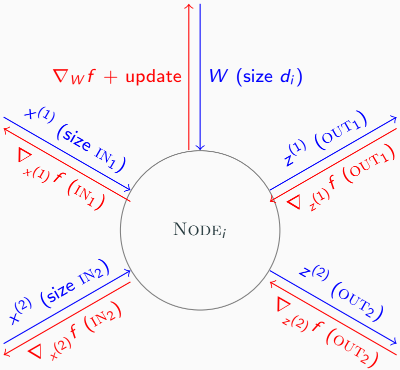
Memory Issues

Source of memory problems

Heavy models
This problem occurs when the weights of the model take a lot of memory space. That causes the problem in inference as well.

Heavy training
The problem occurs when the activations are too expensive to store, e.g. batch-size or input sample are too big. The problem does not affect inference stage.

Distributed DL: forward propagation and backward propagation



- $\frac{\partial f}{\partial x_i^{(1)}} = \frac{\partial f}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial x_i^{(1)}} + \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial x_i^{(1)}}$
- $\frac{\partial f}{\partial x_i^{(2)}} = \frac{\partial f}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial x_i^{(2)}} + \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial x_i^{(2)}}$
- $\frac{\partial f}{\partial W_i} = \frac{\partial f}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W_i} + \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W_i}$

- Example
  $f(x) = \sigma(wx + b) = \sigma(z)$
- $\frac{\partial f}{\partial x} = \sigma'(z)w$
- $\frac{\partial f}{\partial w} = \sigma'(z)x$

DL: forward propagation and backward propagation

- forward propagation
  - propagate the input through the network to compute loss
- backward propagation
  - compute gradients with respect to loss
  - update the weights with gradients

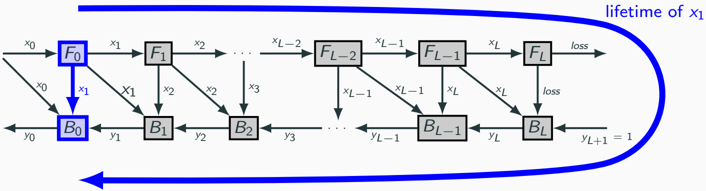


Figure 1: Linearized view of neural network

Memory is consumed by activations throughout the entire training!

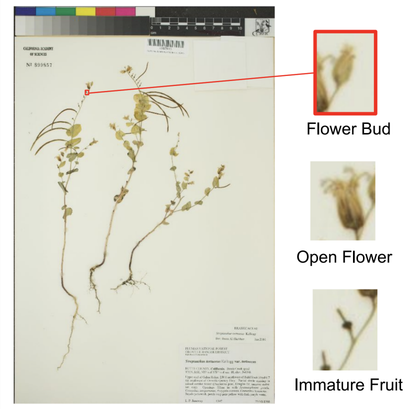
Memory consumption

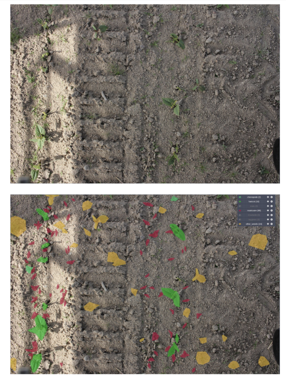
	ResNet$_x$				
image width/height	$x = 18$	$x = 34$	$x = 50$	$x = 101$	$x = 152$
224	0.60	0.98	2.22	3.41	4.78
350	1.22	1.93	4.90	7.45	10.47
500	2.31	3.60	9.63	14.69	20.76
650	3.79	5.86	15.99	24.13	34.06

Table 1: Memory requirement for each model to keep all weights and activations for the batch_size = 8, the amount is given in GB. The shaded values correspond to the cases where the model cannot fit into a 8GB memory.

Value Proposition
An innovative citizen science platform making use of machine learning to help people identify plants through their mobile phone

Two examples of memory-consuming tasks (in the context of Pl@ntNet)





(i) Detection & counting of small reproductive structures in digitized herbarium

(ii) Early **detection & classification** of weeds in precision agriculture

Two examples of memory-consuming tasks (in the context of Pl@ntNet)

Performance with a state-of-the-art model and largest image size fitting in GPU memory is strongly affected by object's size

Model: Mask R-CNN
*Image size:*1200x2048
GPU memory: 16Gb
Mini-batch size: 1

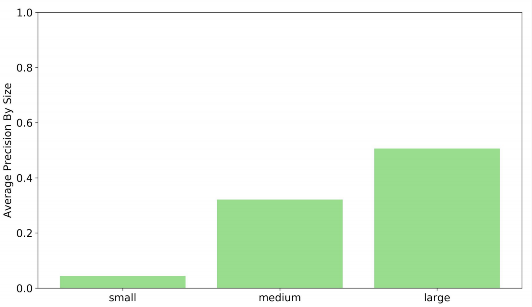


Figure 3: Detection & classification of weeds (performance by object's size)

Memory saving techniques

Special neural networks:

- **Memory efficient architectures:**
  - Reversible neural networks (RevNet);
  - Quantized neural networks;
  - MobileNet;
  - ShuffleNet;
- **Layer optimization:**
  - memory-efficient batch-normalization layer

Usage of several machines:

- Data parallelism;
- Model parallelism;
- Spatial parallelism;

Single Node Memory Saving Techniques

Efficient training on one node/GPU

- **Rematerialization**
  - work more and stock less (discard some data and recompute it after);
  - known as checkpointing in Automatic Differentiation;
- **Offloading:**
  - Use lower memory hierarchy:
    - train on GPU;
    - send activations (or weights) to CPU

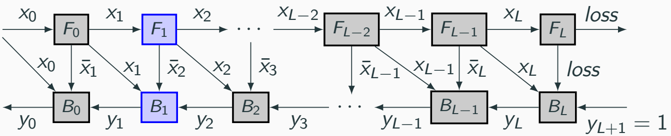
Pros & Cons

− Overhead cost: extra computations or occupation of the PCI Bus

+ Suitable for training any NN architecture with limited resources

Rematerialization

Rematerialization



Main idea
To work more and stock less: instead of keeping all activations we store some of them and recompute others once we need them.

Analogous to Automatic Differentiation
This technique is very common in AD. The optimal schedule for checkpointing can be found with the help of Dynamic Programming.

Single Adjoint Chain Computation problem

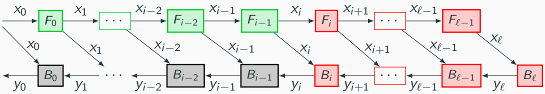


Figure 4: The data dependencies in the AC chain.

Input: cost of one forward step u_f, cost of one backward step u_b, chain length ℓ and total memory size m.

$$\mathrm{Opt}_0(\ell, 3) = \frac{\ell(\ell+1)}{2} u_f + (\ell+1) u_b$$

$$\mathrm{Opt}_0(0, m) = u_b$$

$$\mathrm{Opt}_0(\ell, m) = \min_{1 \le i \le \ell} \{ i u_f + \mathrm{Opt}_0(\ell - i, m-1) + \mathrm{Opt}_0(i-1, m) \}$$

Single Adjoint Chain Computation problem

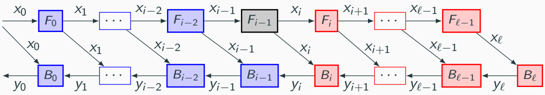


Figure 4: The data dependencies in the AC chain.

Input: cost of one forward step u_f, cost of one backward step u_b, chain length ℓ and total memory size m.

$$\mathrm{Opt}_0(\ell, 3) = \frac{\ell(\ell+1)}{2} u_f + (\ell+1) u_b$$

$$\mathrm{Opt}_0(0, m) = u_b$$

$$\mathrm{Opt}_0(\ell, m) = \min_{1 \le i \le \ell} \{ i u_f + \mathrm{Opt}_0(\ell - i, m - 1) + \mathrm{Opt}_0(i - 1, m) \}$$

Extension to Heterogeneous chain

$\mathrm{Opt}(i, \ell, m)$: execution time for a heterogeneous chain from i to ℓ with memory m.

$$\mathrm{Opt}(i, i, m) = \begin{cases} u_{B_i} & \text{for } m \geq x_i + y_{i+1} + y_i \\ \infty, & \text{otherwise} \end{cases}$$

$$\mathrm{Opt}(i, \ell, m) = \begin{cases} \min\limits_{j=i+1,\ldots,\ell} \left\{ \sum\limits_{k=i}^{j-1} u_{F_k} + \mathrm{Opt}(j, \ell, m - x_j) + \mathrm{Opt}(i, j-1, m) \right\} \\ \infty \qquad \text{if } m < \max\{x_{i+1}, x_{i+1} + x_{i+2}, \ldots, x_{\ell-1} + x_\ell\} \end{cases}$$

Difference with $\mathbf{Opt}_0(\ell, m)$:

- new parameter: position in the subchain (instead of length only)
- memory costs are not anymore unitary, but all values are integers
- it is not optimal anymore in general case (not memory persistent)

DNN frameworks rematerialization

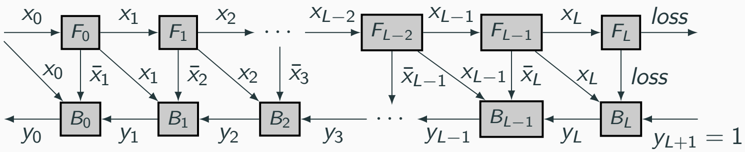


Figure 5: The data dependencies in the Adjoint Computation graph in PyTorch.

- extra dependencies ($\downarrow$-edges)
- different ways of checkpointing: (recording or saving only input)
- new dynamic programming is required
- and it should be suitable for most part of the state-of-the-art models

Optimal checkpointing for general sequential models

$$\mathrm{Opt}_{\mathrm{BP}}\,(i,\ell,m) = \min \begin{cases} \mathrm{Opt}_1(i,\ell,m) \\ \mathrm{Opt}_2(i,\ell,m) \end{cases}$$

$$\mathrm{Opt}_1(i,\ell,m) = \min_{j=i+1,\ldots,\ell} \sum_{k=i}^{j-1} u_{F_k} + \mathrm{Opt}_{\mathrm{BP}}\,(j,\ell,m-x_j) + \mathrm{Opt}_{\mathrm{BP}}\,(i,j-1,m)$$

$$\mathrm{Opt}_2(i,\ell,m) = u_{F_i} + \mathrm{Opt}_{\mathrm{BP}}\,(i+1,\ell,m-\bar{x}_{i+1}) + u_{B_i}$$

Formulas are valid as long as **memory constraints are not violated!**

Optimal checkpointing for general sequential models

$$\mathrm{Opt}_{\mathrm{BP}}\,(i,\ell,m) = \min \begin{cases} \mathrm{Opt}_1(i,\ell,m) \\ \mathrm{Opt}_2(i,\ell,m) \end{cases}$$

$$\mathrm{Opt}_1(i,\ell,m) = \min_{j=i+1,\dots,\ell} \sum_{k=i}^{j-1} u_{F_k} + \mathrm{Opt}_{\mathrm{BP}}\,(j,\ell,m-x_j) + \mathrm{Opt}_{\mathrm{BP}}\,(i,j-1,m)$$

$$\mathrm{Opt}_2(i,\ell,m) = u_{F_i} + \mathrm{Opt}_{\mathrm{BP}}\,(i+1,\ell,m-\bar{x}_{i+1}) + u_{B_i}$$

Formulas are valid as long as **memory constraints are not violated!**

We implemented this dynprog in **rotor** (see Lionel's talk)

Optimal checkpointing for general sequential models

$$\mathrm{Opt}_{\mathrm{BP}}\left(i,\ell,m\right) = \min \begin{cases} \mathrm{Opt}_1(i,\ell,m) \\ \mathrm{Opt}_2(i,\ell,m) \end{cases}$$

$$\mathrm{Opt}_1(i,\ell,m) = \min_{j=i+1,\ldots,\ell} \sum_{k=i}^{j-1} u_{F_k} + \mathrm{Opt}_{\mathrm{BP}}\left(j,\ell,m-x_j\right) + \mathrm{Opt}_{\mathrm{BP}}\left(i,j-1,m\right)$$

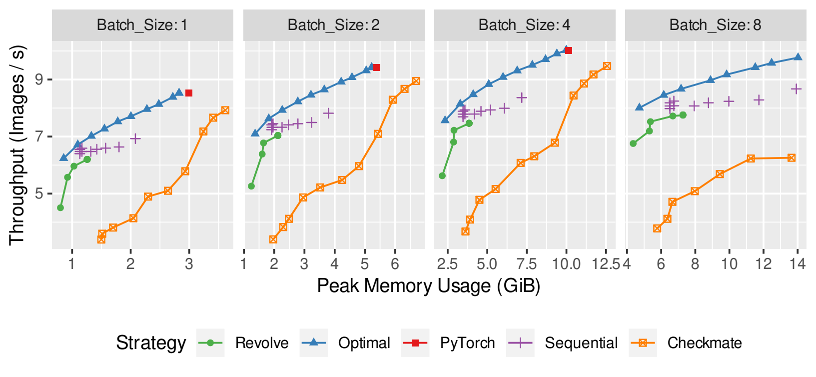
$$\mathrm{Opt}_2(i,\ell,m) = u_{F_i} + \mathrm{Opt}_{\mathrm{BP}}\left(i+1,\ell,m-\bar{x}_{i+1}\right) + u_{B_i}$$

Formulas are valid as long as **memory constraints are not violated!**

We implemented this dynprog in **rotor** (see Lionel's talk)

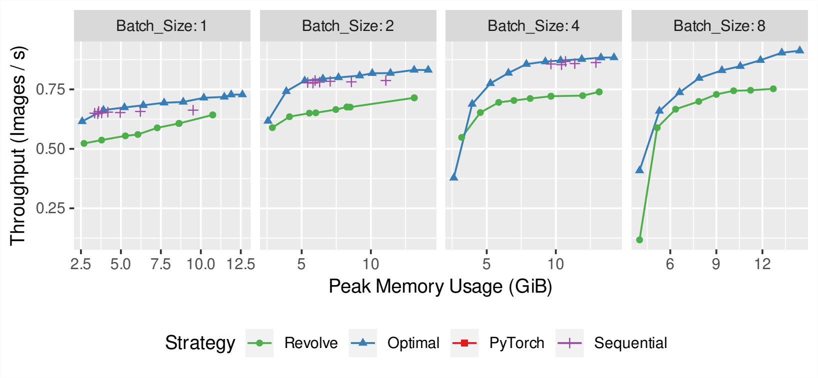
This paper is under major revision in Transactions on Mathematical Software (ACM TOMS).

Comparison of our implementation with other approaches i



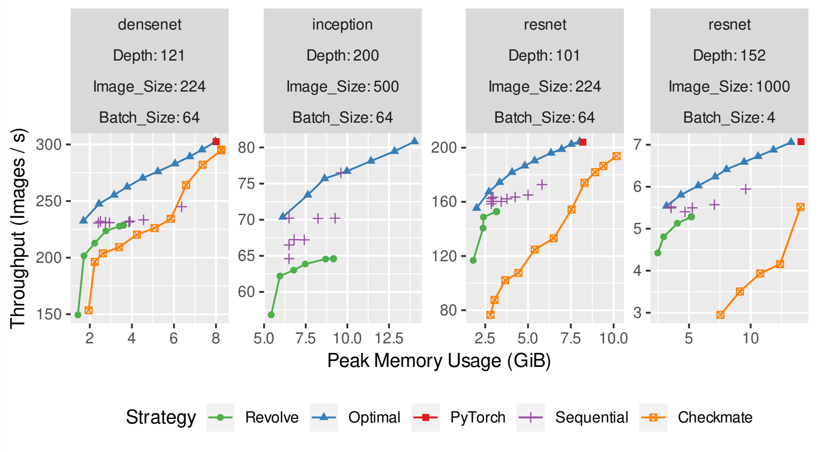
(i) Experimental results for the ResNet network with depth 101 and image size 1000.

Comparison of our implementation with other approaches ii



(ii) Experimental results for the ResNet network with depth 1001 and image size 224.

Comparison of our implementation with other approaches iii



(iii) Experimental results for several situations.

Offloading

Our work

Our goal
To find optimal approaches in identifying which activations to offload

Problem
We proved that it is NP-complete problem in the strong sense, when activations are offloaded entirely with discards only when the entire activation is on CPU

Possible relaxations

- Partial discards on GPU are possible $\rightarrow$ solved by Dynamic Programming
- Partial discards on GPU are possible + partial offloading $\rightarrow$ solved by Greedy algorithm

This work was published in EuroPar2020

Simulation results

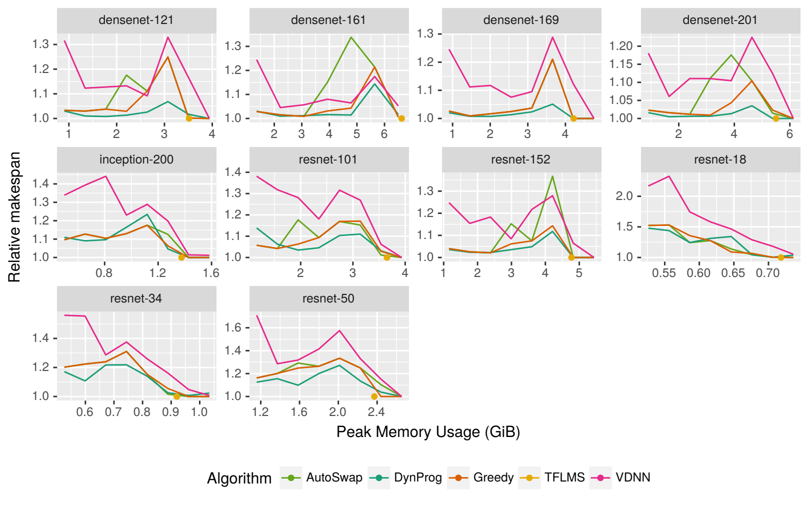


Figure 7: Experimental results for image size 224 and batch size 32

Comparison to Offloading

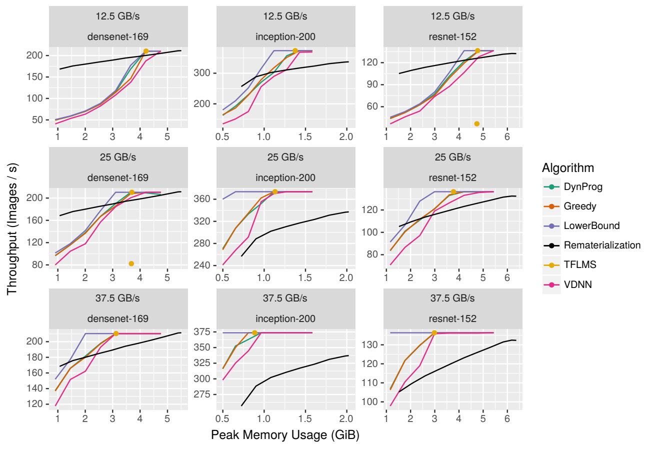


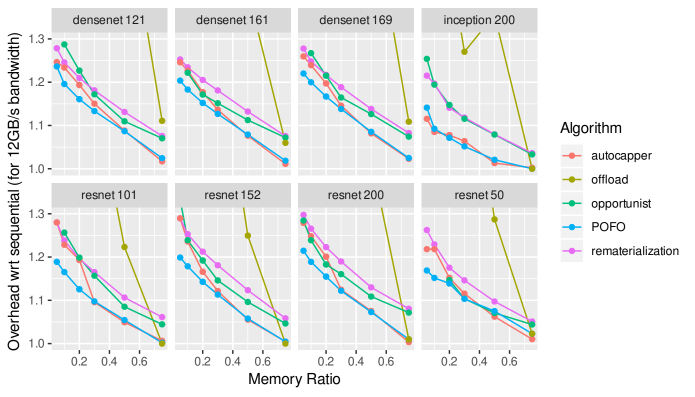
Figure 8: Rematerialization vs Offloading.

Combination of Offloading and Rematerialization

Our work

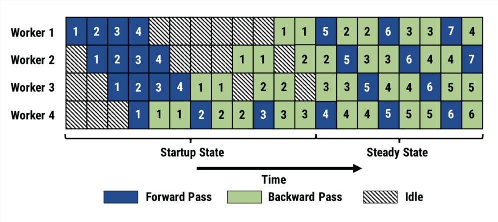
We did

- Merged dynprog for Rematerialization and Offloading in *POFO*;
- Proved its optimality;
- Proposed two heuristics *autocapper* and *opportunist*
- This work is submitted to NeurIPS 2021



Model Parallelism and Pipelining

PipeDream



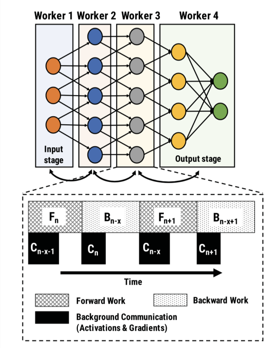


Figure 9: Example PipeDream with 4 workers.
Left picture shows the pipelining. Right picture how the model is parallelized and executed with 1F1B schedule.

Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." 2019.

Our Contribution

- Considered the limitations of the previous state-of-the-art
- Proved complexity results for load balancing and scheduling problems
- Designed the ILP to find non-contiguous allocations
- Proposed k-periodic schedules
- Proposed MADpipe (a dynamic programming that finds some non-contiguous allocations)

Part of these contributions have been published in EuroPar 2021

Conclusion

Conclusion

- It is important to reduce memory consumption
- Rematerialization is a promising solution
- We implemented rematerialization for heterogeneous chains in PyTorch (see Lionel's talk)
- In practice, most suitable for long quasi-homogeneous chains
- Additionally,
  - Offloading is another possible alternative
  - Combination of Rematerialization and Offloading improves both methods
  - Model Parallelism is suitable for distributed setting
  - **Future work:** we plan to consider offloading weights too