université de BORDEAUX

Inria

Memory Saving Strategies for Deep Neural Network Training

04/02/2021

Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrman, Alexis Joly
Alena Shilova

Introduction

- PhD thesis coadvised between 2 teams: HiePACS and Zenith
- Artificial Intelligence is a very popular direction
- PI@ntNet relies on neural networks
- Their training is time and memory-costly process
- Inference should be cheap
- Scheduling techniques should be applied to optimize both processes
- In my work we have concentrated on solving memory issues

Key words

Neural Networks

- Layers (Dense and Convolutional);
- Training and Inference;
- Forward and Backward propagations;
- Batches, Activations and Gradients;
- Weights (layer parameters)

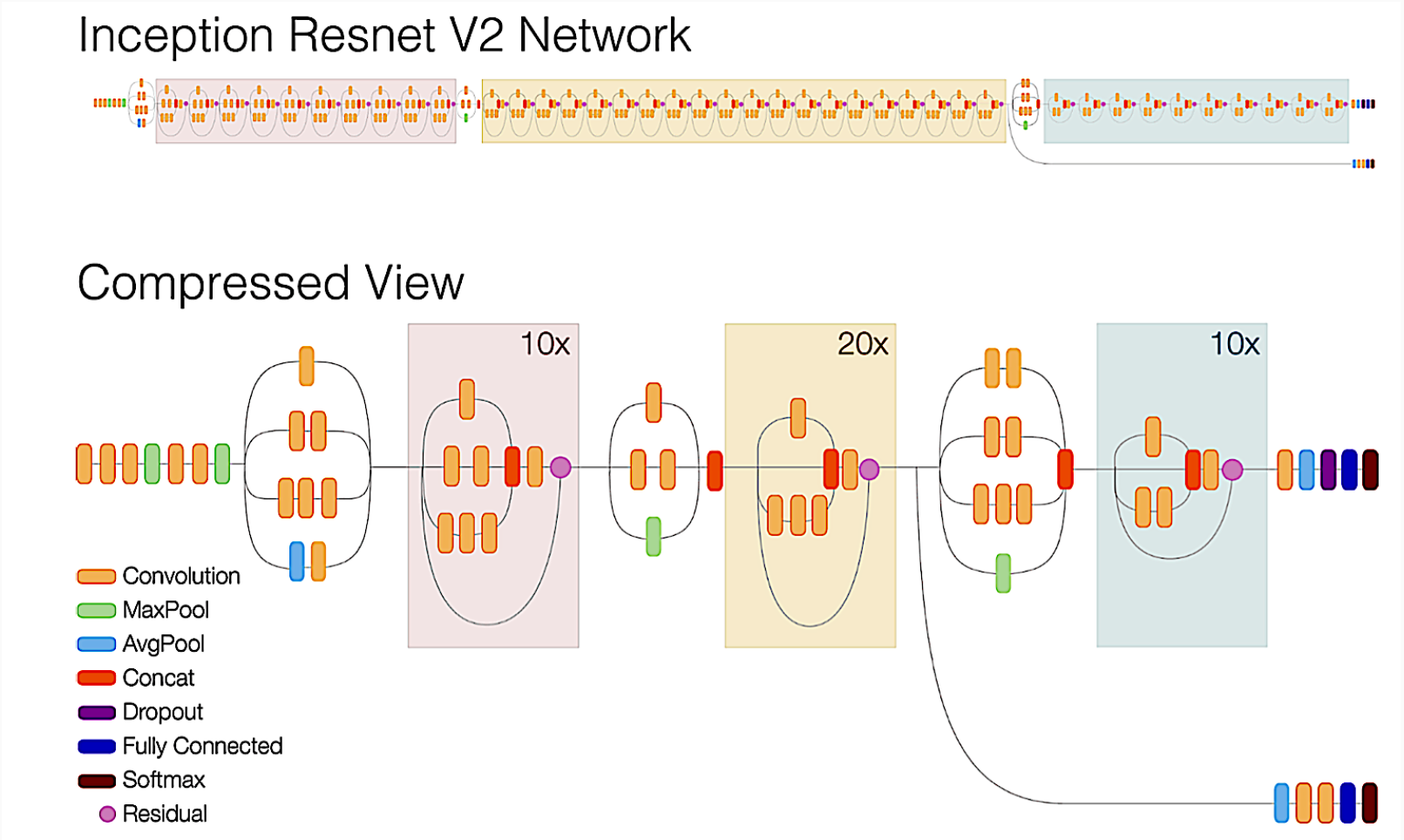
Memory issues

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.

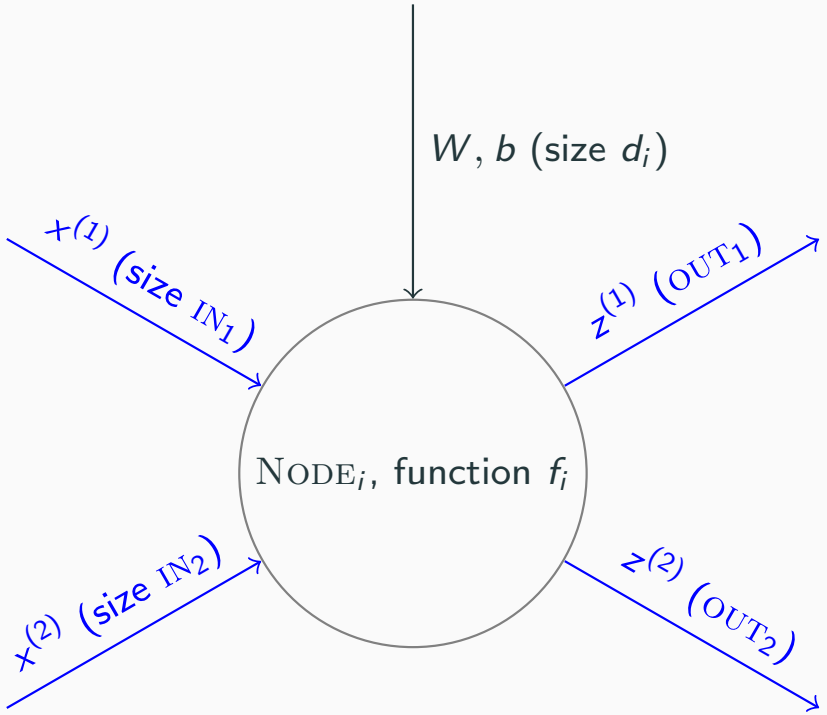
DL training phase: computational DAG



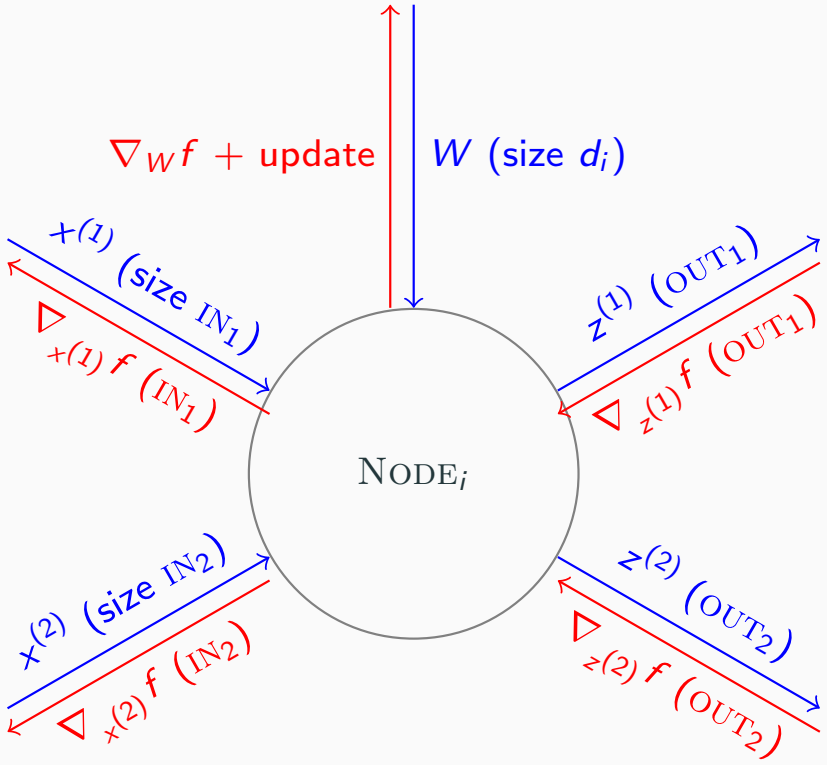
DL training phase: computational DAG

for instance,
$f_i = RELU(Wx + b)$

Node of the DAG



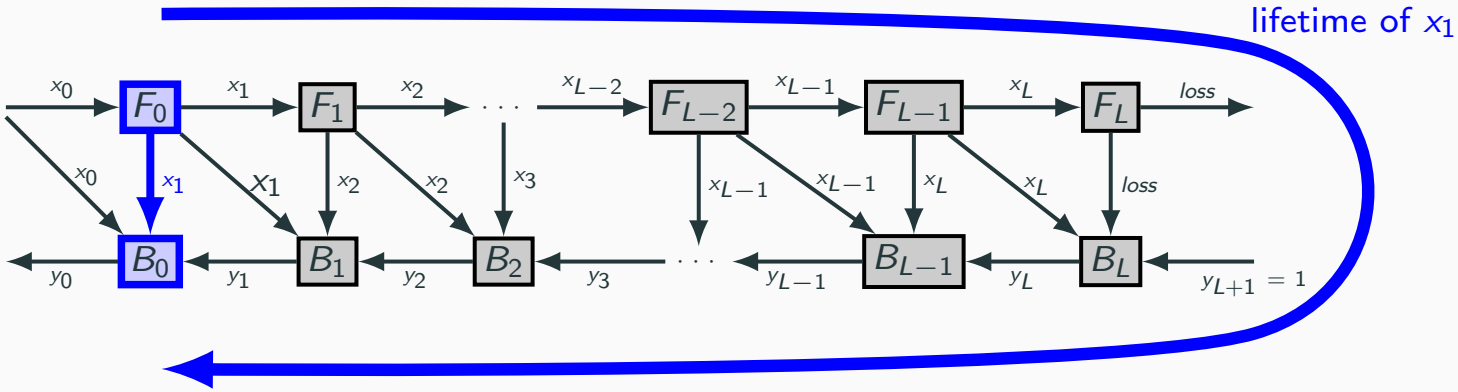
DL: forward propagation and backward propagation



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

Training requires a lot of memory

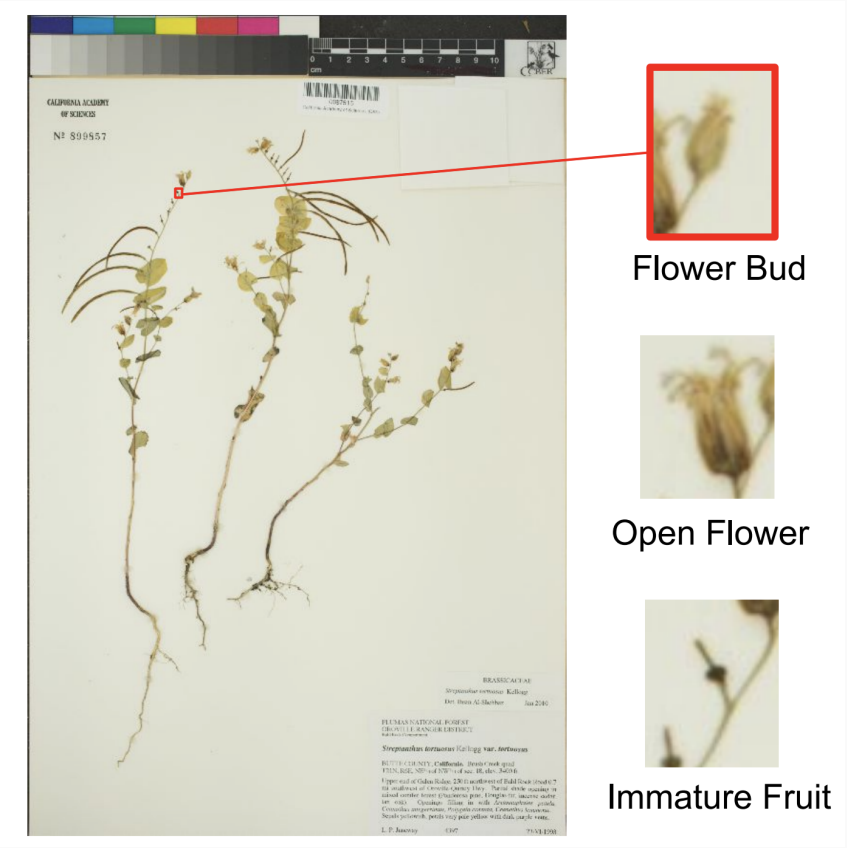
- forward propagation
  - propagate the input through the network to compute loss
- backward propagation
  - compute gradients with respect to loss
  - update the weights with gradients

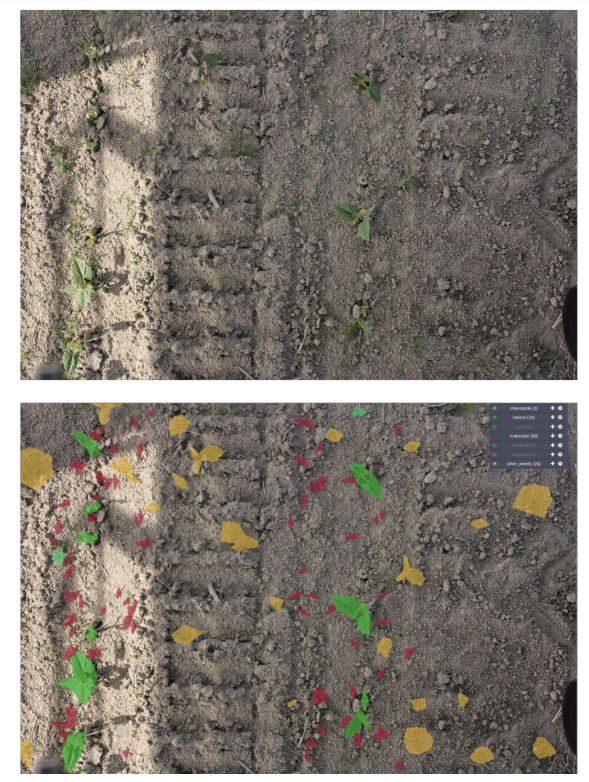


Memory is consumed by storing activations throughout the entire training

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

Pl@ntNet uses machine learning to identify plant species





(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 (PI@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

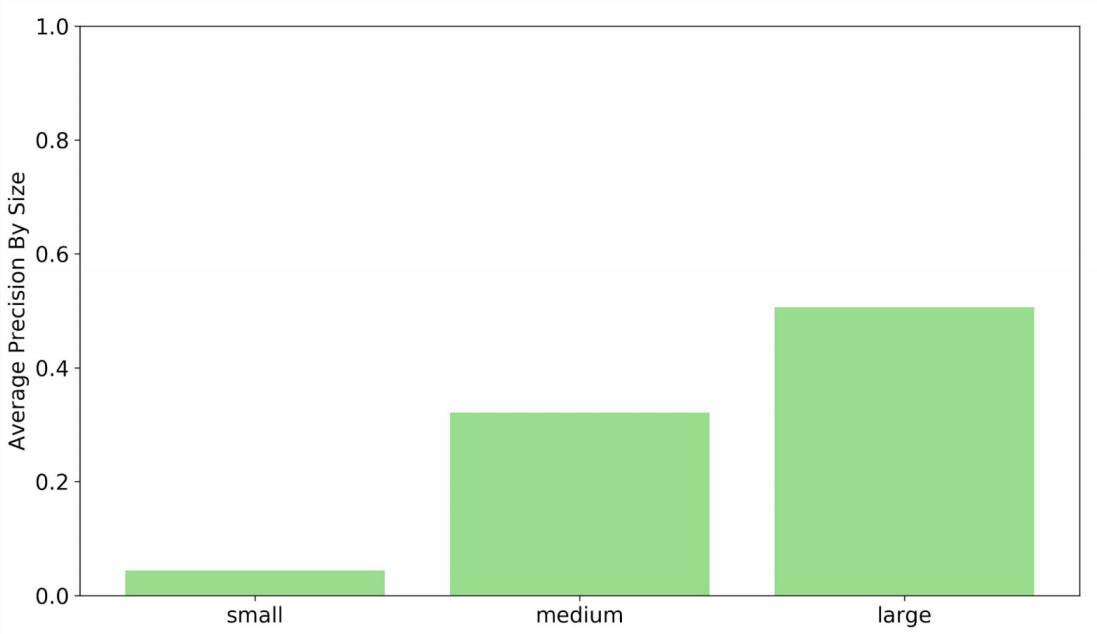


Figure 2: Detection & classification of weeds (performance by object's size)

Memory saving techniques

Special neural networks:

- **Memory efficient architectures:**
  - Reversible Neural Networks (RevNet)
  - Quantized Neural Networks
  - MobileNets
  - ShuffleNet
- **Layer optimization:**
  - memory-efficient batch-normalization layer

Usage of several machines:

- Data Parallelism
- Model Parallelism
- Spatial Parallelism

Memory saving techniques

Efficient training on one node/GPU

- **Rematerialization:**
  - work more and stock less
  - checkpointing strategies
- **Offloading:**
  - use lower memory hierarchy (training on GPU, activations on CPU)

Pros & Cons

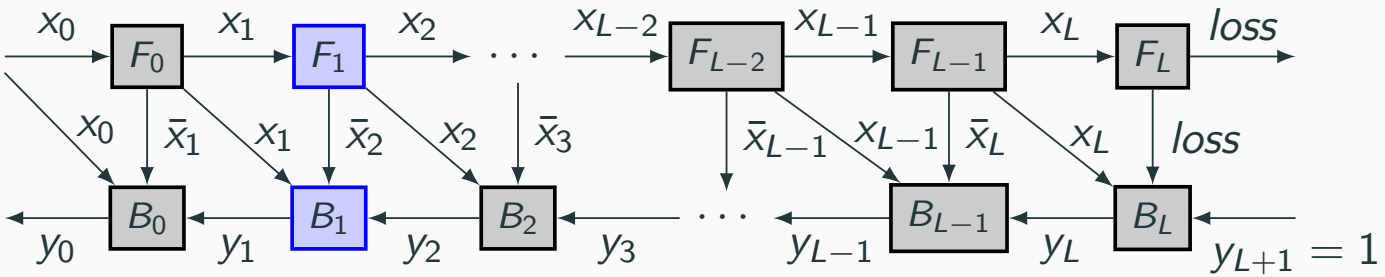
− imposes overhead costs (recomputations or communication delays)

\+ suitable for training any NN architecture with limited resources

\+ could be combined with parallelization strategies

Rematerialization

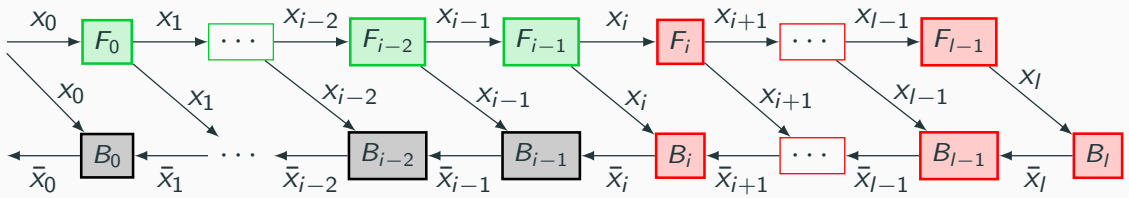
Rematerialization



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

Analogous to Automatic Differentiation
The optimal schedule can be found with the help of Dynamic Programming

Single Adjoint Chain Computation problem



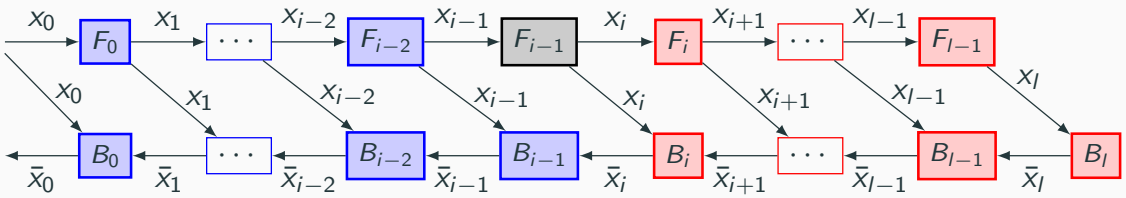
Input: cost of one forward step u_f, cost of one backward step u_b, chain length ℓ and total memory size c.

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

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

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

Single Adjoint Chain Computation problem



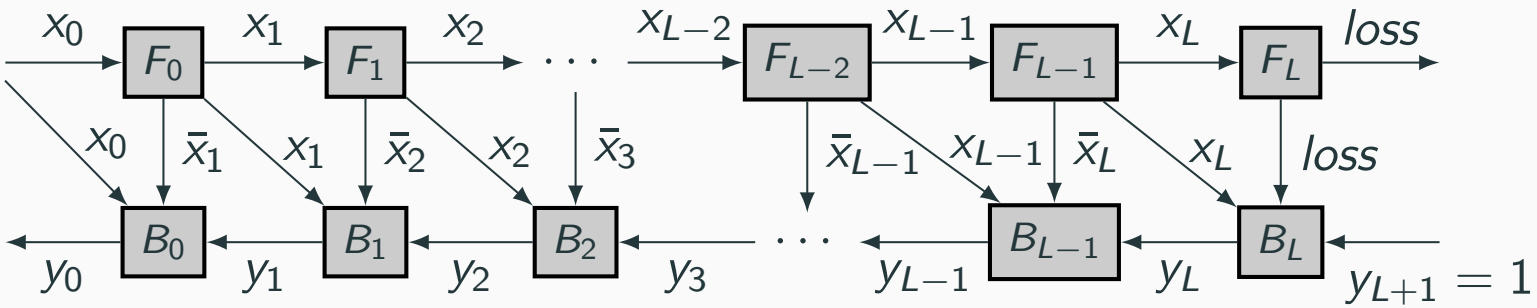
Input: cost of one forward step u_f, cost of one backward step u_b, chain length ℓ and total memory size c.

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

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

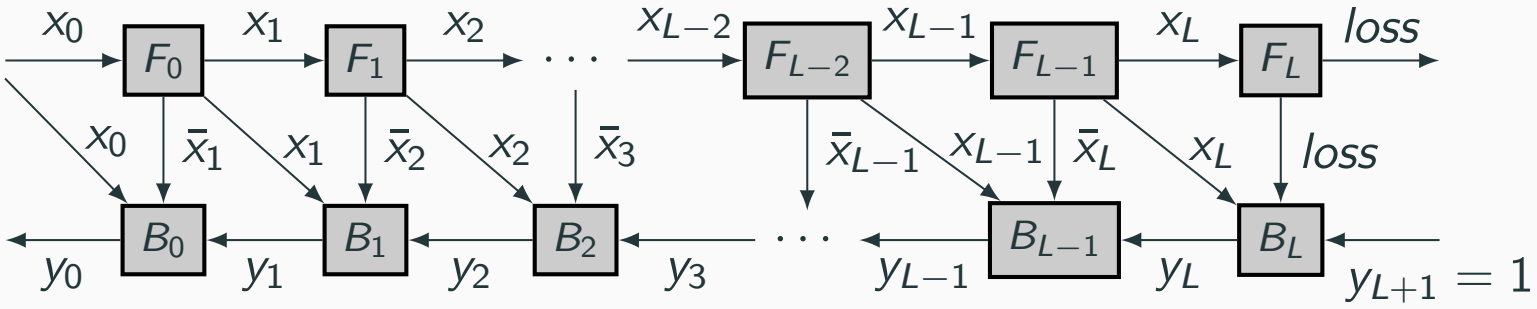
$$\mathrm{Opt}_0(\ell, c) = \min_{1 \leq i \leq \ell-1} \{ i u_f + \mathrm{Opt}_0(\ell - i, c-1) + \mathrm{Opt}_0(i-1, c) \}$$

DNN frameworks checkpointing



- heterogeneous costs
- 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



$$\mathrm{Opt}_{\mathrm{BP}}(i, \ell, m) = \min \begin{cases} \mathrm{Opt}_1(i, \ell, m) \\ \mathrm{Opt}_2(i, \ell, m) \end{cases} \tag{1}$$

$$\mathrm{Opt}_1(i, \ell, m) = \min_{s=i+1,\ldots,\ell} \sum_{k=i}^{s-1} u_f[k] + \mathrm{Opt}_{\mathrm{BP}}(s, \ell, m - \omega_x^s) + \mathrm{Opt}_{\mathrm{BP}}(i, s-1, m)$$

$$\mathrm{Opt}_2(i, \ell, m) = u_f[i] + \mathrm{Opt}_{\mathrm{BP}}\left(i+1, \ell, m - \omega_{\bar{x}}^{i+1}\right) + u_b[i]$$

Formulas are valid when memory constrained are not violated !

Implementation in PyTorch

Parameter estimation

- measure memory and time costs of operations
- $\hookrightarrow$ do a simulation run with some test sample

Computing optimal sequence

- discretize memory costs for dynamic programming.
- find the schedule with the dynamic programming

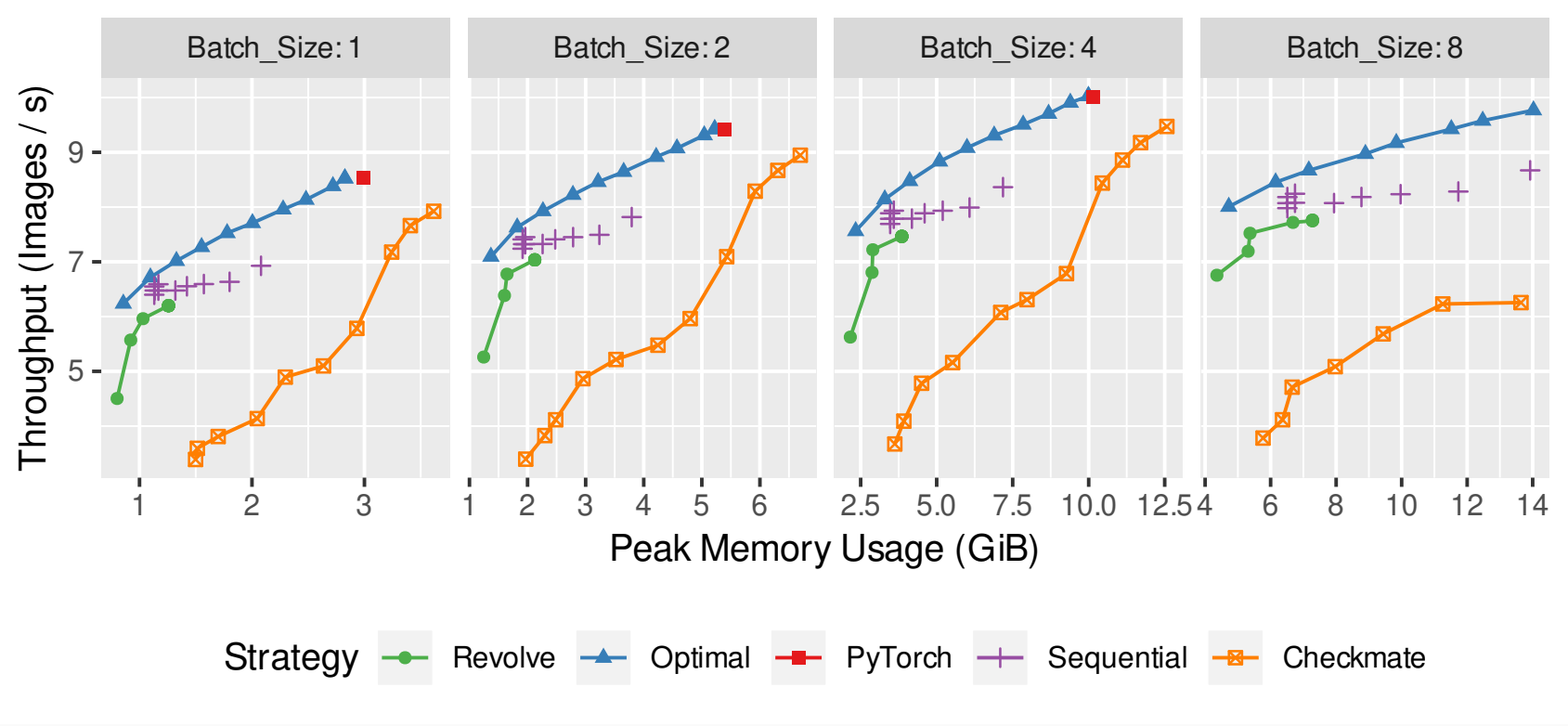
Executing training iteration

- apply the schedule to each iteration of the training
- done by a "wrapper" over NN which controls the operation order.

rotor: Rematerializing Optimally with pyTORch

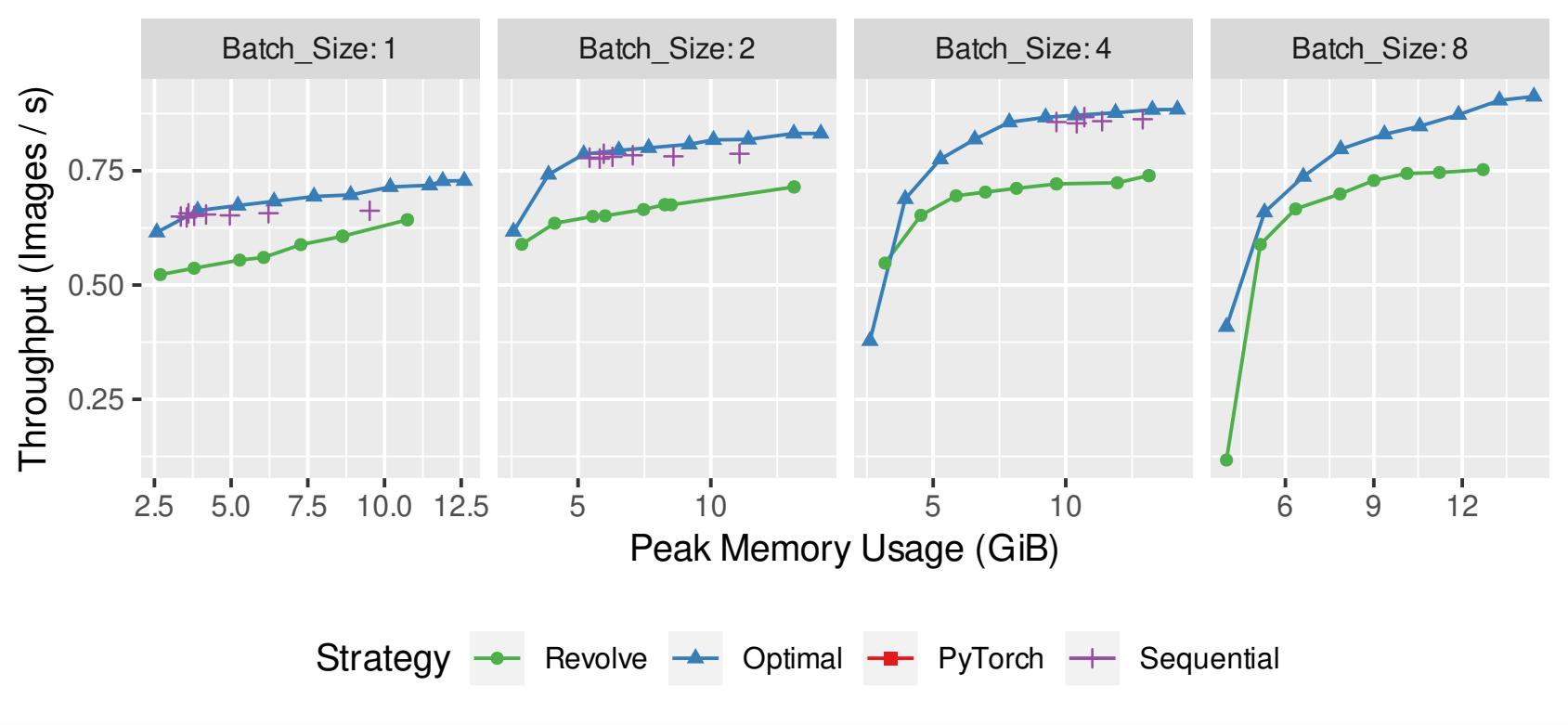
```
import rotor
import torch

device = torch.device("cuda")
net = rotor.models.resnet18()
net.to(device=device)
net_check = rotor.Checkpointable(net)              #create model
shape = (32, 3, 224, 224)
memory = 700*1024*1024                             #memory budget of 700MB
sample = torch.rand(*shape, device=device)
net_check.measure(sample)                          #measure execution costs
net_check.compute_sequence(mem_limit=memory)       #find the schedule

#forward and backward with checkpointing
data = torch.rand(*shape, device=device)
data.requires_grad = True
result = net_check(data).sum()
result.backward()
grad = data.grad
```

Comparison of our implementation with other approaches i



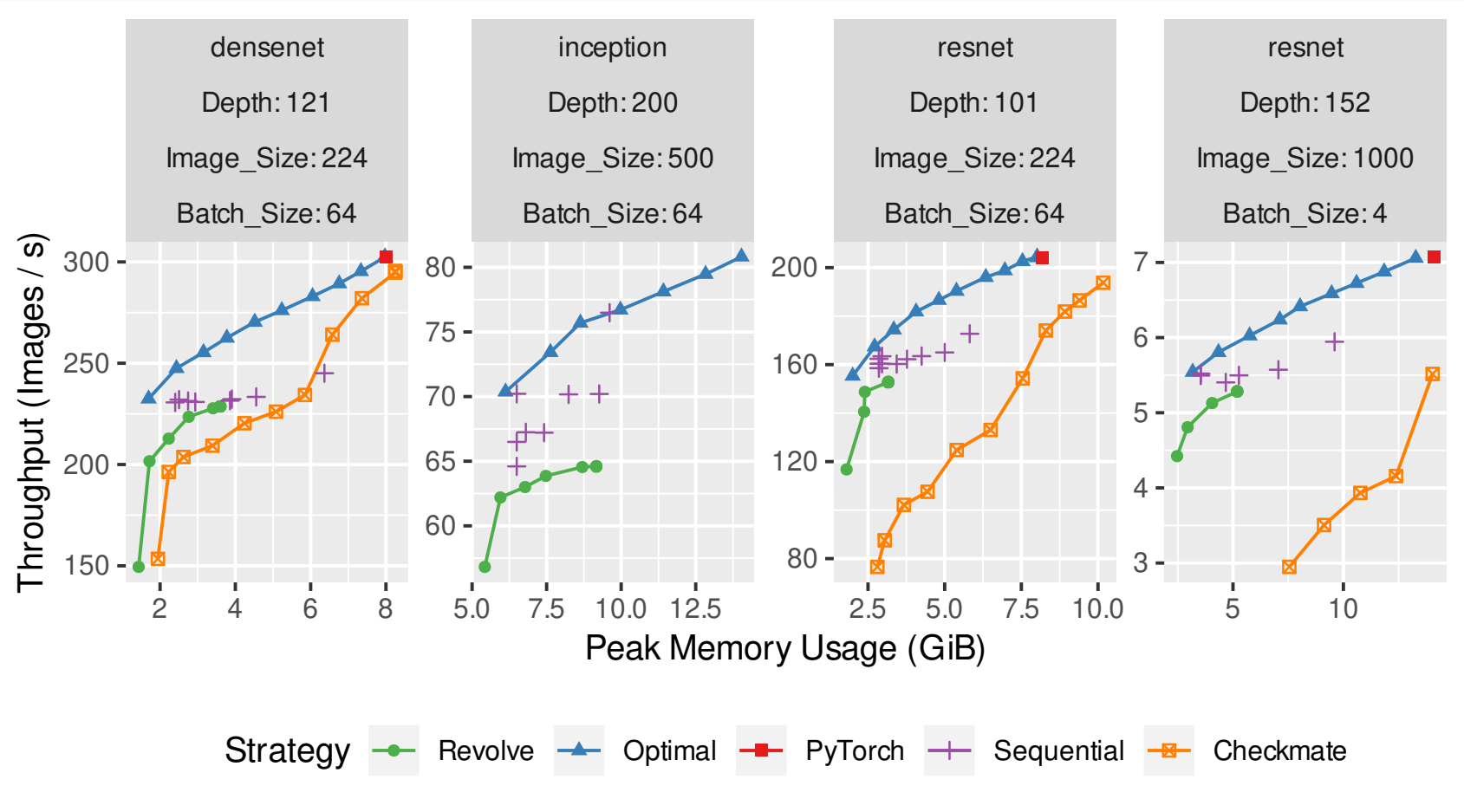
(i) Experimental results for the ResNet network with depth 101 and image size 1000.

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

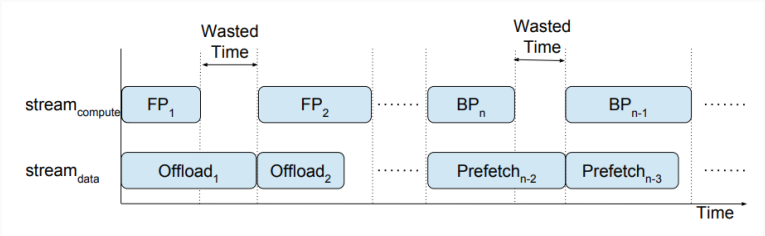


(iii) Experimental results for several situations.

Offloading

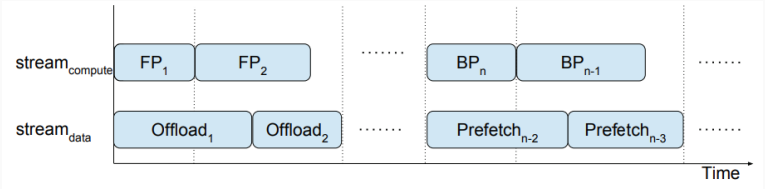
vDNN and vDNN++

vDNN[Rhu et al, 2016]



- what activations to offload:
  - offload everything
  - offload only inputs of convolutional layers
- synchronizes each forward step with the corresponding offloading
- prefetching is done synchronously but as soon as possible

vDNN++[Shriram et al, 2019]



Improvements:

- offloading is performed asynchronously with the forward steps
- solving memory fragmentation issues on the GPU
- decreasing memory needs at CPU by using compression

Other approaches

TFLMS[Tung D Le et al, 2018]

- a module in TensorFlow
- rewrite the computational graph of a neural network
- using swap-in and swap-out operations
- offloading decision is based on distances between nodes

AutoSwap[Zhang et al, 2019]

- CUDA based implementation
- using swap-in and swap-out operations
- decision is based on assigned priority scores
- bayesian optimization is used to optimize the priority scores

Our model

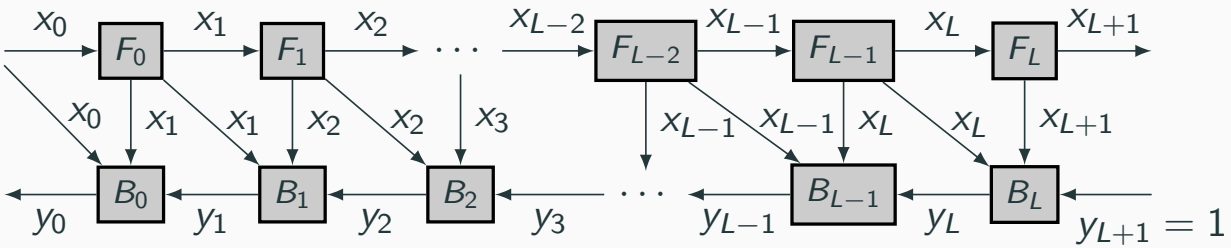


Figure 4: Data dependencies during the training phase of Sequential DNNs.

Problem
We are given

- an adjoint chain with L operations with known
  - processing times $u_f[i]$ and $u_b[i]$
  - data sizes $|x_i|$ and $|y_i|$
- a processing device with memory $M_{\mathrm{GPU}} < \infty$
- a main memory with memory $M_{\mathrm{CPU}} = \infty$
- a network connecting both devices with bandwidth β

Can we execute the given chain on the given platform within time T?

Our contribution

Main problem

- activations are offloaded entirely
- they are discarded only after the offloading is complete

Fractional communications

- activations are offloaded entirely
- already offloaded part of an activation can be immediately discarded

Fractional relaxation

- activations can be offloaded partially
- already offloaded part of an activation can be immediately discarded

Our contribution

Main problem [NP complete in the strong sense]

- activations are offloaded entirely
- they are discarded only after the offloading is complete

Fractional communications

- activations are offloaded entirely
- already offloaded part of an activation can be immediately discarded

Fractional relaxation

- activations can be offloaded partially
- already offloaded part of an activation can be immediately discarded

Our contribution

Main problem [NP complete in the strong sense]

- activations are offloaded entirely
- they are discarded only after the offloading is complete

Fractional communications [NP complete in the weak sense]

- activations are offloaded entirely
- already offloaded part of an activation can be immediately discarded

Solution: Dynamic Programming

Fractional relaxation

- activations can be offloaded partially
- already offloaded part of an activation can be immediately discarded

Our contribution

Main problem [NP complete in the strong sense]

- activations are offloaded entirely
- they are discarded only after the offloading is complete

Fractional communications [NP complete in the weak sense]

- activations are offloaded entirely
- already offloaded part of an activation can be immediately discarded

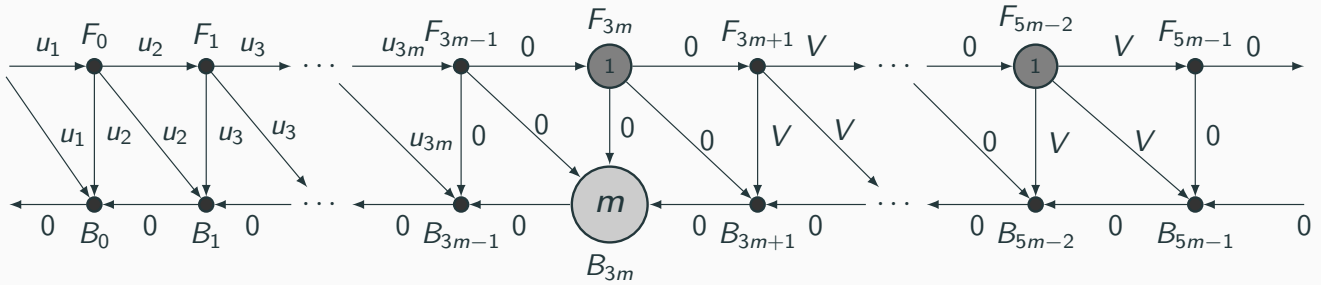
Solution: Dynamic Programming

Fractional relaxation [Polynomial]

- activations can be offloaded partially
- already offloaded part of an activation can be immediately discarded

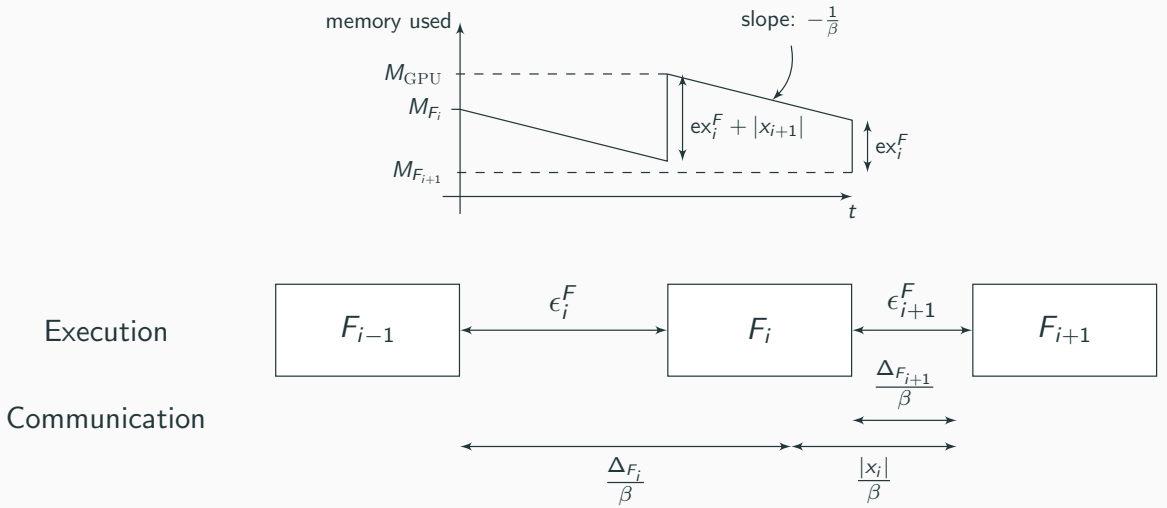
Solution: Greedy algorithm

Main Problem: NP-completeness in the strong sense



- $L = 5m$, $\beta = V$, $M_{\mathrm{GPU}} = mV$, $T = 2m$;
- $u_f[i] = 0$ and $|x_i| = u_i$ for $1 \leq i < 3m$;
- $u_f[i] = 1$ and $|x_i| = 0$ for $i = 3m + 2k, 0 \leq k < m$;
- $u_f[i] = 0$ and $|x_i| = V$ for $i = 3m + 2k + 1, 0 \leq k < m$;
- $u_b[i] = 0$ and $|y_i| = 0$ for all i, except $u_b[3m] = m$.

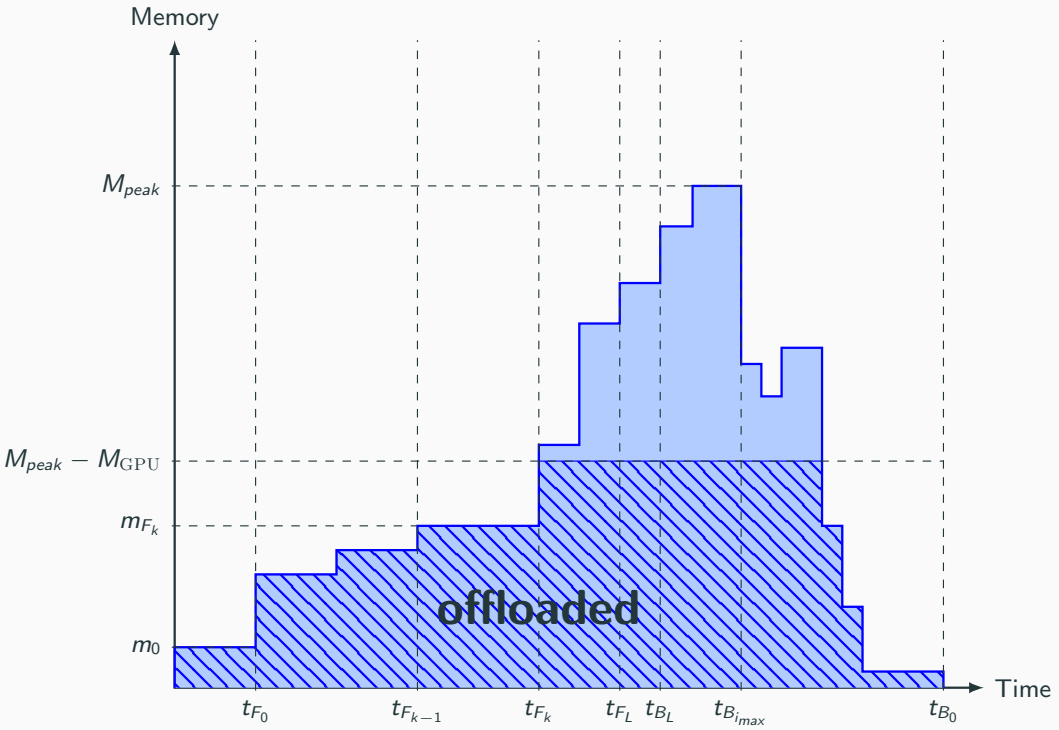
Fractional communications



- find ϵ_i^F using $\epsilon_i^F = \max\left(0, \frac{M_{F_i} + \text{ex}_i^F + |x_{i+1}| - M_{\text{GPU}}}{\beta}\right)$
- update $M_{F_{i+1}}$ and $\Delta_{F_{i+1}}$
- find total idle time starting from F_0, taking the decision layer by layer
- keep track of all decisions and eliminate invalid ones
- symmetrical for the backward phase

Fractional relaxation

Greedy algorithm finds a solution in this case



The schedule is

- no-wait
- eager
- ordered
- offloading $M_{peak} - M_{\mathrm{GPU}}$ data

no-wait - perform operations ASAP

eager - offload first activations

ordered - offload in increasing order of indices

Results

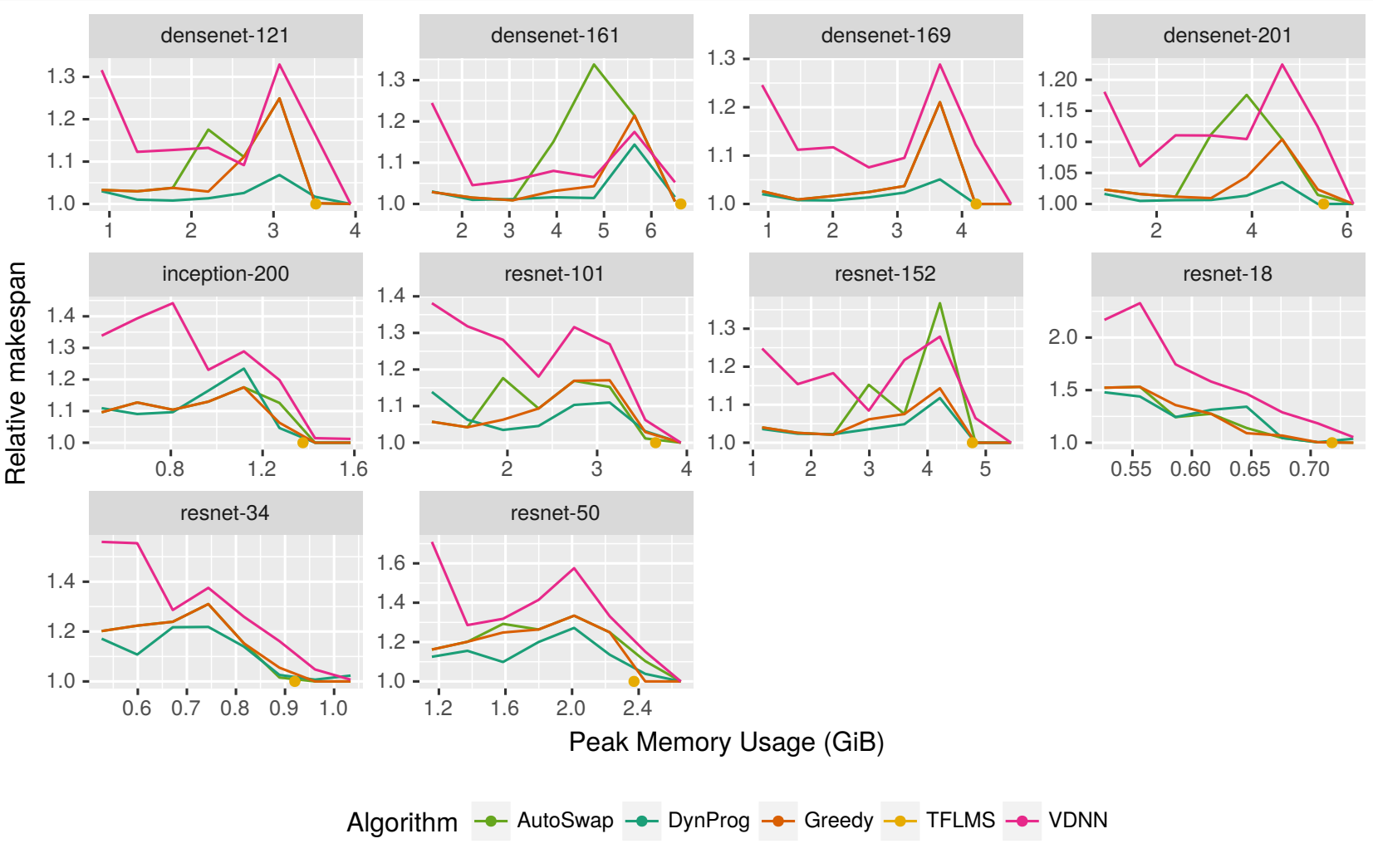


Figure 5: Experimental results (image size 224 and batch size 32). The plots show the ratio of the makespan to the lower bound $LB = \max(\sum_i u_f[i] + u_b[i], 2\frac{M_{peak} - M_{\text{GPU}}}{\beta})$

Results

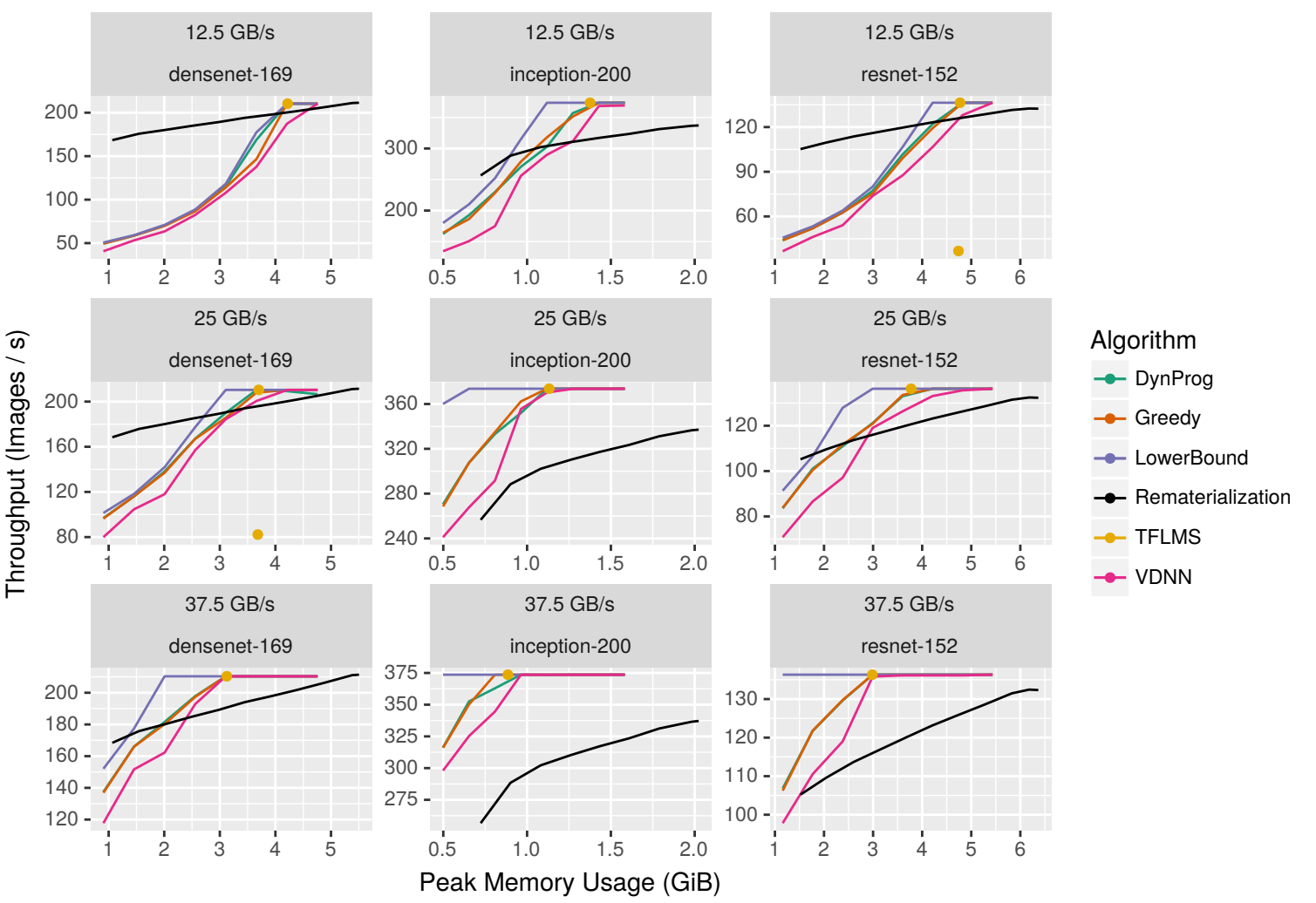


Figure 6: Offloading vs Rematerialization (image size 224 and batch size 32). Lower bound is defined with $LB = \max(\sum_i u_f[i] + u_b[i], 2\frac{M_{peak} - M_{\text{GPU}}}{\beta})$

Conclusion

- It is important to reduce memory consumption
- Storing activations could be more expensive than storing weights
- Rematerialization is an efficient way to store less activations ($2|x_i|/\beta \geq u_f[i] + u_b[i]$)
- Offloading is an efficient way to store less activations ($2|x_i|/\beta \leq u_f[i] + u_b[i]$)
- Our solutions outperform the state of the art techniques

What next?

- combine Offloading with Rematerialization
- combine them with parallelization techniques
- apply these techniques on real use-cases (PI@ntNet)

The end

Thank you for the attention !