université de BORDEAUX Inria LaBRI

Memory Saving Strategies for Deep Neural Network Training

Workshop ACDA
Applied and Computational Discrete Algorithms

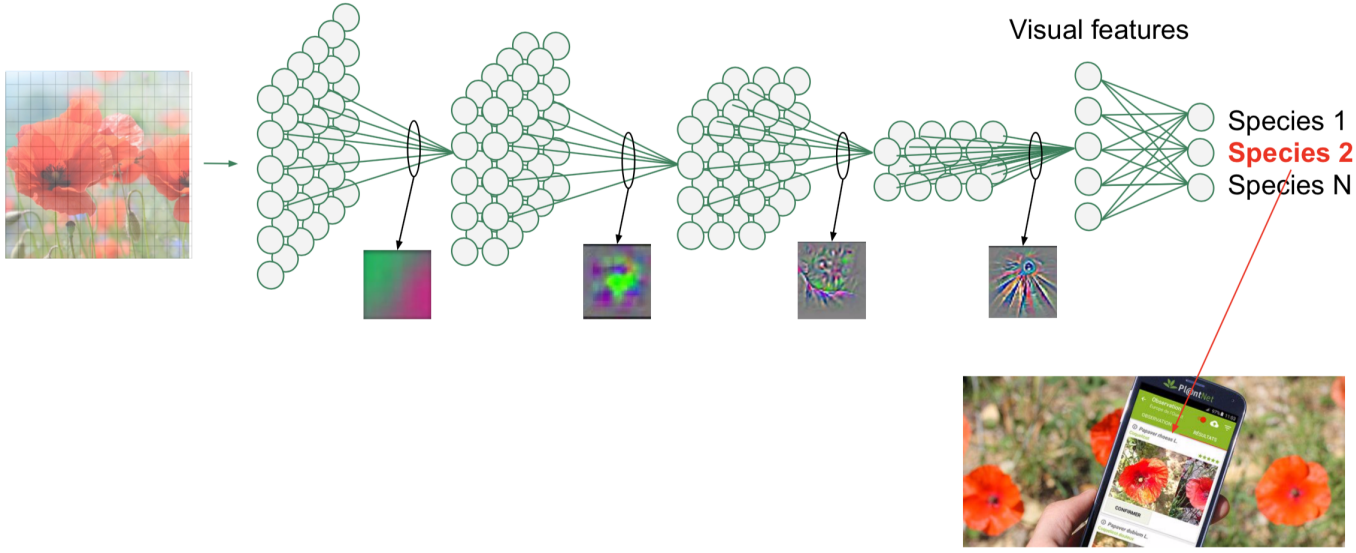
Olivier Beaumont

General Overview and Joint Works with
Alexis Joly, Lionel Eyraud-Dubois, Alena Shilova and Xunyi Zhao

September 8, 2022

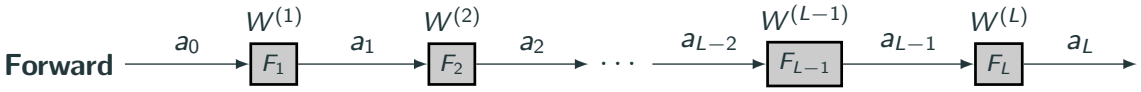
Introduction

Context



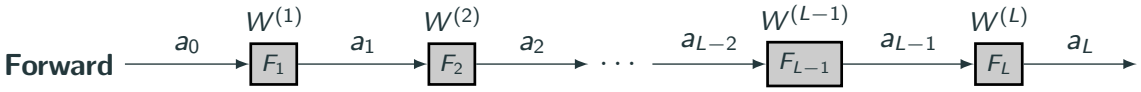
- Collaboration between two teams:
  - HiePACS (HPC team) in Bordeaux
  - Zenith (Big Data/AI team, Pl@ntNet) in Montpellier
- Pl@ntNet is a platform that identify plants from pictures
- Pl@ntNet relies on Deep Neural Networks
- it plans to go larger (better model, more species, better accuracy)
- the training is thus more time and memory-consuming
  - Time Consuming: increase parallelism
  - Memory Saving: parallelism is not enough, need for dedicated strategies

Inference and training as task graphs



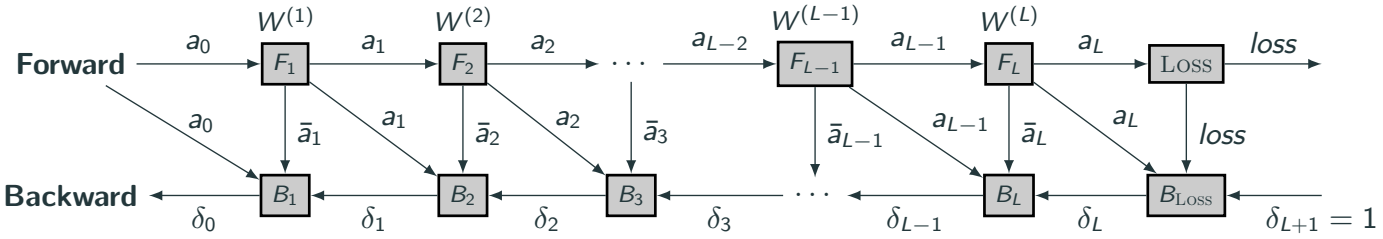
- a_0 is raw data, *e.g.* an image of a plant 
- a_0 could be composed of several samples, forming a **batch**
- a_L is the neural network **prediction**, *e.g.* probability of iris
- F_i is the operation of layer i used for prediction (**inference**)
- $W^{(i)}$ is the parameter (**weights**) of layer i

Inference and training as task graphs



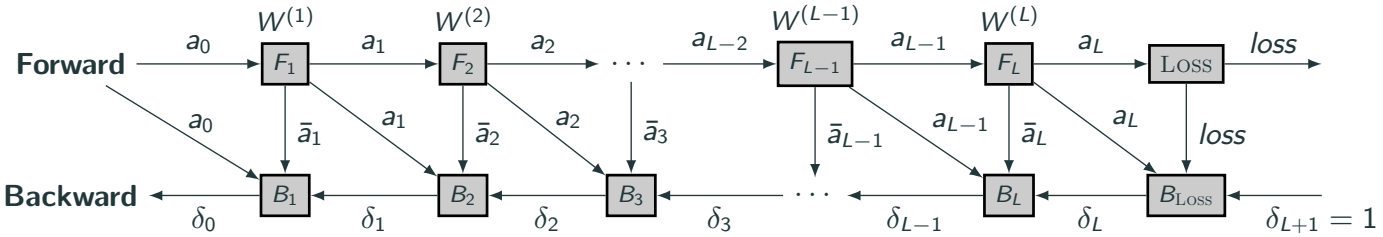
- a_0 is raw data, *e.g.* an image of a plant 
- a_0 could be composed of several samples, forming a **batch**
- a_L is the neural network **prediction**, *e.g.* probability of iris
- F_i is the operation of layer i used for prediction (**inference**)
- $W^{(i)}$ is the parameter (**weights**) of layer i
- at first, weights are **random**, thus prediction is erroneous

Inference and training as task graphs



- a_0 is raw data, *e.g.* an image of a plant 
- a_0 could be composed of several samples, forming a **batch**
- a_L is the neural network **prediction**, *e.g.* probability of iris
- F_i is the operation of layer i used for prediction (**inference**)
- $W^{(i)}$ is the parameter (**weights**) of layer i
- at first, weights are **random**, thus prediction is erroneous
- *loss* is the **error** of the model; the lower, the better
- B_i is the operation of layer i used for updating weights

Inference and training as task graphs

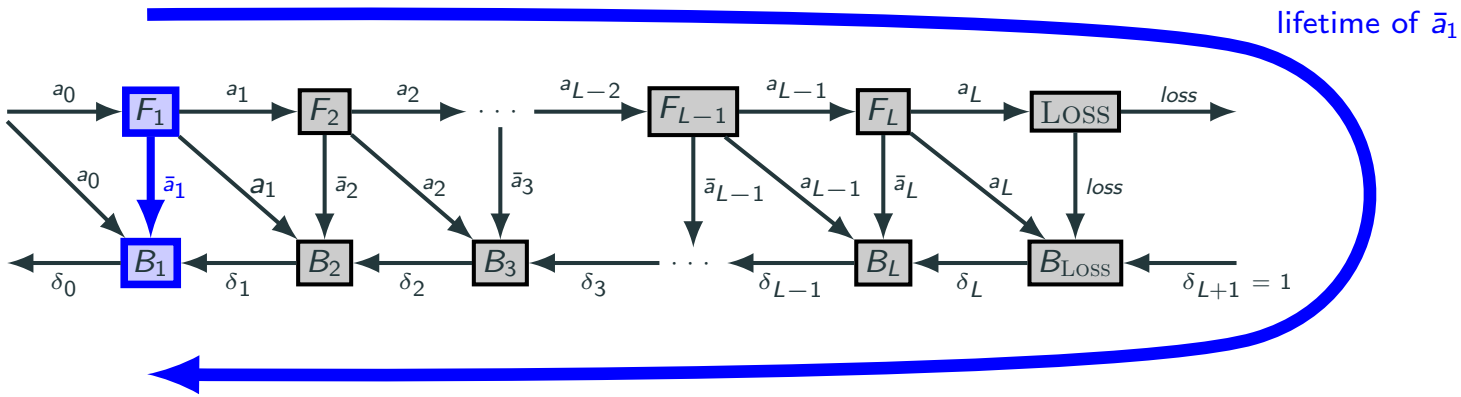


- a_0 is raw data, *e.g.* an image of a plant 
- a_0 could be composed of several samples, forming a **batch**
- a_L is the neural network **prediction**, *e.g.* probability of iris
- F_i is the operation of layer i used for prediction (**inference**)
- $W^{(i)}$ is the parameter (**weights**) of layer i
- at first, weights are **random**, thus prediction is erroneous
- *loss* is the **error** of the model; the lower, the better
- B_i is the operation of layer i used for updating weights
- model is **trained** by iterating **forward-backward propagation**

Memory for training

Default settings

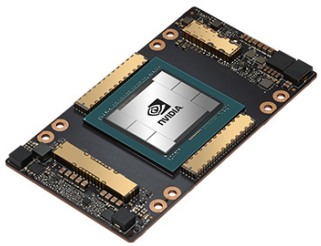
- Store **weights** $W^{(i)}$ + **gradients of weights** + **optimizer states**
- **activations** (a_i and $\bar{a}_i$) are required for backward propagation
- some activations have a long lifetime



Memory is consumed by activations throughout the entire training!

About hardware

- GPUs and TPUs are used for training
- they become more performant and their memory increases
  - *e.g.* Nvidia A100



  - $\sim$ 250 times faster than one CPU, memory 80GB
- still not enough to process some DNNs
- buying a new GPU is not always an option
  - very expensive
  - negative Carbon Impact (CI)

Memory Saving Techniques – Rematerialization on Chains

Memory saving techniques. Single GPU

Rematerialization (gradient checkpointing)

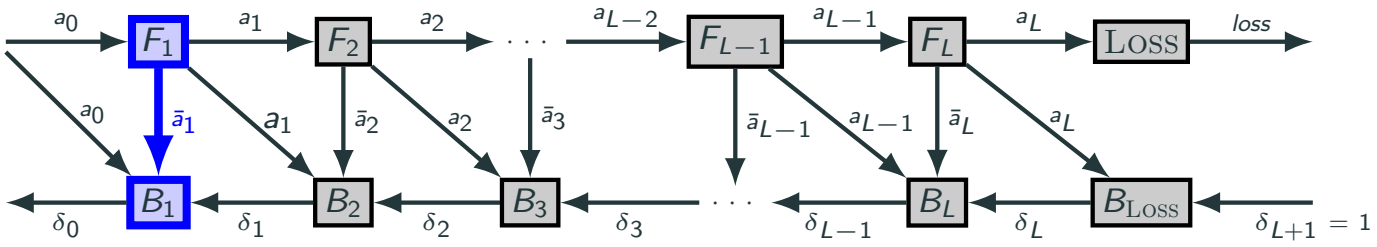
- delete some activations and recompute them during the backward
- less storage, more computations

Offloading

- move activations from memory of GPU to CPU
- can be applied to weights too
- bandwidth-bounded
- but when bandwidth is high, then zero overhead

Both methods produce the exact same results for training!

Rematerialization (linear chains)



Main idea

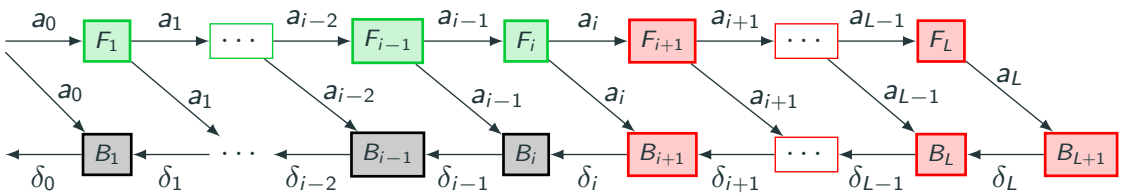
To work more and store less: instead of keeping all activations, store some of them, delete the others and recompute them when needed.

Analogy with Automatic Differentiation

This technique is very common in AD for homogeneous chains.

Homogeneous chain computation problem (Revolve)

Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation, A. Griewank et al., ACM TOMS'00



Input: forward step cost $\mathbf{u_F}$, backward step cost $\mathbf{u_B}$
activation/gradient size is $\mathbf{1}$
chain length $\ell = \mathbf{L}$, total memory size $\mathbf{m}$.

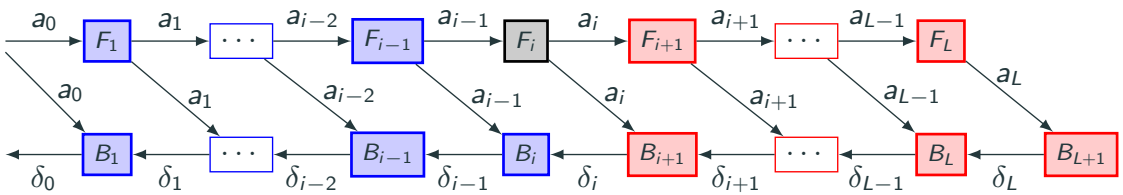
$$\mathrm{Opt}_0(\ell, 1) = \frac{\ell(\ell+1)}{2} u_F + (\ell+1) u_B$$

$$\mathrm{Opt}_0(1, m) = u_F + 2u_B$$

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

Homogeneous chain computation problem (Revolve)

Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation, A. Griewank et al., ACM TOMS'00



Input: forward step cost $\mathbf{u_F}$, backward step cost $\mathbf{u_B}$

activation/gradient size is $\mathbf{1}$

chain length $\ell = \mathbf{L}$, total memory size $\mathbf{m}$.

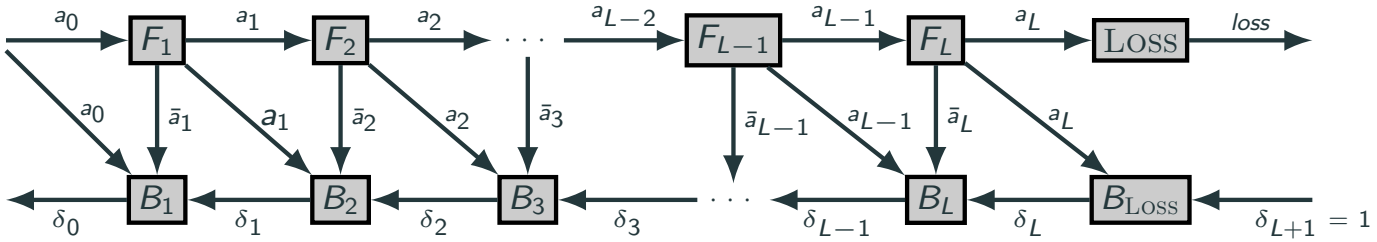
$$\mathrm{Opt}_0(\ell, 1) = \frac{\ell(\ell+1)}{2} u_F + (\ell+1) u_B$$

$$\mathrm{Opt}_0(1, m) = u_F + 2u_B$$

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

Rematerialization in DNN frameworks

Source: `https://gitlab.inria.fr/hiepacs/rotor`



DNN characteristics

- Main differences with Automatic Differentiation
  - Extra dependencies ($\downarrow$-edges) + heterogeneous weights
  - No closed form formula to find the checkpoints: NP-Completeness results
- different ways of saving: recording or saving only input (PyTorch)

Rematerialization operations

- F_i^n computes the output of F_i, and forgets the input.

```
with torch.no_grad():
    x = F[i](x)
```

- F_i^c computes the output of F_i, and keeps the input.

```
with torch.no_grad():
    y = F[i](x)
```

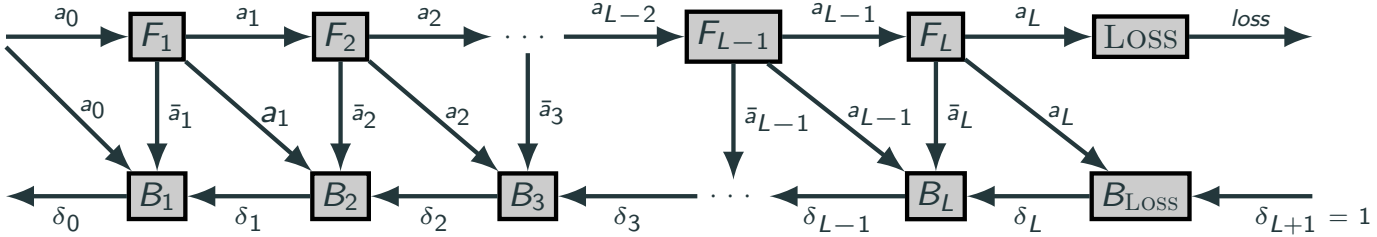
- F_i^e computes the output of F_i, enabling gradient computation.

```
with torch.enable_grad():
    y = F[i](x)
```

- B_i computes the backward of layer i.

```
y.backward(g)
g = x.grad
```

Persistent dynamic programming solution



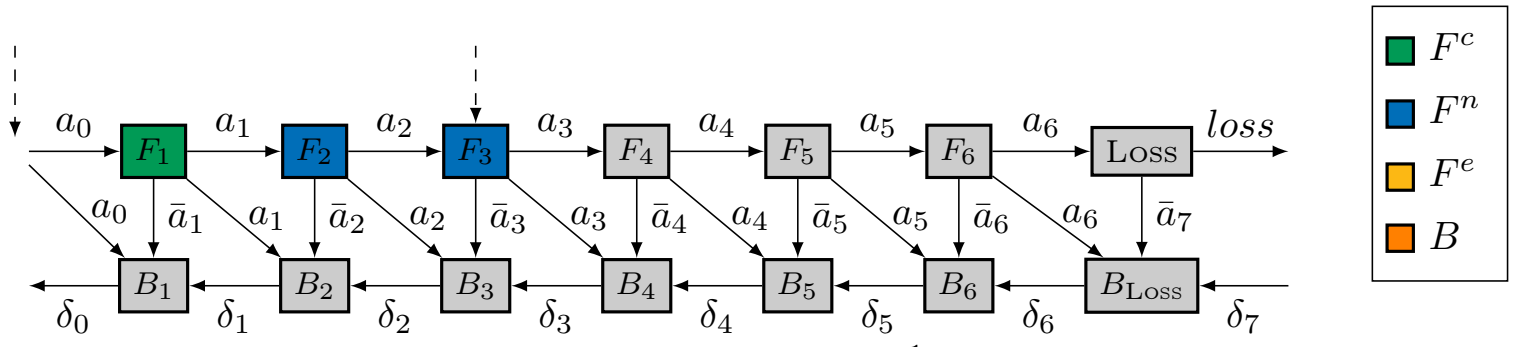
$$\text{Opt}_{\text{BP}}(i, \ell, m) = \min \begin{cases} \text{Opt}_{F^c}(i, \ell, m) \\ \text{Opt}_{F^e}(i, \ell, m) \end{cases}$$

$$\text{Opt}_{F^c}(i, \ell, m) = \min_{s=i,\dots,\ell-1} \sum_{k=i}^{s} u_{F_k} + \text{Opt}_{\text{BP}}(s+1, \ell, m - a_s) + \text{Opt}_{\text{BP}}(i, s, m)$$

$$\text{Opt}_{F^e}(i, \ell, m) = u_{F_i} + \text{Opt}_{\text{BP}}(i+1, \ell, m - \bar{a}_i) + u_{B_i}$$

If memory constraints are violated, then $\text{Opt}_{\text{BP}}(i, \ell, m) = \infty$!

Rematerialization sequence. Toy example



Sequence

$F_1^c, F_2^n, F_3^n, F_4^e, F_5^e, F_6^e, \text{Loss}, B_{\text{Loss}}, B_6, B_5, B_4, F_1^c, F_2^n, F_3^e, B_3, F_1^e, F_2^e, B_2, B_1$

Toy network

1. Linear(2000, 2500)
2. Linear(2500, 2800)
3. Linear(2800, 2900)
4. Linear(2900, 2800)
5. Linear(2800, 2500)
6. Linear(2500, 2000)

Peak memory	107 MB
Memory limit	90 MB
Batch size	1000
Makespan	163.1 ms (1.33)
Memory	86.4 MB (0.81)

Not representative for other DNNs

Implementation in PyTorch

Parameter estimation

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

Compute optimal sequence

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

Execute 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

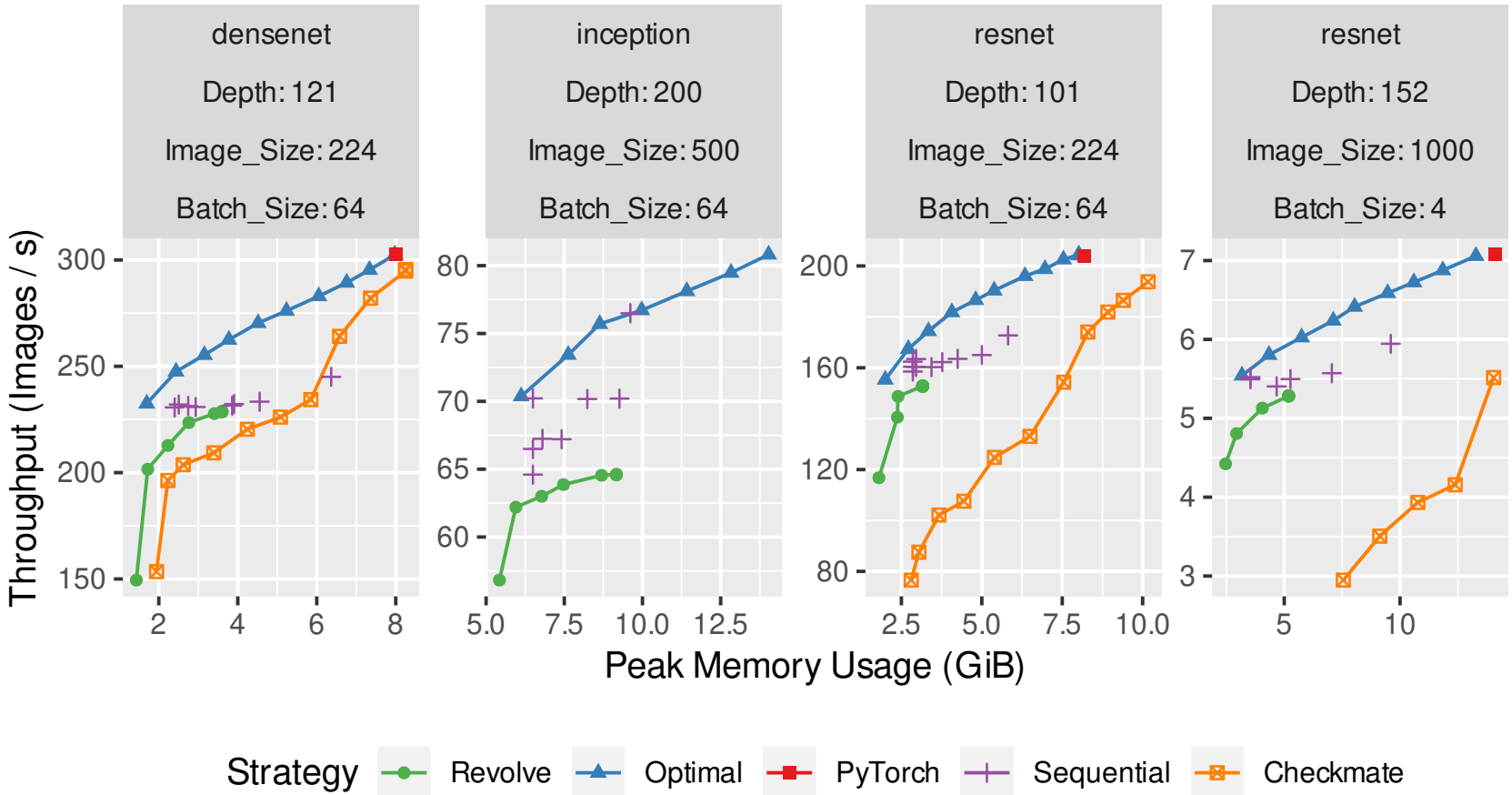
```
import rotor
import torch

device = torch.device("cuda")
net = rotor.models.resnet18().to(device)          #Build model
shape = (32, 3, 224, 224)
memory = 700*1024*1024                          #700MB memory limit

#Wrap model in rotor:
#Automatically measure and compute optimal sequence
net_check = rotor.Checkpointable(net, mem_limit=memory)

#Forward and Backward is unchanged, with rematerialization
data = torch.rand(*shape, device=device)
data.requires_grad = True
result = net_check(data).sum()
result.backward()
grad = data.grad
```

Comparison of `rotor` with other approaches



(i) Experimental results for several situations.

Memory Saving Techniques – Rematerialization on General DAGs

ILP based solution: Checkmate

Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization, Proceedings of Machine Learning and Systems (MLSys 2020), Paras Jain et al.

- Checkmate assumes that there is a natural evaluation ordering of nodes
- Step t starts after the evaluation of node $t-1$ and ends with evaluation of node t
- $S_{t,i}$ says if the result of i is kept at the end of step $t-1$
- $R_{t,i}$ says if the result of i is recomputed during step t

$$\begin{aligned}
&\underset{R,S}{\arg\min} && \sum_{t=1}^{n}\sum_{i=1}^{t} C_i R_{t,i} && && \text{(1a)}\\
&\text{subject to} \\
&\quad R_{t,j} \le R_{t,i} + S_{t,i} && && \forall t\, \forall (v_i, v_j) \in E, && \text{(1b)}\\
&\quad S_{t,i} \le R_{t-1,i} + S_{t-1,i} && && \forall t \ge 2\, \forall i, && \text{(1c)}\\
&\quad \textstyle\sum_i S_{1,i} = 0, && && && \text{(1d)}\\
&\quad \textstyle\sum_t R_{t,n} \ge 1, && && && \text{(1e)}\\
&\quad R_{t,i}, S_{t,i} \in \{0,1\} && && \forall t\, \forall i && \text{(1f)}
\end{aligned}$$

- $+$ additional constraints to determine when memory can be released

ILP based solution: Checkmate

Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization, Proceedings of Machine Learning and Systems (MLSys 2020), Paras Jain et al.

- Checkmate assumes that there is a natural evaluation ordering of nodes
- Step t starts after the evaluation of node $t-1$ and ends with evaluation of node t
- $S_{t,i}$ says if the result of i is kept at the end of step $t-1$
- $R_{t,i}$ says if the result of i is recomputed during step t

$$\begin{aligned}
&\underset{R,S}{\arg\min} && \sum_{t=1}^{n}\sum_{i=1}^{t} C_i R_{t,i} && && \text{(1a)}\\
&\text{subject to} \\
&\quad R_{t,j} \le R_{t,i} + S_{t,i} && && \forall t\, \forall (v_i, v_j) \in E, && \text{(1b)}\\
&\quad S_{t,i} \le R_{t-1,i} + S_{t-1,i} && && \forall t \ge 2\, \forall i, && \text{(1c)}\\
&\textstyle\sum_i S_{1,i} = 0, && && && \text{(1d)}\\
&\textstyle\sum_t R_{t,n} \ge 1, && && && \text{(1e)}\\
&R_{t,i}, S_{t,i} \in \{0,1\} && && \forall t\, \forall i && \text{(1f)}
\end{aligned}$$

- $+$ additional constraints to determine when memory can be released
- Issues: limited to small size networks $+$ ordering

Treewidth decomposition based rematerialization

Efficient Rematerialization for Deep Networks, NeurIPS'19, Ravi Kumar et al.

- Targets general DAGs (not specifically forward-backward graphs)
- Assumes that treewidth decomposition of the graph is known (treewidth k)
- Assumes that all n output files have unit size
- With n nodes, it builds a schedule of length $O(kn^{\log k})$ with peak memory usage $O(k \log n)$
- Main result: it shows that treewidth helps to design optimal schedules.
- Issues: does not target a specific memory limit + assumes that the decomposition is known

Rematerialization. Conclusion

Summary

- allows to save memory at the cost of recomputations
- schedules can be found with dynamic programming for chains (`rotor` tool)
- reduces memory twice for 15-20% more computations

Perspectives

- Solutions for more general DAGs are highly needed
  - Checkmate: too expensive
  - Treewidth decompositions: too many assumptions
- Partial solution:
  - Hybridation of Rotor and Checkmate is possible
  - consider DAGs as chains of complex sub-networks
  - seems to work well on transformer-based graphs
- In terms of complexity
  - weakly NP-hard for weighted chains (+ Dynamic Programming Solution)
  - no strongly NP-hard result for general (homogeneous) chains

Offloading

Offloading

source: Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training, O Beaumont et al., EuroPar 2020

Main idea

By choosing

- which activations to send to CPU and
- when to offload and prefetch them,

execute training under memory limit M_{GPU} in minimal time.

Time overhead

- If the next operation does not fit into memory, then introduce idle time to wait some tensors to finish their offloading.
- If the next operation requires data not yet prefetched from CPU, then introduce idle time to wait until its prefetching is completed.

Offloading operations

PyTorch mechanisms

- Asynchronous data transfers with a dedicated stream:

```
comm_stream = torch.cuda.Stream(torch.device('cuda'))
```

- Capturing saved tensors in PyTorch:

```
saved_tensors_hooks(pack_hook, unpack_hook)
```

Operations

- O_i offloads activation a_i. Equivalent to:

```
with torch.cuda.stream(comm_stream):
    with torch.no_grad():
        x_cpu.copy_(x_gpu, non_blocking=True)
```

- P_i prefetches activation a_i. Equivalent to:

```
with torch.cuda.stream(comm_stream):
    with torch.no_grad():
        x_gpu.copy_(x_cpu, non_blocking=True)
```

Combination of Offloading and Rematerialization

source: Efficient Combination of Rematerialization and Offloading for Training DNNs, O Beaumont et al., NeurIPS 2021

Problem statement

Find a sequence of operations F_i^c, F_i^n, F_i^e, B_i, O_i and P_i that processes a training iteration under a memory limit M_{GPU} in minimal time.

Our contribution: POFO

Optimal combination of Offloading and Rematerialization under the assumptions above, found with **dynamic programming**.

Sequence (high bandwidth)

$F_1^e, O_1, F_2^e, O_2, F_3^e, F_4^e, F_5^e, F_5^e, \text{LOSS}, B_{\text{LOSS}} B_6, B_5, B_4, P_2, B_3, P_1, B_2, B_1$

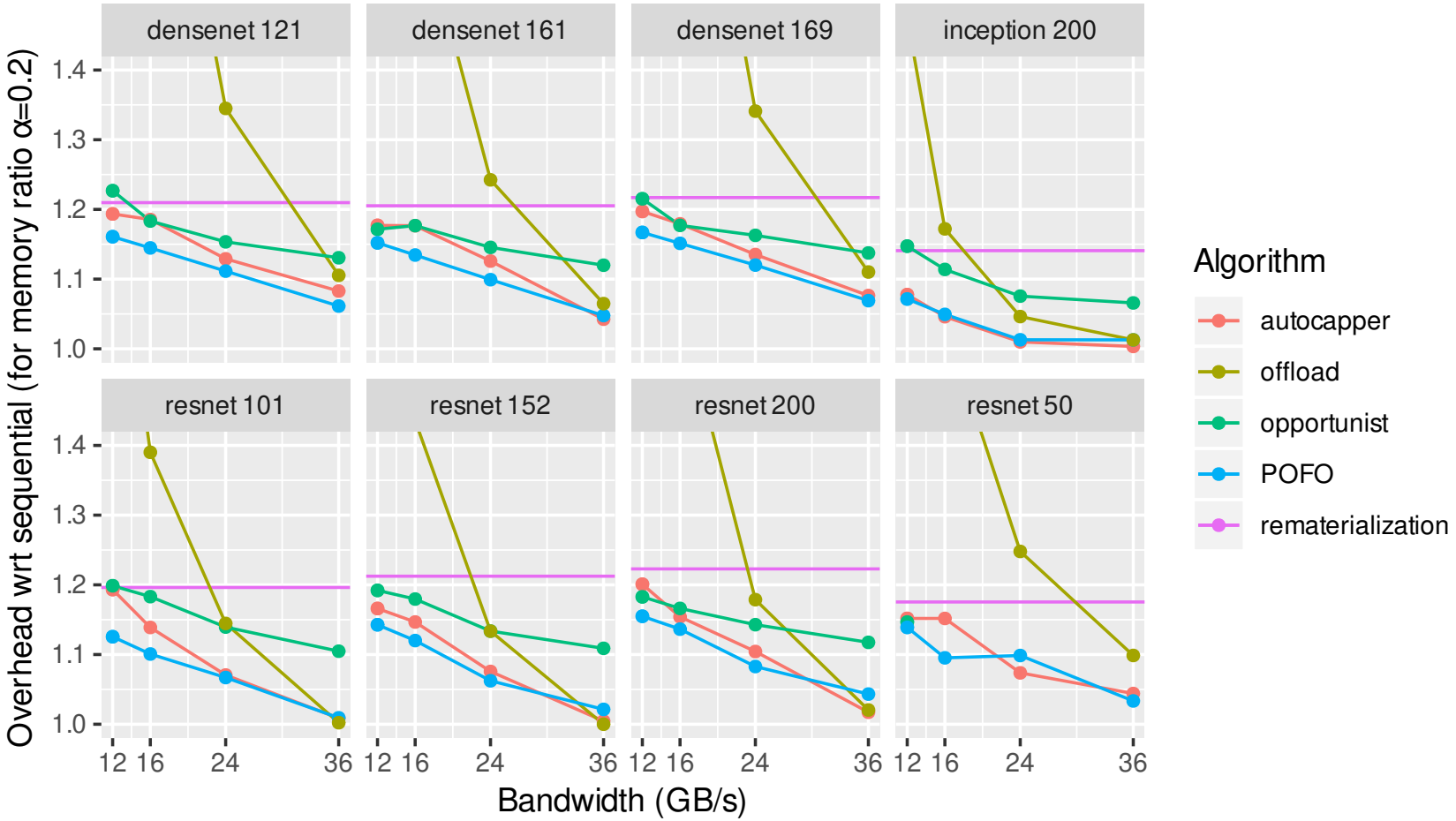
Sequence (low bandwidth)

$F_1^c, F_2^n, O_2, F_3^e, F_4^e, F_5^e, F_5^e, \text{LOSS}, B_{\text{LOSS}} B_6, B_5, B_4, P_2, B_3, F_1^e, F_2^e, B_2, B_1$

Peak memory	107 MB
Memory limit	90 MB
Batch size	1000
Makespan	124.3 ms (≈ 1)
Memory	87.3 MB (0.82)

Peak memory	107 MB
Memory limit	90 MB
Batch size	1000
Makespan	160.6 ms (1.31)
Memory	86.4 MB (0.81)

Simulation results. Memory ratio $= 1/5$ of peak memory

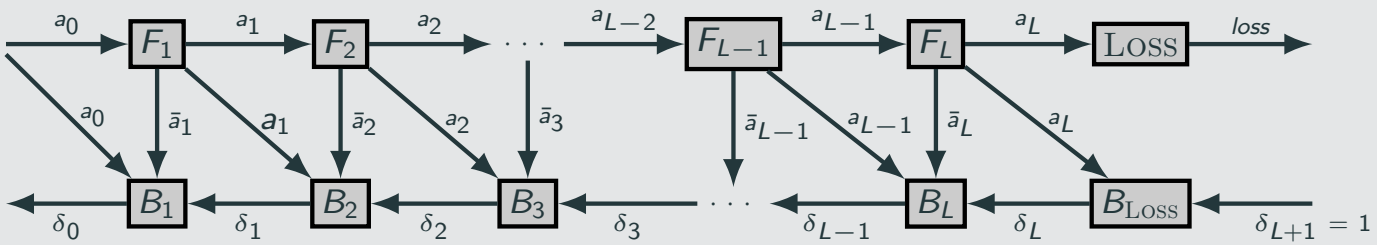


Comparison of different methods based on different bandwidth.

Conclusion and perspectives

Memory Saving Techniques for Training

- Single GPU Strategies
  - rematerialization, activation and weight offloading,...
  - do not change the result
  - combined, typically enable to save half the memory for 10% time overhead
  - `rotor` software provides a **transparent** implementation (for chains)
- Use of Parallel Strategies
  - In the Forward-Backward chain, Work = Critical Path
  - but you can pipeline several images + tasks are parallel
  - data, model, filter, kernel, *-parallelism,...
  - at large scale, affect accuracy



Perspectives

There are other orthogonal strategies to save memory

- DNNs are increasingly complex graphs
- Compression (Tensor Decomposition), use of mixed precision
- Pruning (zeroing small weights), Quantization (using very few bits)
- These techniques potentially change accuracy and convergence...
- There are many important related Combinatorial Optimization and Graph problems to solve
  - Combination of parallelism with offloading, rematerialization is completely open
  - But there are already solutions that combine everything like DeepSpeed https://www.deepspeed.ai (not optimally at all!)

The end

Thank you for your attention!

olivier.beaumont@inria.fr