Optimizing Pipeline Parallelism for Deep Learning with Activation Checkpointing

Min-Yen Chiang[1], Tzu-Hsien Tsai[1], Ding-Yong Hong[2], Pangfeng Liu[1], Jan-Jan Wu[2]

[1]Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
[2]Institute of Information Science, Academia Sinica, Taipei, Taiwan

November 26, 2025

Introduction

- We increase the throughput of DNN training by better pipeline parallelism and activation checkpointing.

The Importance of Deep Neural Networks

- Deep neural networks solve problems across various fields.
  - Object detection
  - Text summarization
  - Protein structure prediction
  - Traffic forecast

DNN Training Procedure

- Let F_i be the forward pass of layer i.
- Let B_i be the backward pass of layer i.
- Let a_i be the activation of layer i.
- Let g_i be the gradient of layer i.

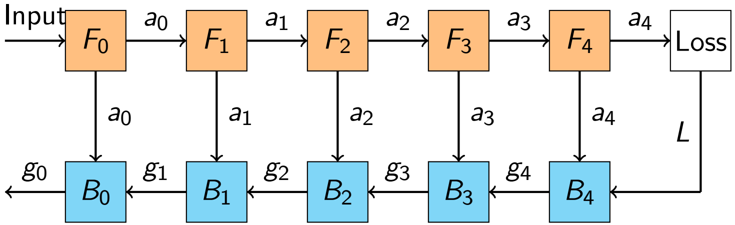

Figure 1: The training procedure of DNNs

Pipeline Parallelism

- The training procedure of a model is partitioned into multiple stages.
  - Each stage consists of consecutive layers in the model.
  - Each stage is trained on different devices.
- A batch of data is divided into multiple micro batches.

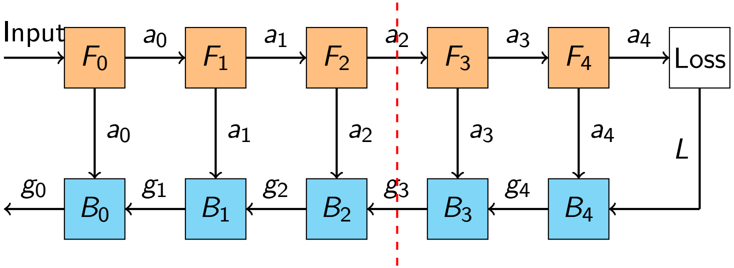

Figure 2: The DNN training procedure using pipeline parallelism

The Memory Usage of Training a Stage

- The activations occupy most of the memory during DNN training[1].

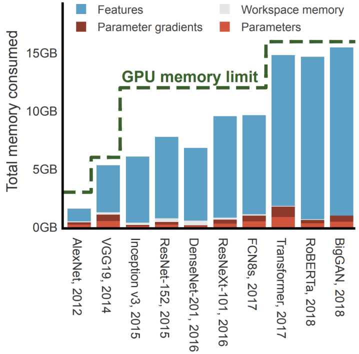

Figure 3: The memory usage of DNN training

[1]Paras Jain et al. "Checkmate: Breaking the memory wall with optimal tensor rematerialization". In: *MLSys*. Vol. 2. 2020, pp. 497–511.

Activation Checkpointing

- Activation Checkpointing stores a subset of activations during the forward pass and recomputes the discarded activations during the backward pass.
  - The stored activations are called *checkpoints*.

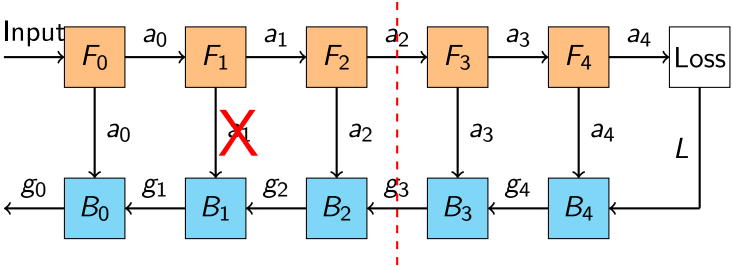

Figure 4: Activation Checkpointing

Fast-forward Models

- A fast-forward model is the DNN that contains a layer whose output is passed to a non-adjacent layer.
  - The fast-forward edges do not cross each other.
  - Many widely-used DNNs are fast-forward models, including ResNet, GPT-3, and Llama.
- We consider the training procedure of fast-forward models.

Problems

- How to schedule the micro batches into the pipeline?
- How to partition a model into stages?
- How to select checkpoints in a stage?

Pipeline Schedule

- We use the one-forward-one-backward (1F1B)[2] pipeline schedule.
- Let p be the number of stages.
- The 1F1B pipeline schedule passes a micro batch backward whenever the input to this backward pass is ready.
- If there is no such a micro batch on the i-th stage and the number of micro batches whose activations are stored on the device does not exceed $p - i + 1$, then a micro batch is passed forward.

[2] Deepak Narayanan et al. "PipeDream: Generalized pipeline parallelism for DNN training". In: *SOSP*. 2019, pp. 1–15.

Previous Work on Training Fast-forward Models

- AdaPipe[3] uses a heuristic to partition a fast-forward model.
- AdaPipe uses a heuristic to find the checkpoints in a fast-forward model.

[3]Zhenbo Sun et al. "AdaPipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning". In: *ASPLOS*. 2024, pp. 86–100.

Contribution

- Our model partition and activation checkpointing algorithms are based on mathematical formulation and dynamic programming.
- Our model partition minimize the maximum stage computation time in the pipeline.
- Our activation checkpointing minimize the stage computation time with memory usage no more than the device memory capacity.
- Our algorithms achieve a higher throughput and a more accurate memory estimation than AdaPipe.

The Goal of Model Partitioning

- Let n be the number of layers in a model.
- Let h_i be the first layer in the i-th stage.
    - We set $h_1 = 1$ and $h_{p+1} = n + 1$.
- Let the training time of the i-th stage be $C_i(h_i, h_{i+1} - 1)$.
- Our goal is to find the sequence $(h_1, h_2, \ldots, h_p, h_{p+1})$ that minimizes S in Equation 1.

$$S = \max_{1 \leq i \leq p} C_i(h_i, h_{i+1} - 1) \tag{1}$$

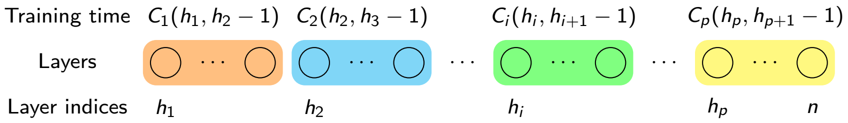

Figure 5: The model partition and stage training time

The Algorithm for Model Partitioning

- Let $D(i, q)$ be the shortest maximum stage training time when layers between layer i and layer n are partitioned into q stages.
- The minimum of S is $D(1, p)$.

$$S = \max_{1 \le i \le p} C_i(h_i, h_{i+1} - 1)$$

- We derive $D(i, q)$ by enumerating the last layer of the stage that starts from i.

$$D(i, q) = \min_{i \le k < n} \max \left(C_{p-q+1}(i, k), D(k + 1, q - 1) \right) \tag{2}$$

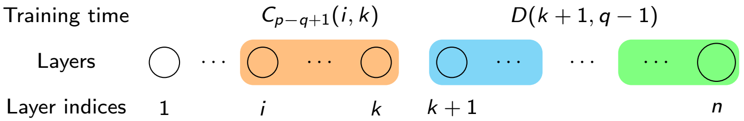

Figure 6: The idea behind Recurrence 2

PyTorch Constraint

- PyTorch does not support the model partition that splits a fast-forward edge.
- Let P be the set of layers that are not crossed by a fast-forward edge.

$$D(i, q) = \min_{\substack{i \le k < n \\ k \in P}} \max\left(C_{p-q+1}(i, k), D(k+1, q-1)\right) \tag{3}$$

Stage Training Time and Activation Checkpointing

- Now we construct the stage training time $C_i(\cdot,\cdot)$.
- The value of $C_i(\cdot,\cdot)$ is related to the checkpoint selection.
- We aim to select checkpoints that minimize $C_i(\cdot,\cdot)$.
- Each activation is recomputed at most once.
    - This constraint is adopted by many existing checkpointing methods[4][5][6].
- We consider deriving the value of $C_i(1,n)$.

[4] Tianqi Chen et al. *Training deep nets with sublinear memory cost*. arXiv preprint arXiv:1604.06174. 2016.

[5] Jianwei Feng and Dong Huang. "Optimal gradient checkpoint search for arbitrary computation graphs". In: *CVPR*. 2021, pp. 11433–11442.

[6] Ding-Yong Hong et al. "GPU memory usage optimization for backward propagation in deep network training". In: *JPDC* 199 (2025), p. 105053.

The Goal of Activation Checkpointing

- Let f_j be the time for passing forward a micro batch through layer j.
- Let b_j be the time for passing backward a micro batch through layer j.
- Let A be the set of checkpoints.
- The stage training time is the sum of the forward passing time, the backward passing time, and the time for recomputing the discarded activation, as shown in Equation 4.

$$C_i(1,n) = \sum_{1 \le j \le n} f_j + \sum_{1 \le j \le n} b_j + (\sum_{1 \le j \le n} f_j - \sum_{j \in A} f_j) \tag{4}$$

- Minimizing $C_i(1,n)$ is equal to maximizing $C'_i(1,n)$ in Equation 5.

$$C'_i(1,n) = \sum_{j \in A} f_{c_j} \tag{5}$$

The Constraint of Activation Checkpointing

- The term $C'_i(1, n)$ is maximized when A contains every layer in the model.
$$C'_i(1, n) = \sum_{j \in A} f_{c_j}$$
- The memory usage for storing every activations may exceed the device memory capacity.
- Therefore, our goal of activation checkpointing becomes maximizing the checkpoint computation time $C'_i(1, n)$ with the memory usage no more than the device memory capacity.

The Memory Usage for Backward Passing

- The memory usage for training a stage consists of two parts.
  - The checkpoints
  - The gradients and the recomputed activations
- Let l be the maximum number of micro batches whose activations are stored on the device.
- Let d_i be the size of the activation of layer i.
- The memory usage of selecting layer i as a checkpoint is $l \times d_i$.
- Let $s(i, j)$ be the memory usage for storing the gradients and the recomputed activations when passing a micro batch backward from layer j to layer i.

Maximum Checkpoint Computation Time with Memory Constraint

- Let $T(i, m)$ be the maximum checkpoint computation time of layers between layer i and layer n with the memory usage no more than m and layer i as a checkpoint.
- The maximum checkpoint computation time is $T(1, M)$, where M is the device memory capacity.
- We derive $T(i, m)$ by enumerating the first checkpoint after layer i.

$$T(i, m) = f_i + \max_{\substack{i<j\le n \\ s(i,j)\le m-l\times d_i}} T(j, m - l \times d_i) \tag{6}$$

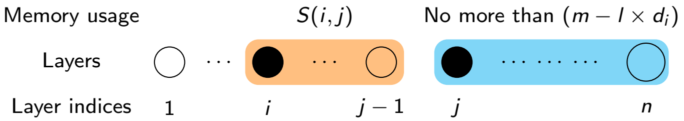

Figure 7: The idea behind Recurrence 7

PyTorch Constraint for Activation Checkpointing

- PyTorch selects the starting point of a fast-forward edge e_f as a checkpoint under the following two conditions.
  - Neither the starting point nor the endpoint of e_f is a checkpoint.
  - The edge e_f crosses a checkpoint.
- PyTorch applies this mechanism to avoid storing too many activations in the memory
- To ensure that our memory estimation is consistent with the memory usage of PyTorch, our algorithm adopts the PyTorch constraint.

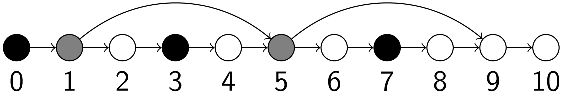

Figure 8: Checkpoints added by PyTorch

Maximum Checkpoint Computation Time with Memory and PyTorch Constraints

- Let $u(i)$ be the first layer after layer i that is the starting point or the endpoint of a fast-forward edge.
- When $j > u(i)$ and layer j is crossed by a fast-forward edge e_f, our algorithm selects $u(j)$ as the checkpoint.

$$T(i, m) = f_i + \max_{\substack{i<j\leq n \\ s(i,j)\leq m-l\times d_i}} T(j, m - l \times d_i) \tag{7}$$

- The fast-forward edges do not cross each other.

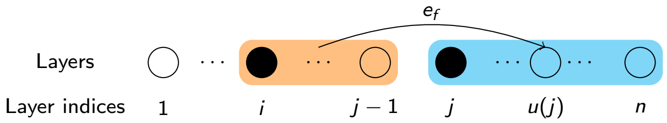

Figure 9: The situation that may violate the PyTorch constraint

Maximum Checkpoint Computation Time with Memory and PyTorch Constraints

- The function $T^*(i, m)$ is $T(i, m)$ with an additional constraint that $u(i)$ is the checkpoint.
- Let U be the set consisting of all layers that may violate the PyTorch constraint.

$$T(i, m) = f_i + \max_{\substack{i<j\leq n \\ s(i,j)\leq m-l\times d_i}} \begin{cases} T(j, m - l \times d_i) & j \notin U \\ T^*(j, m - l \times d_i) & j \in U \end{cases} \quad (8)$$

$$T^*(i, m) = f_i + \max_{\substack{i<j\leq u(i) \\ s(i,j)\leq m-l\times d_i}} \begin{cases} T^*(j, m - l \times d_i) & j < u(i) \\ T(j, m - l \times d_i) & j = u(i) \end{cases} \quad (9)$$

Environment

- We conduct our experiments on the server with 32 2.1 GHz Intel(R) Xeon(R) Gold 6530 processors and four Nvidia A6000 GPUs.
- We use PyTorch to implement our experiments.

Memory Prediction

- The average difference of our memory estimation and the memory usage reported by PyTorch is 0.9%.

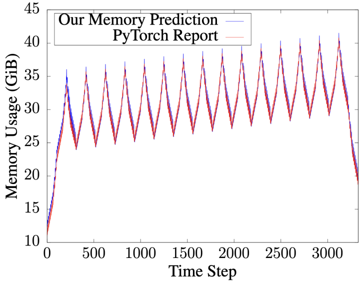

Figure 10: Our memory prediction and the memory usage of PyTorch implementation

Throughput

- Our algorithm outperforms the state-of-the-art algorithm AdaPipe[7] on models with various sizes and structures.

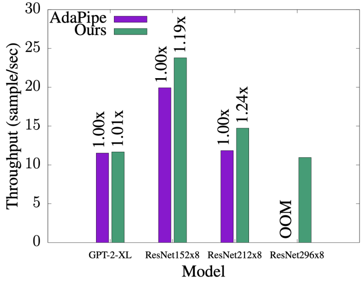

Figure 11: The training throughput of our algorithm and that of AdaPipe

[7] Sun et al., "AdaPipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning".

Conclusion

- We increase the training throughput of fast-forward models by better pipeline parallelism and activation checkpointing.
- Our model partition minimizes the maximum stage training time.
- Our checkpoint selection minimizes the stage training time under the memory capacity.
- Our methods follow the constraint of PyTorch and thus estimate the memory usage accurately.

Q & A

Q & A