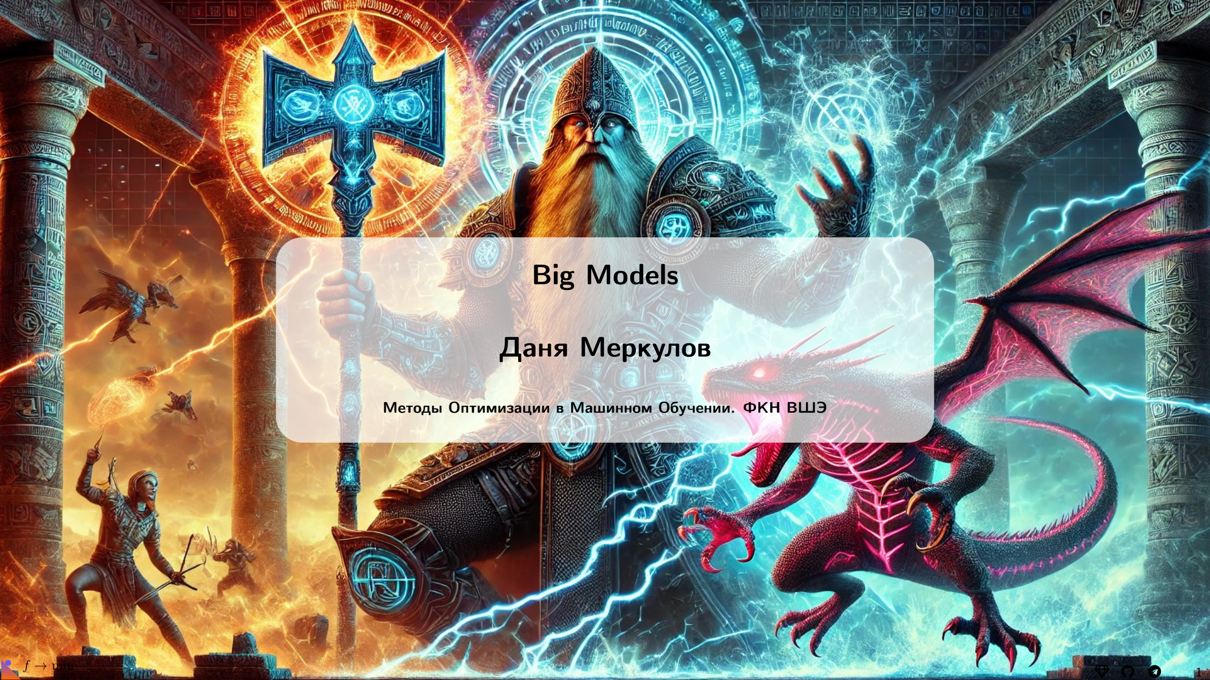


Big Models

Даня Меркулов

Методы Оптимизации в Машинном Обучении. ФКН ВШЭ

Тренды

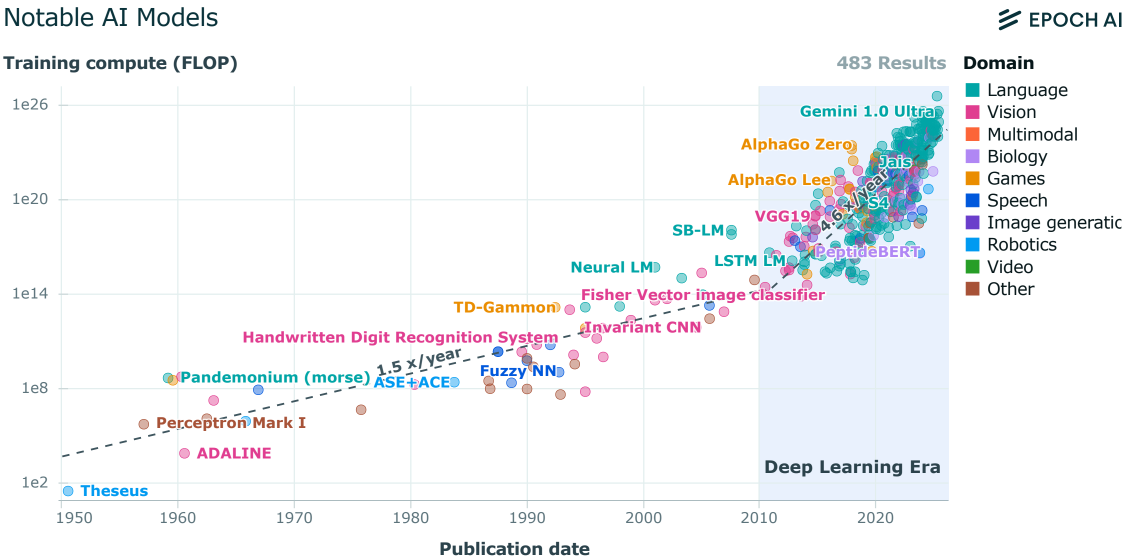


Figure 1: Динамика вычислений, необходимых для обучения моделей. Источник

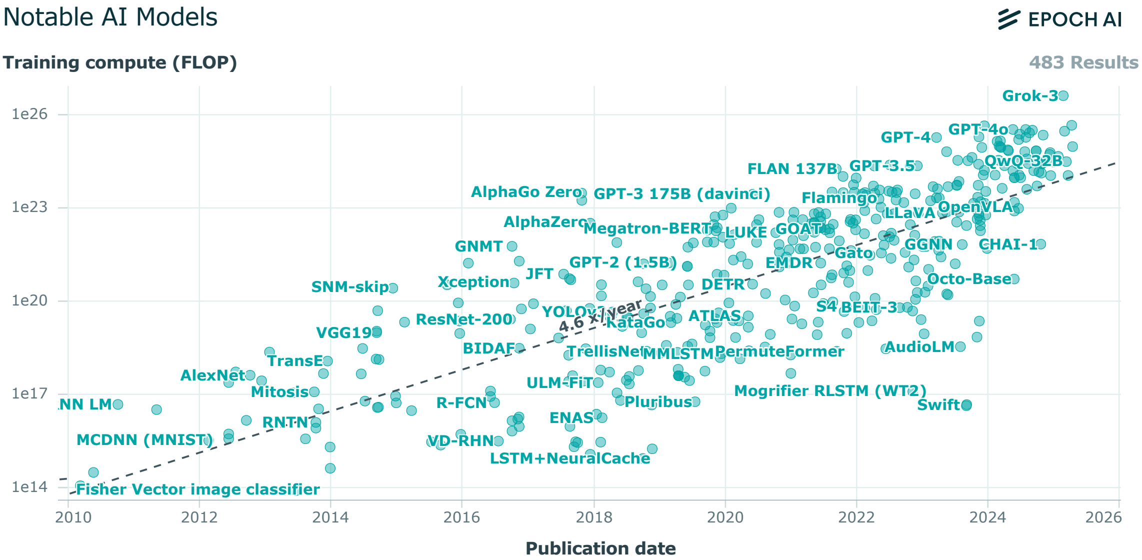


Figure 2: Динамика вычислений, необходимых для обучения нейросетевых моделей. Источник

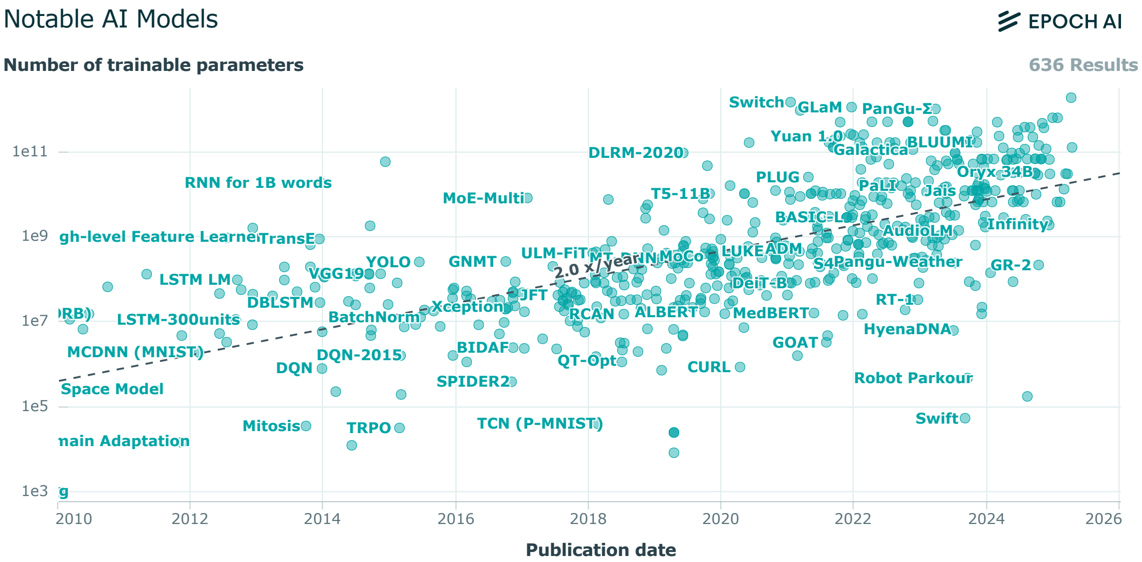
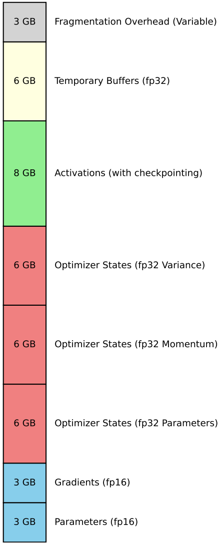


Figure 3: Динамика количества обучаемых параметров нейросетевых моделей. Источник

GPT-2 training Memory footprint

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

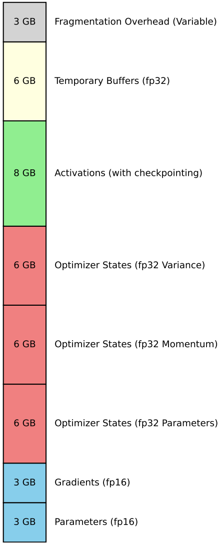
Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.

Memory Requirements Example:

Residual Memory Consumption:

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

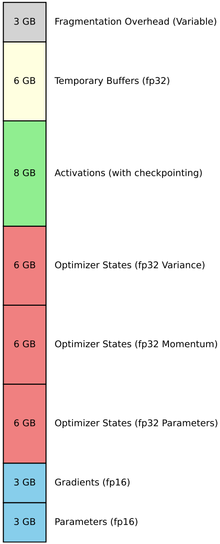
Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

Memory Requirements Example:

Residual Memory Consumption:

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Model States:

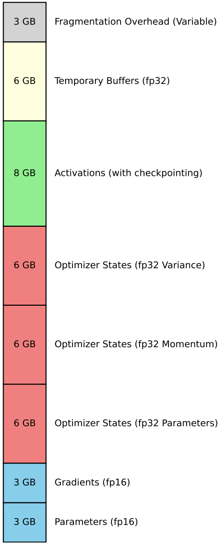
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).

Residual Memory Consumption:

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Model States:

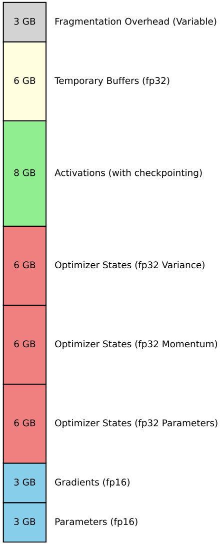
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).
- Total: 16Ψ bytes; for GPT-2 with 1.5B parameters, this equals 24GB.

Residual Memory Consumption:

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

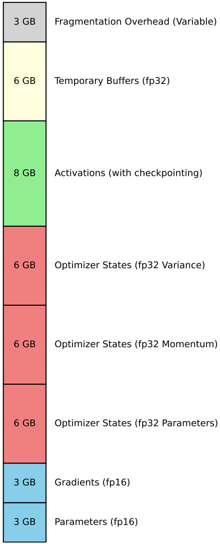
Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).
- Total: 16Ψ bytes; for GPT-2 with 1.5B parameters, this equals 24GB.

Residual Memory Consumption:

- Activations: Significant memory usage, e.g., 1.5B parameter GPT-2 model with sequence length 1K and batch size 32 requires ~60GB.

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

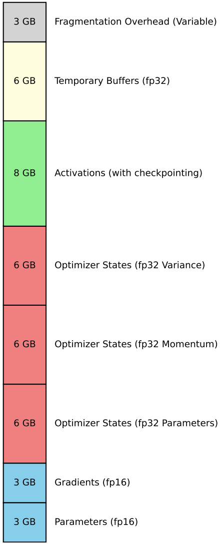
Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).
- Total: 16Ψ bytes; for GPT-2 with 1.5B parameters, this equals 24GB.

Residual Memory Consumption:

- Activations: Significant memory usage, e.g., 1.5B parameter GPT-2 model with sequence length 1K and batch size 32 requires ~60GB.
- Activation checkpointing can reduce activation memory by about 50%, with a 33% recomputation overhead.

GPT-2 training Memory footprint



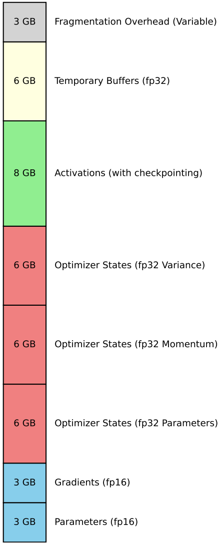
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.

Memory Fragmentation:

GPT-2 training Memory footprint



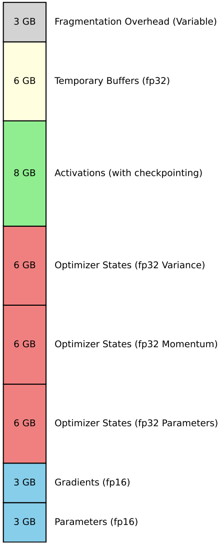
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

Memory Fragmentation:

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

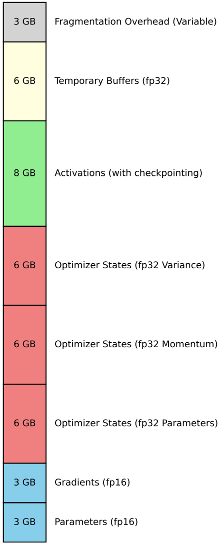
Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

Memory Fragmentation:

- Memory fragmentation can cause out-of-memory issues despite available memory, as contiguous blocks are required.

GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

Memory Fragmentation:

- Memory fragmentation can cause out-of-memory issues despite available memory, as contiguous blocks are required.
- In some cases, over 30% of memory remains unusable due to fragmentation.

Scaling Laws

Scaling Laws [1]

- **Эмпирическое правило**: кросс-энтропия уменьшается по степенному закону

$$L(N, D, C) \propto N^{-\alpha} D^{-\beta} C^{-\gamma}$$

где N — параметры, D — токены, C — FLOPs.

Scaling Laws [1]

- **Эмпирическое правило**: кросс-энтропия уменьшается по степенному закону

$$L(N, D, C) \propto N^{-\alpha} D^{-\beta} C^{-\gamma}$$

  где N — параметры, D — токены, C — FLOPs.
- **Compute allocation**: при фиксированном C оптимально $N \propto D^{0.74}$ – крупнее модель, меньше данных.

Scaling Laws [1]

- **Эмпирическое правило**: кросс-энтропия уменьшается по степенному закону

$$L(N, D, C) \propto N^{-\alpha} D^{-\beta} C^{-\gamma}$$

  где N — параметры, D — токены, C — FLOPs.
- **Compute allocation**: при фиксированном C оптимально $N \propto D^{0.74}$ – крупнее модель, меньше данных.
- **Предсказание качества**: линейность на $\log$—$\log$-графике сохраняется вплоть до GPT-3-scale.

Scaling Laws [1]

- **Эмпирическое правило**: кросс-энтропия уменьшается по степенному закону

$$L(N, D, C) \propto N^{-\alpha} D^{-\beta} C^{-\gamma}$$

  где N — параметры, D — токены, C — FLOPs.
- **Compute allocation**: при фиксированном C оптимально $N \propto D^{0.74}$ – крупнее модель, меньше данных.
- **Предсказание качества**: линейность на $\log$—$\log$-графике сохраняется вплоть до GPT-3-scale.
- Практически scaling-законы помогают подбирать размеры корпуса и останавливать обучение до переобучения.

[1] Kaplan et al., 2020

Chinchilla [2]

- DeepMind обучили **Chinchilla 70 B** на **1.4 T** токенов при том же compute, что и Gopher 280 B.

Chinchilla [2]

- DeepMind обучили **Chinchilla 70 B** на **1.4 T** токенов при том же compute, что и Gopher 280 B.
- **Результат**: +7 pp на MMLU и существенный прирост на BIG-bench vs GPT-3.

Chinchilla [2]

- DeepMind обучили **Chinchilla 70 B** на **1.4 T** токенов при том же compute, что и Gopher 280 B.
- **Результат**: +7 pp на MMLU и существенный прирост на BIG-bench vs GPT-3.
- **Compute-optimal scaling**: при ограниченных FLOPs соотношение «токенов-на-параметр»

$$\frac{D}{N} \approx 20$$

обеспечивает максимум качества.

Chinchilla [2]

- DeepMind обучили **Chinchilla 70 B** на **1.4 T** токенов при том же compute, что и Gopher 280 B.
- **Результат**: +7 pp на MMLU и существенный прирост на BIG-bench vs GPT-3.
- **Compute-optimal scaling**: при ограниченных FLOPs соотношение «токенов-на-параметр»

$$\frac{D}{N} \approx 20$$

  обеспечивает максимум качества.
- Вывод: лучше «дольше учить меньшую модель», чем «коротко учить огромную».

[2] Hoffmann et al., 2022

Chinchilla scaling laws

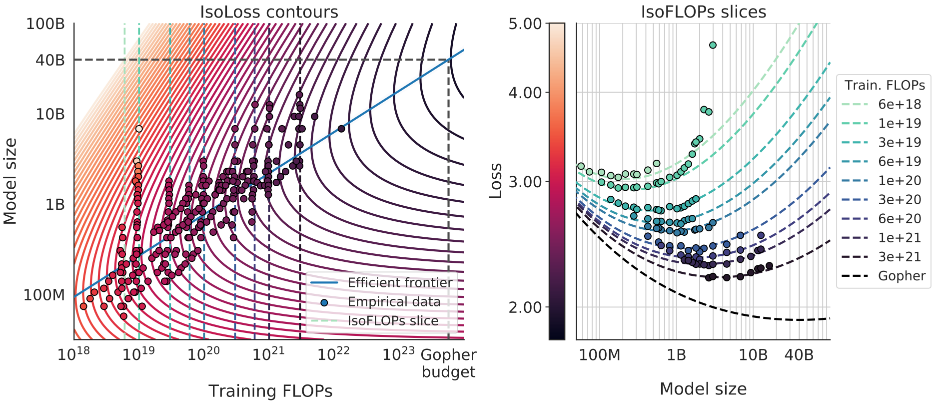


Figure 4: Parametric modeling of the loss L(N, D) with contour plot (left) and isoFLOP slices (right). Each isoFLOP slice corresponds to a dashed line in the left plot. The efficient frontier is shown in blue, forming a line in log-log space. The curve intersects each iso-loss contour at the point of minimum FLOPs. The optimal model size for the Gopher FLOP budget is projected to be 40B parameters.

Automatic Mixed Precision (AMP)

Activations [3]

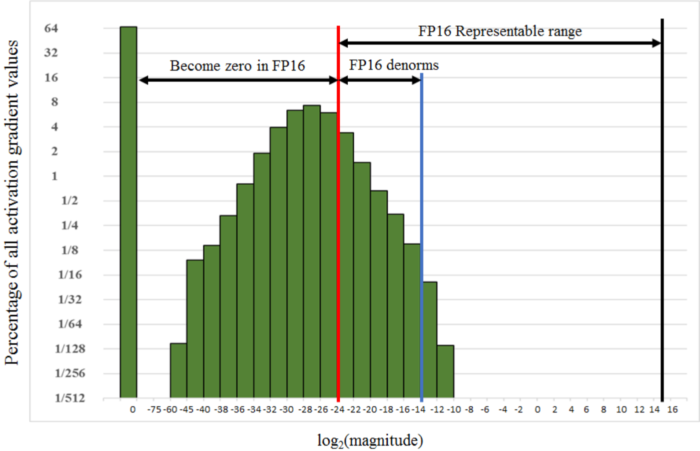


Figure 5: Histogram of activation gradient values during the training of Multibox SSD network. Note that the bins on the x-axis cover varying ranges and there's a separate bin for zeros. For example, 2% of the values are in the $[2^{-34}, 2^{-32})$ range, 2% of values are in the $[2^{-24}, 2^{-23})$ range, and 67% of values are zero.

[3] Mixed Precision Training

Weights [4]

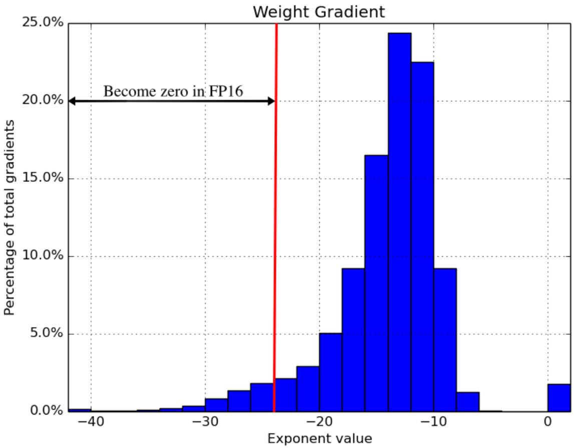
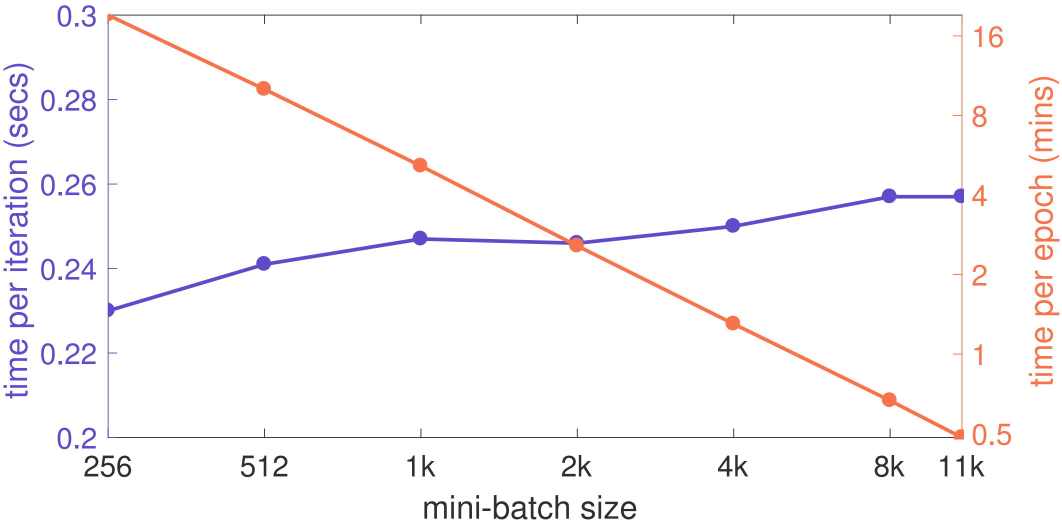


Figure 6: Histogram for the exponents of weight gradients for DeepSpeech 2 model (215 M parameters) training on Mandarin speech recognition. The gradients are sampled every 4,000 iterations during training for all the layers in the model.

[4] Mixed Precision Training

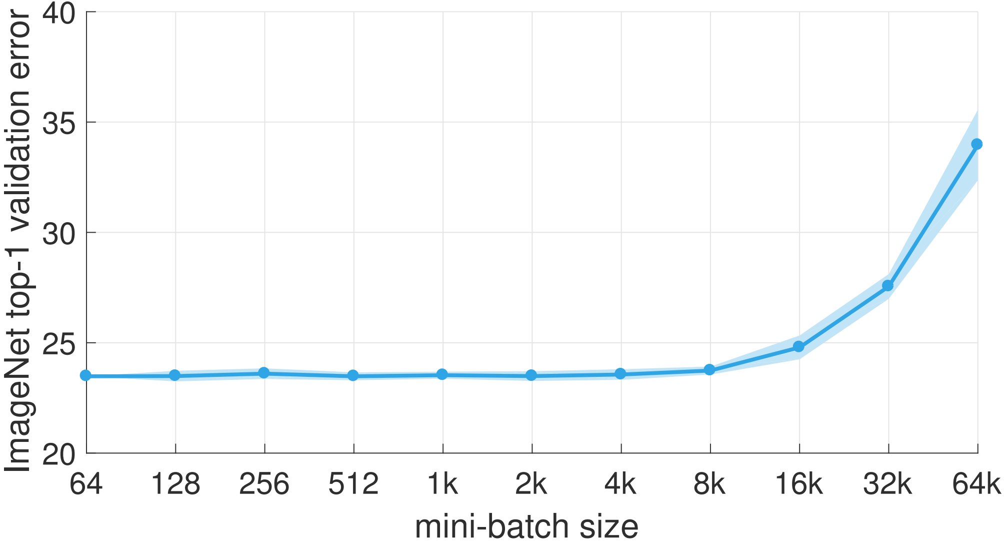
Large batch training

Large batch training [5]



[5] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Large batch training [6]



[6] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Large batch training [7]

Effective batch size (kn)	α	top-1 error (%)
256	0.05	23.92 $\pm$ 0.10
256	0.10	23.60 $\pm$ 0.12
256	0.20	23.68 $\pm$ 0.09
8k	$0.05 \cdot 32$	24.27 $\pm$ 0.08
8k	$0.10 \cdot 32$	23.74 $\pm$ 0.09
8k	$0.20 \cdot 32$	24.05 $\pm$ 0.18
8k	0.10	41.67 $\pm$ 0.10
8k	$0.10 \cdot \sqrt{32}$	26.22 $\pm$ 0.03

Comparison of learning rate scaling rules. ResNet-50 trained on ImageNet. A reference learning rate of $\alpha = 0.1$ works best for $kn = 256$ (23.68% error). The linear scaling rule suggests $\alpha = 0.1 \cdot 32$ when $kn = 8$k, which again gives best performance (23.74% error). Other ways of scaling α give worse results.

[7] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Linear and square root scaling rules

When training with large batches, the learning rate must be adjusted to maintain convergence speed and stability. The **linear scaling rule**[8] suggests multiplying the learning rate by the same factor as the increase in batch size:

$$\alpha_{\text{new}} = \alpha_{\text{base}} \cdot \frac{\text{Batch Size}_{\text{new}}}{\text{Batch Size}_{\text{base}}}$$

The **square root scaling rule**[9] proposes scaling the learning rate with the square root of the batch size increase:

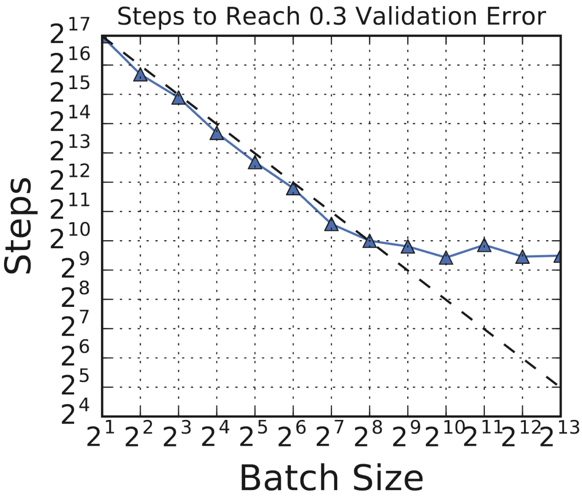
$$\alpha_{\text{new}} = \alpha_{\text{base}} \cdot \sqrt{\frac{\text{Batch Size}_{\text{new}}}{\text{Batch Size}_{\text{base}}}}$$

Authors claimed, that it suits for adaptive optimizers like Adam, RMSProp and etc. while linear scaling rule serves well for SGD.

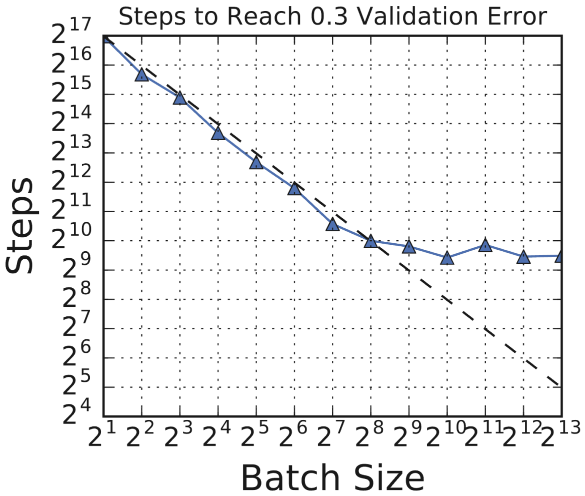
[8] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

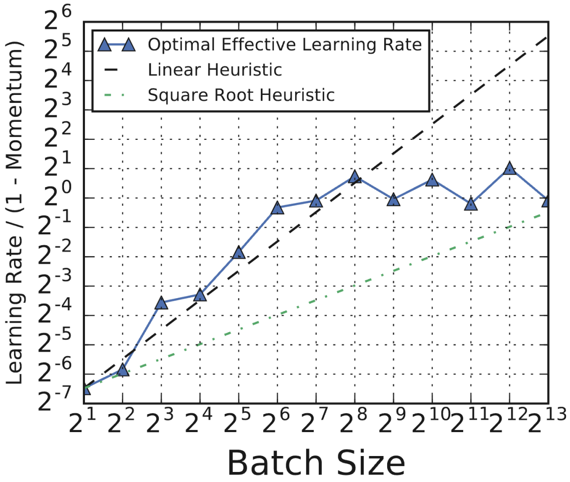
[9] Learning Rates as a Function of Batch Size: A Random Matrix Theory Approach to Neural Network Training

Batch size scaling



Batch size scaling





Gradual warmup [10]

Gradual warmup helps to avoid instability when starting with large learning rates by slowly increasing the learning rate from a small value to the target value over a few epochs. This is defined as:

$$\alpha_t = \alpha_{\text{max}} \cdot \frac{t}{T_w}$$

where t is the current iteration and T_w is the warmup duration in iterations. In the original paper, authors used first 5 epochs for gradual warmup.

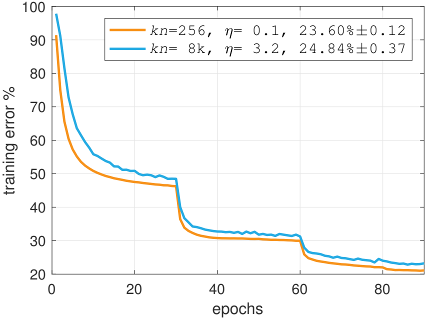


Figure 7: no warmup

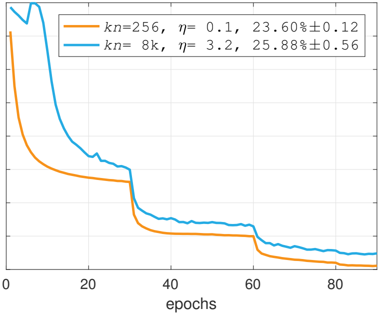


Figure 8: constant warmup

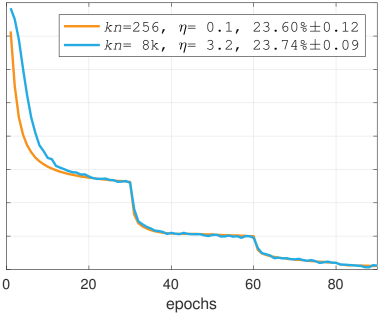
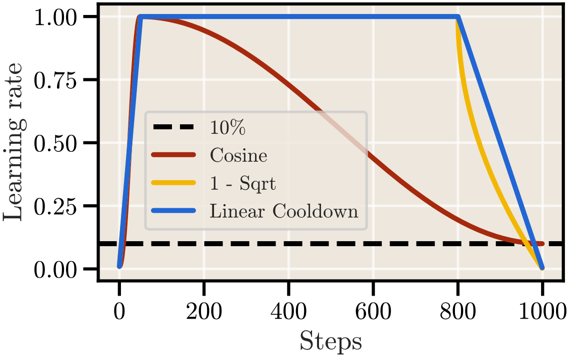
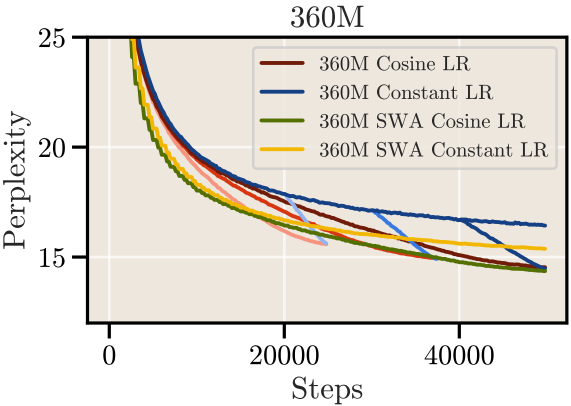


Figure 9: gradual warmup

[10] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Cooldown[11] [12]





[11] Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations
[12] Scaling Vision Transformers

Gradient accumulation

Gradient accumulation allows the effective batch size to be increased without requiring larger memory by accumulating gradients over several mini-batches:

Without gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

Gradient accumulation

Gradient accumulation allows the effective batch size to be increased without requiring larger memory by accumulating gradients over several mini-batches:

Without gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

With gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    if (i+1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

MultiGPU training

Data Parallel training

1. Parameter server sends the full copy of the model to each device

Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes

Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients

Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients
4. Parameter server updates the model

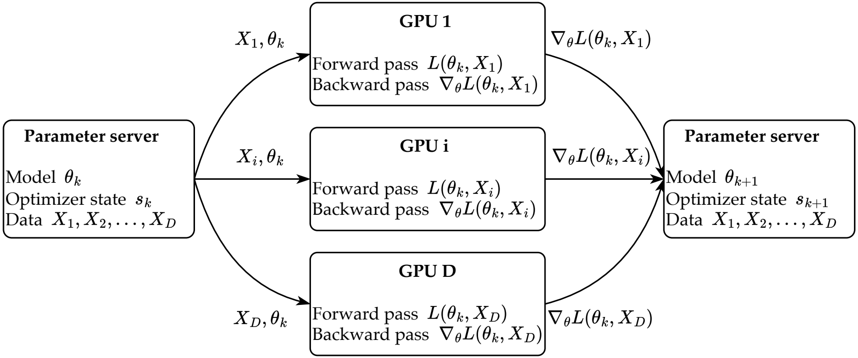
Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients
4. Parameter server updates the model

Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients
4. Parameter server updates the model

Per device batch size: b. Overall batchsize: Db. Data parallelism involves splitting the data across multiple GPUs, each with a copy of the model. Gradients are averaged and weights updated synchronously:



Distributed Data Parallel training

Distributed Data Parallel (DDP) [13] extends data parallelism across multiple nodes. Each node computes gradients locally, then synchronizes with others. Below one can find differences from the PyTorch site. This is used by default in Accelerate library.

DataParallel	DistributedDataParallel
More overhead; model is replicated and destroyed at each forward pass	Model is replicated only once
Only supports single-node parallelism	Supports scaling to multiple machines
Slower; uses multithreading on a single process and runs into Global Interpreter Lock (GIL) contention	Faster (no GIL contention) because it uses multiprocessing

[13] Getting Started with Distributed Data Parallel

Naive model parallelism

Model parallelism divides the model across multiple GPUs. Each GPU handles a subset of the model layers, reducing memory load per GPU. Allows to work with the models, that won't fit in the single GPU Poor resource utilization.

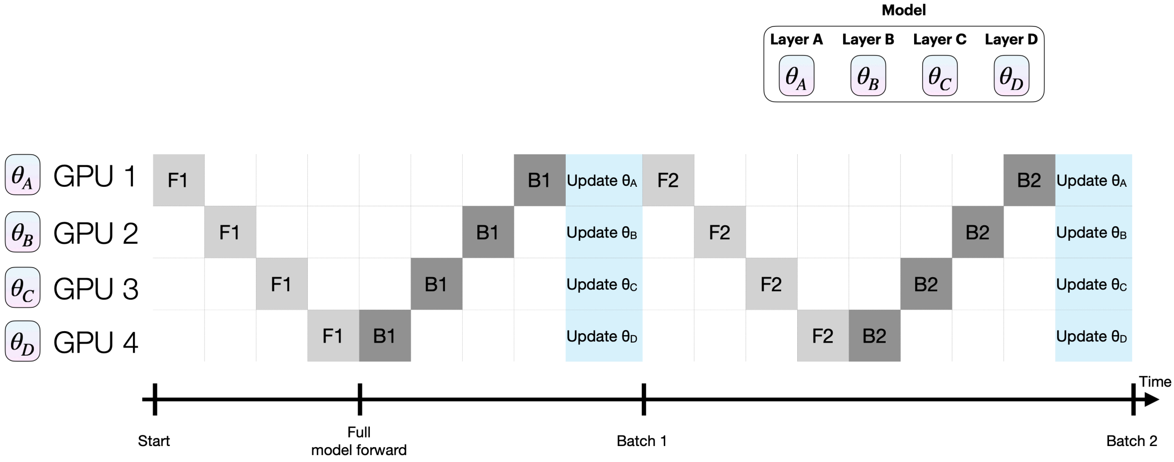
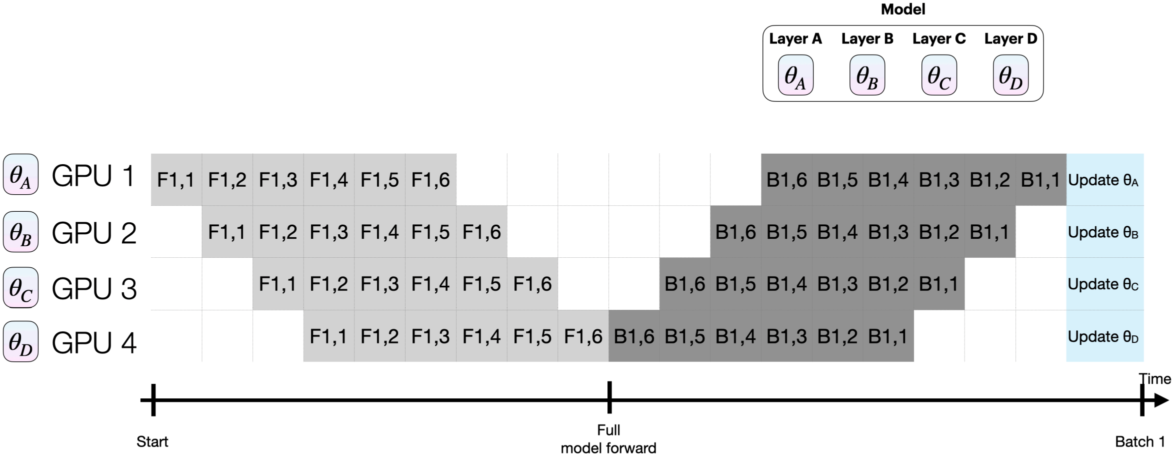


Figure 11: Model parallelism

Pipeline model parallelism (GPipe) [14]

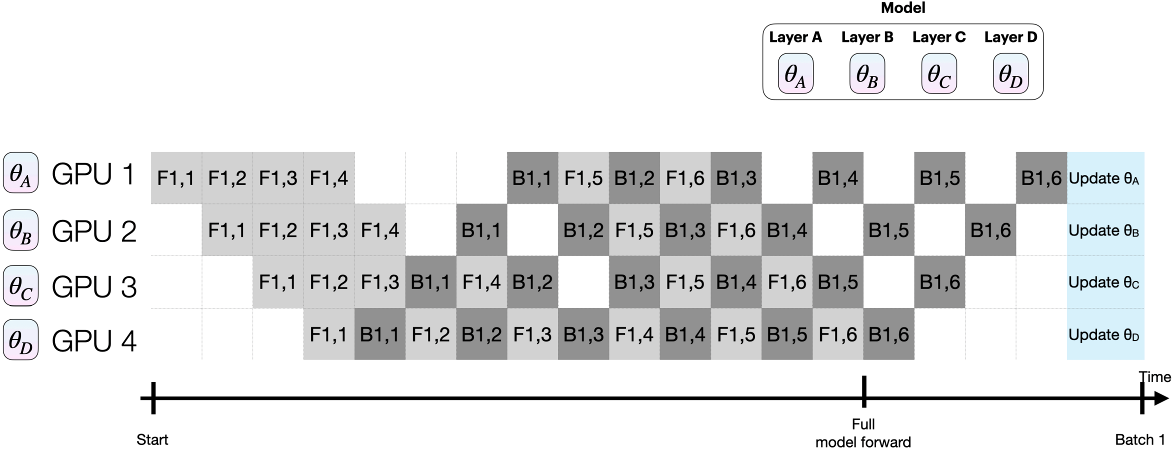
GPipe splits the model into stages, each processed sequentially. Micro-batches are passed through the pipeline, allowing for overlapping computation and communication:



[14] GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

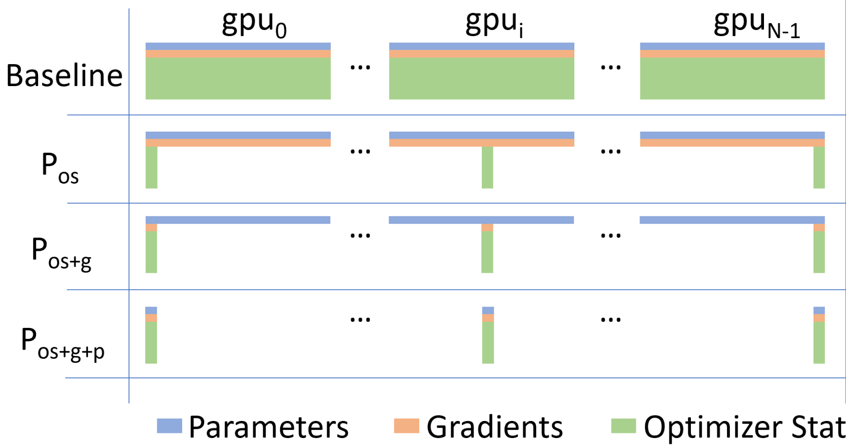
Pipeline model parallelism (PipeDream) [15]

PipeDream uses asynchronous pipeline parallelism, balancing forward and backward passes across the pipeline stages to maximize utilization and reduce idle time:



[15] PipeDream: Generalized Pipeline Parallelism for DNN Training

ZeRO [16]



	Memory Consumed	K=12, $\mathbf{\Psi}$=7.5B, N_d=64
Baseline	$(2+2+K)*\mathbf{\Psi}$	120GB
P_{os}	$2\mathbf{\Psi}+2\mathbf{\Psi}+\frac{K*\mathbf{\Psi}}{N_d}$	31.4GB
P_{os+g}	$2\mathbf{\Psi}+\frac{(2+K)*\mathbf{\Psi}}{N_d}$	16.6GB
P_{os+g+p}	$\frac{(2+2+K)*\mathbf{\Psi}}{N_d}$	1.9GB

[16] ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

FSDP (Fully Sharded Data Parallel) [17]

- Шардинг параметров, градиентов и состояний оптимизатора по процессам $\to$ экономия ⊠**7× памяти** относительно DDP.

FSDP (Fully Sharded Data Parallel) [17]

- Шардинг параметров, градиентов и состояний оптимизатора по процессам $\rightarrow$ экономия ⊠**7**$\times$ **памяти** относительно DDP.
- Обмены выполняются только на границах sync; остальное время модель видит полный тензор.

FSDP (Fully Sharded Data Parallel) [17]

- Шардинг параметров, градиентов и состояний оптимизатора по процессам $\to$ экономия ⊠**7$\times$ памяти** относительно DDP.
- Обмены выполняются только на границах sync; остальное время модель видит полный тензор.
- Поддержка CPU-offload, mixed-precision, активационного checkpointinga.

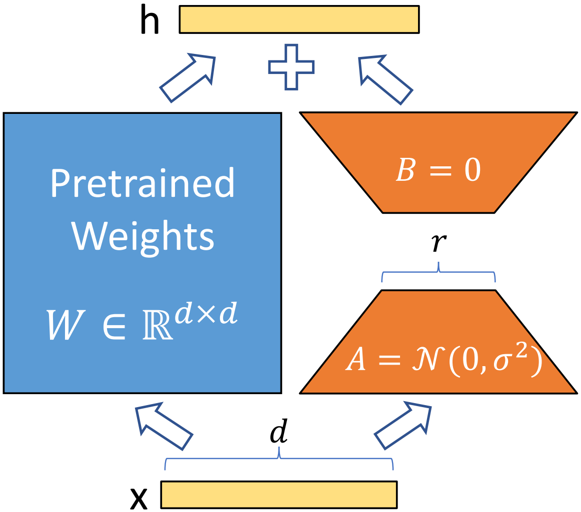
FSDP (Fully Sharded Data Parallel) [17]

- Шардинг параметров, градиентов и состояний оптимизатора по процессам → экономия ⌧**7× памяти** относительно DDP.
- Обмены выполняются только на границах sync; остальное время модель видит полный тензор.
- Поддержка CPU-offload, mixed-precision, активационного checkpointinga.
- Минимальный пример:

```python
import torch
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
torch.cuda.set_device(device_id)
sharded_module = FSDP(my_module)
optim = torch.optim.SGD(sharded_module.parameters(), lr=0.0001)
x = sharded_module(x, y=3, z=torch.Tensor([1]))
loss = x.sum()
loss.backward()
optim.step()
```

[17] PyTorch docs

LoRA [18]



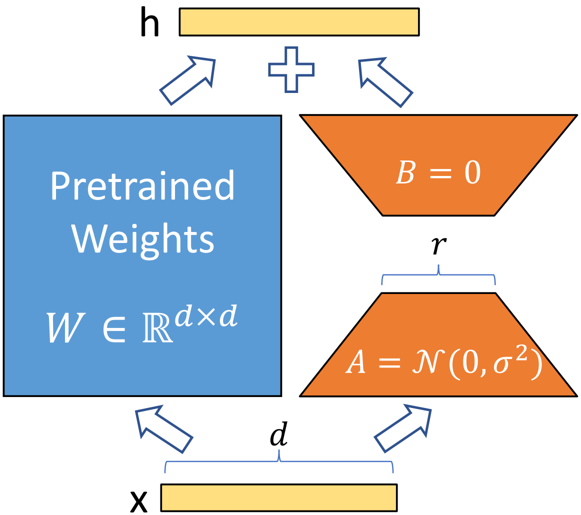
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where $\Delta W = AB^T$, with A and B being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- A is initialized as usual, while B is initialized with zeroes in order to start from identity mapping

LoRA [18]



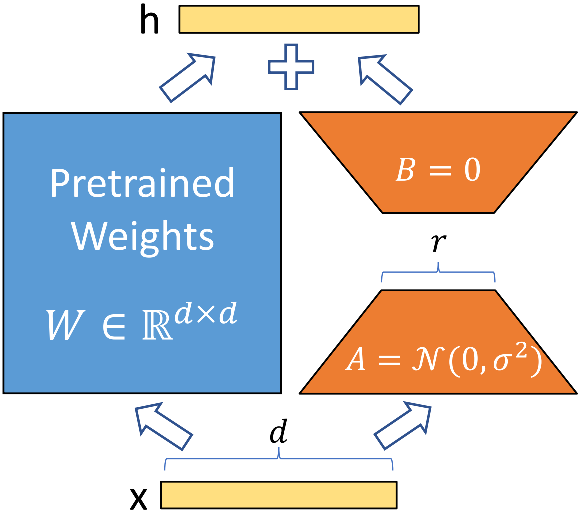
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where $\Delta W = AB^T$, with A and B being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- A is initialized as usual, while B is initialized with zeroes in order to start from identity mapping
- r is typically selected between 2 and 64

LoRA [18]



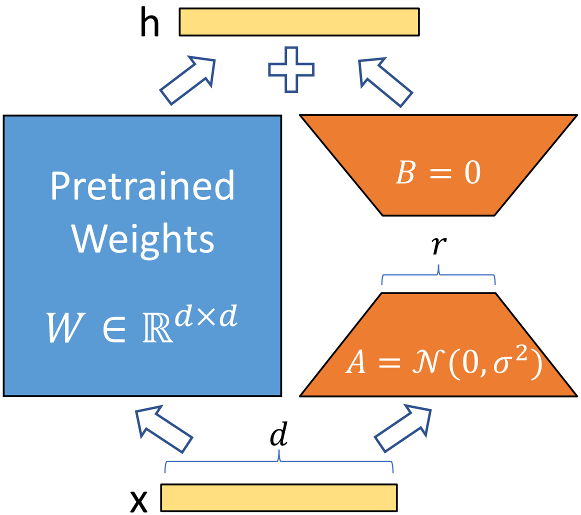
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where $\Delta W = AB^T$, with A and B being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- A is initialized as usual, while B is initialized with zeroes in order to start from identity mapping
- r is typically selected between 2 and 64
- Usually applied to attention modules

LoRA [18]



LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

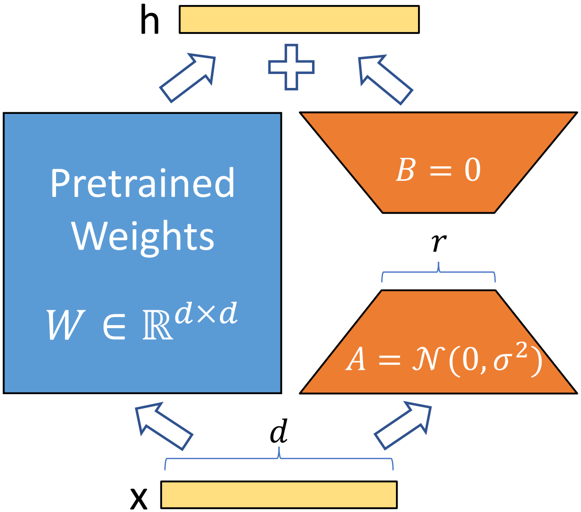
$$W_{\text{new}} = W + \Delta W$$

where $\Delta W = AB^T$, with A and B being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- A is initialized as usual, while B is initialized with zeroes in order to start from identity mapping
- r is typically selected between 2 and 64
- Usually applied to attention modules

[18] LoRA: Low-Rank Adaptation of Large Language Models

LoRA [18]



LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where $\Delta W = AB^T$, with A and B being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- A is initialized as usual, while B is initialized with zeroes in order to start from identity mapping
- r is typically selected between 2 and 64
- Usually applied to attention modules

$$h = W_{\text{new}}x = Wx + \Delta Wx = Wx + AB^Tx$$

[18] LoRA: Low-Rank Adaptation of Large Language Models

Feedforward Architecture

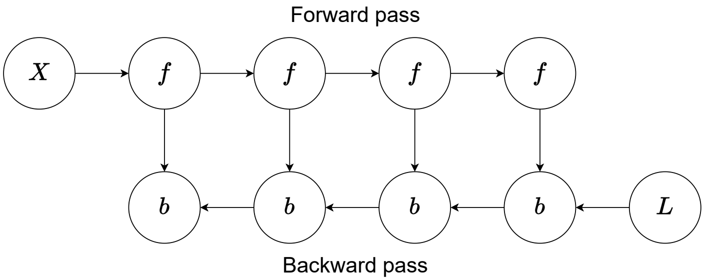


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

Feedforward Architecture

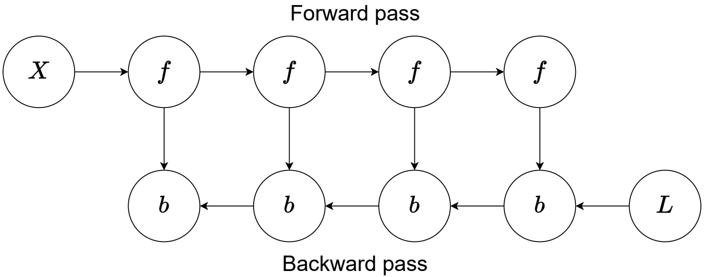


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

! Important

The results obtained for the f nodes are needed to compute the b nodes.

Vanilla backpropagation

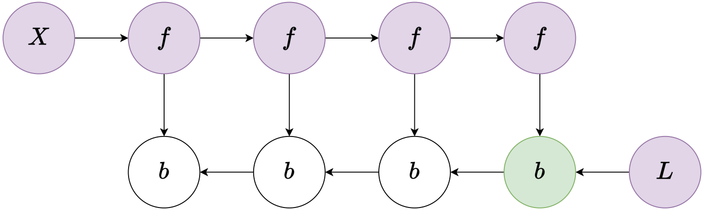


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Vanilla backpropagation

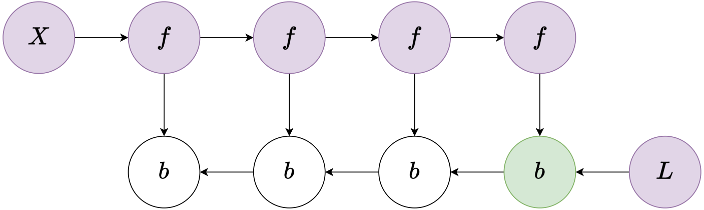


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

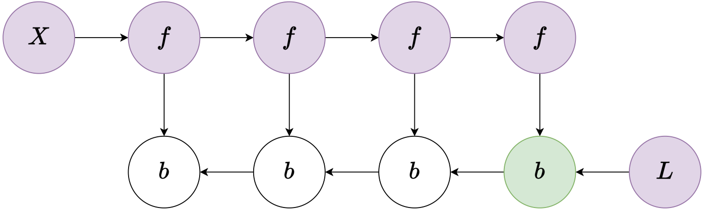


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

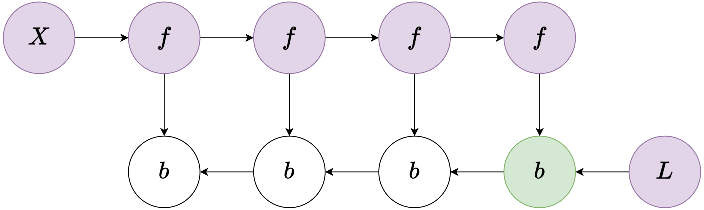


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

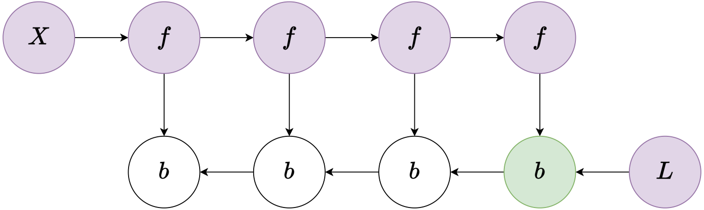


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

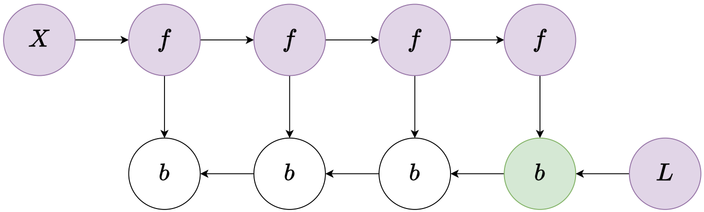


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

- High memory usage. The memory usage grows linearly with the number of layers in the neural network.

Memory poor backpropagation

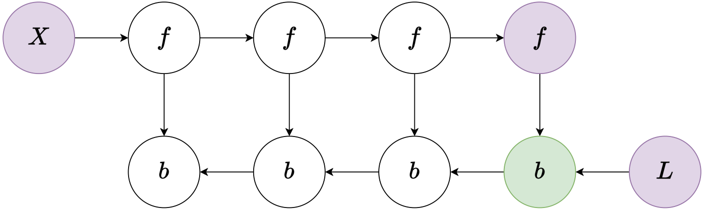


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Memory poor backpropagation

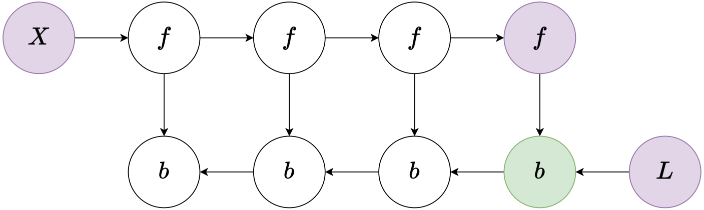


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

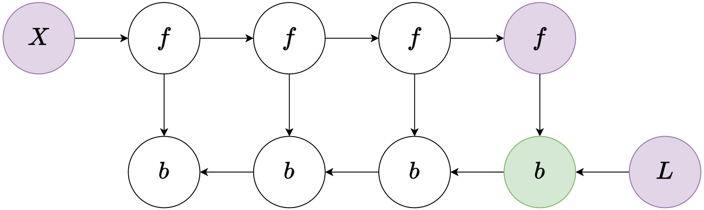


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

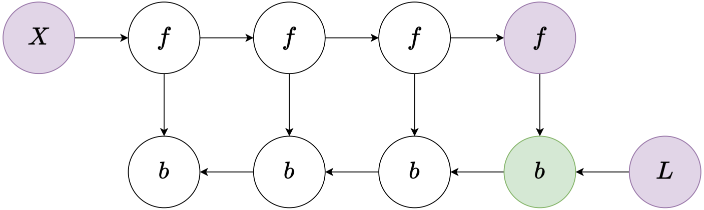


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

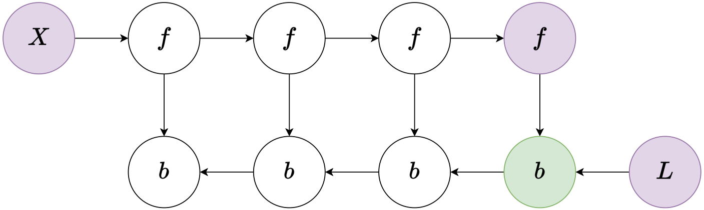


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

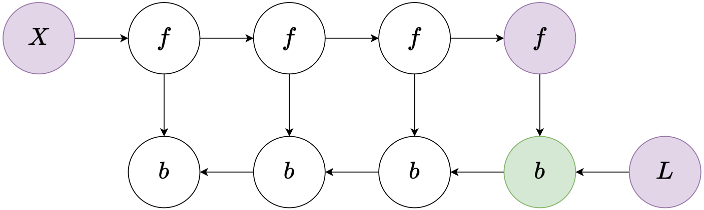


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

- Computationally inefficient. The number of node evaluations scales with n^2, whereas it vanilla backprop scaled as n: each of the n nodes is recomputed on the order of n times.

Checkpointed backpropagation

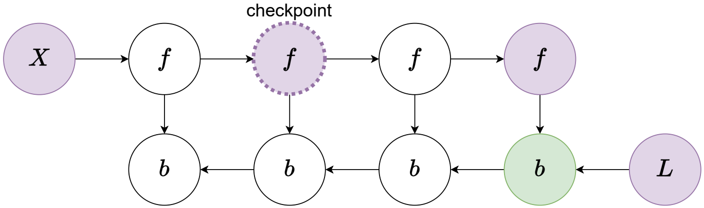


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Checkpointed backpropagation

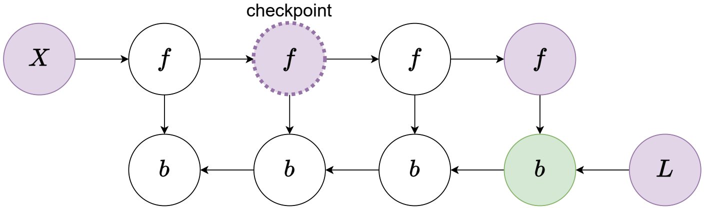


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

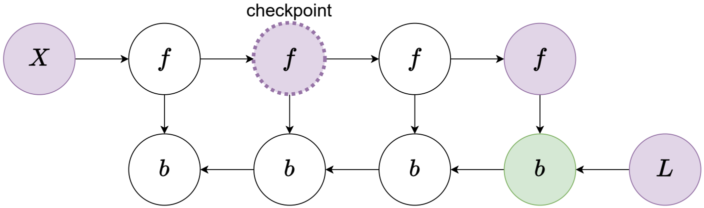


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

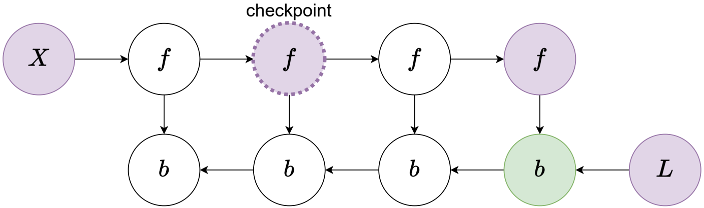


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

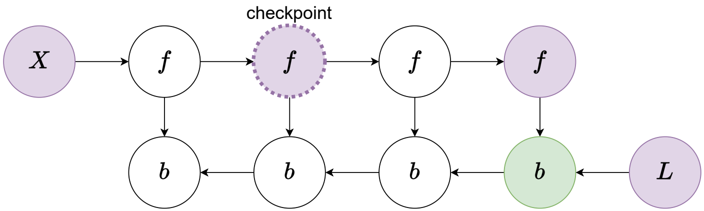


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

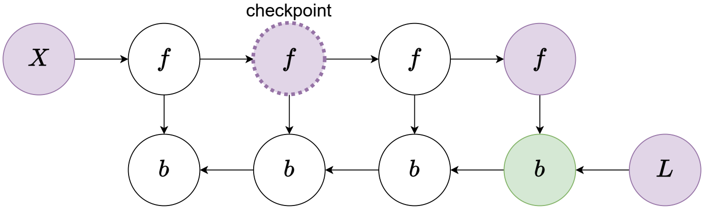


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

- Memory consumption depends on the number of checkpoints. More effective then **vanilla** approach.

Gradient checkpointing visualization

The animated visualization of the above approaches

An example of using a gradient checkpointing

Quantization

Split the weight matrix into 2 well clustered factors [19]

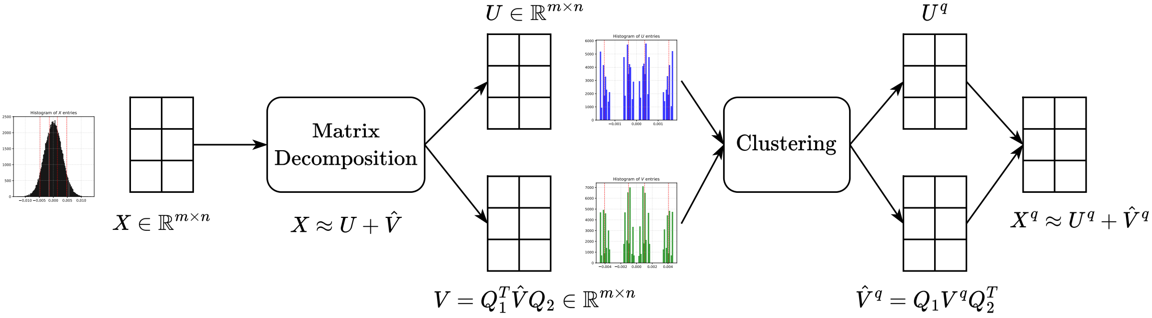


Figure 16: Scheme of post-training quantization approach.

[19] Quantization of Large Language Models with an Overdetermined Basis