Methods for training large-scale deep learning models

Ivan Oseledets,
Center for Artificial Intelligence Technologies @Skoltech

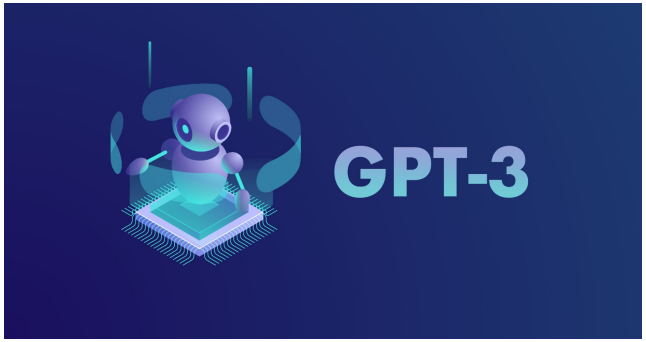
Skoltech
Skolkovo Institute of Science and Technology

Large scale deep learning models

- Standard way to train: stochastic gradient descent
- **For many domains**, it has been found that larger models and larger datasets give better performance
- It includes: natural language processing (NLP), self-supervised learning, vision transformers, contrastive language image pretraining, etc.

The time when you can train a large model on 1 GPU or on several GPUs is quickly going away!

Green AI



Requires 190 MW*hours, 85 tonnes of CO_2.

126 houses in Denmark

Or car drive from Moon

Skoltech
Skolkovo Institute of Science and Technology

Challenges in training

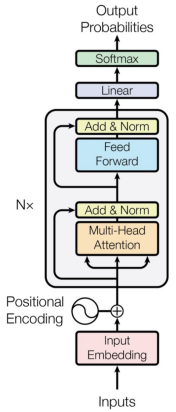
Two challenges:

Computational time

Memory for the model and batch

GPT-1 -> GPT-2 -> GPT-3

Models are getting bigger!



Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

How much (from «Language models are few-shot learners»)?

- Two challenges: computational time and memory for the model and batch

Model	Total train compute (PF-days)	Total train compute (flops)	Params (M)	Training tokens (billions)	Flops per param per token	Mult for bwd pass	Fwd-pass flops per active param per token	Frac of params active for each token
T5-Small	2.08E+00	1.80E+20	60	1,000	3	3	1	0.5
T5-Base	7.64E+00	6.60E+20	220	1,000	3	3	1	0.5
T5-Large	2.67E+01	2.31E+21	770	1,000	3	3	1	0.5
T5-3B	1.04E+02	9.00E+21	3,000	1,000	3	3	1	0.5
T5-11B	3.82E+02	3.30E+22	11,000	1,000	3	3	1	0.5
BERT-Base	1.89E+00	1.64E+20	109	250	6	3	2	1.0
BERT-Large	6.16E+00	5.33E+20	355	250	6	3	2	1.0
RoBERTa-Base	1.74E+01	1.50E+21	125	2,000	6	3	2	1.0
RoBERTa-Large	4.93E+01	4.26E+21	355	2,000	6	3	2	1.0
GPT-3 Small	2.60E+00	2.25E+20	125	300	6	3	2	1.0
GPT-3 Medium	7.42E+00	6.41E+20	356	300	6	3	2	1.0
GPT-3 Large	1.58E+01	1.37E+21	760	300	6	3	2	1.0
GPT-3 XL	2.75E+01	2.38E+21	1,320	300	6	3	2	1.0
GPT-3 2.7B	5.52E+01	4.77E+21	2,650	300	6	3	2	1.0
GPT-3 6.7B	1.39E+02	1.20E+22	6,660	300	6	3	2	1.0
GPT-3 13B	2.68E+02	2.31E+22	12,850	300	6	3	2	1.0
GPT-3 175B	3.64E+03	3.14E+23	174,600	300	6	3	2	1.0

Some computational costs

- **CLIP model**: 18 days on 592 V100 GPUs (ResNet backbone)
  12 days on 256 V100 GPUs
- **DALLE-E model**: 1024 V100 GPU
- **VIT model**: 2500 TPU v3 core-days
- **PanGu-alpha model**: 2048 Ascend 910 AI processors ? days.

Recent «world record»: Megatron-LM

Model size	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Batch size	Achieved teraFlOPs per GPU	Percentage of theoretical peak FLOPs	Achieved aggregate petaFLOPs
1.7B	2304	24	1.7	1	32	512	137	44%	4.4
3.6B	3072	30	3.6	2	64	512	138	44%	8.8
7.5B	4096	36	7.5	4	128	512	142	46%	18.2
18B	6144	40	18.4	8	256	1024	135	43%	34.6
39B	8192	48	39.1	16	512	1536	138	44%	70.8
76B	10240	60	76.1	32	1024	1792	140	45%	143.8
145B	12288	80	145.6	64	1536	2304	148	47%	227.1
310B	16384	96	310.1	128	1920	2160	155	50%	297.4
530B	20480	105	529.6	280	2520	2520	163	52%	410.2
1T	25600	128	1008.0	512	3072	3072	163	52%	502.0

https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/

How to train a big model?

We have: the model and the data.

We train using stochastic gradient descent (SGD)

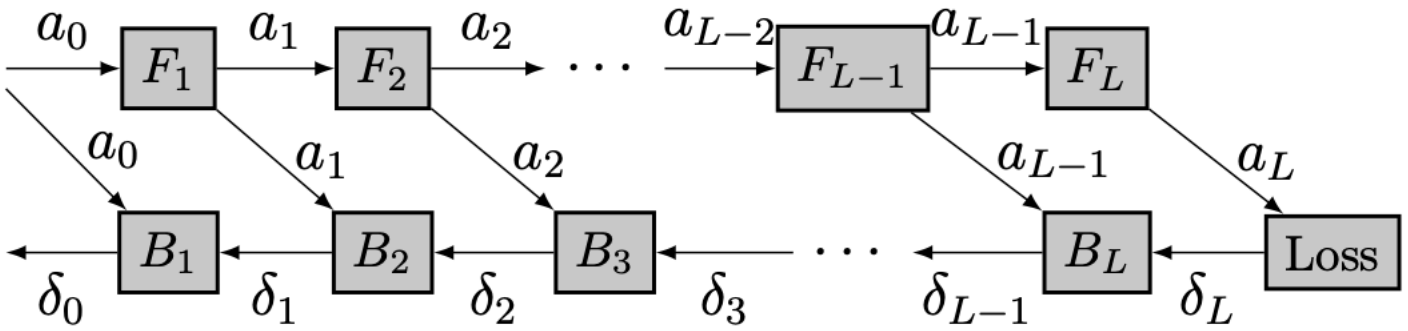
Given a batch x of size $B \times D$

we compute: forward $f(x, \theta)$, backward $\nabla_\theta f(x, \theta)$

f is a feedforward neural network

What is computed

For a backward pass, we need to store activations!
They consume 0.1 - 10x of the memory of the model
(depending on the batch size)



Type of parallelism

We need to use multi-node (GPU, Accelerators) system!

- **Data parallelism:** split the input batch into sub-batches
- **Model parallelism:** split the parameters of the model between different computational nodes
- **Pipeline parallelism:** minimize communication in forward & backward passes
- **Tensor parallelism**: split the feature dimension between different GPUs

Memory reduction techniques

We can use CPU memory to store things:

- **Activation checkpointing:** store some activations in the CPU memory, recompute others (increases computational time, decreases memory cost)
- **Offloading:** upload parts of the model weights/ activations to the CPU

Faster computations

Techniques that do not fall into ones above:

- **Low-precision computations:** use FP16 (or even less) for computations. Increases throughput, decreases memory may lead to bad convergence; low-precision approximation of (some) activations.
- **1-bit optimizers + PowerSGD:** Approximate gradients (even 1-bit), approximate the block

Data parallelism

The classical approach, implemented in software, is **data parallelism**

Each computational unit holds a copy of the model, processes its own batch and aggregates the gradients

This is equivalent to large batch; It also requires scatter-gather operation

$$g_1 = \nabla_\theta f(x_1, \theta)$$

$$g_2 = \nabla_\theta f(x_2, \theta)$$

$$g_3 = \nabla_\theta f(x_3, \theta)$$

$$g_{all} = g_1 + g_2 + g_3$$

Data parallelism: large-batch training

This is equivalent to **large batch**;

How can we train with large batch size without worse convergence?

We need to scale the learning rate accordingly.

$$g_1 = \nabla_\theta f(x_1, \theta)$$

$$g_2 = \nabla_\theta f(x_2, \theta)$$

$$g_3 = \nabla_\theta f(x_3, \theta)$$

$$g_{all} = g_1 + g_2 + g_3$$

Large-batch training

This is equivalent to **large batch**;

We need to scale the learning rate accordingly.

Theorem 1 ((Ghadimi & Lan, 2013b)). *With large batch $b = T$ and using appropriate learning rate, we have the following for the iterates of* SGD*:*

$$\mathbb{E}\left[\|\nabla f(x_a)\|^2\right] \leq O\left(\frac{(f(x_1) - f(x^*))L_\infty}{T} + \frac{\|\sigma\|^2}{T}\right).$$

You Y. et al. Large batch optimization for deep learning: Training BERT in 76 minutes //arXiv preprint arXiv:1904.00962. 2019.

Large-batch training

This is equivalent to **large batch**;

For example, for SGD we can have

$$x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|g_t^{(i)}\|} g_t^{(i)}$$

You Y. et al. Large batch optimization for deep learning: Training BERT in 76 minutes //arXiv preprint arXiv:1904.00962. 2019.

Large-batch training

This is equivalent to **large batch**;

LARS and LAMB (ADAM + Layer normalization)

Algorithm 1 LARS

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameter $0 < \beta_1 < 1$, scaling function ϕ, $\epsilon > 0$
Set $m_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$
  Compute $g_t = \frac{1}{|\mathcal{S}_t|} \sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1)(g_t + \lambda x_t)$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|m_t^{(i)}\|} m_t^{(i)}$ for all $i \in [h]$
end for

Algorithm 2 LAMB

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameters $0 < \beta_1, \beta_2 < 1$, scaling function ϕ, $\epsilon > 0$
Set $m_0 = 0$, $v_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$.
  Compute $g_t = \frac{1}{|\mathcal{S}_t|} \sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$.
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $m_t = m_t / (1 - \beta_1^t)$
  $v_t = v_t / (1 - \beta_2^t)$
  Compute ratio $r_t = \frac{m_t}{\sqrt{v_t + \epsilon}}$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|r_t^{(i)} + \lambda x_t^{(i)}\|} (r_t^{(i)} + \lambda x_t^{(i)})$
end for

You Y. et al. Large batch optimization for deep learning: Training BERT in 76 minutes //arXiv preprint arXiv:1904.00962. 2019.

Large-batch training

This is equivalent to **large batch**;

BERT training on TPUS.

Solver	batch size	steps	F1 score on dev set	TPUs	Time
Baseline	512	1000k	90.395	16	81.4h
LAMB	512	1000k	91.752	16	82.8h
LAMB	1k	500k	91.761	32	43.2h
LAMB	2k	250k	91.946	64	21.4h
LAMB	4k	125k	91.137	128	693.6m
LAMB	8k	62500	91.263	256	390.5m
LAMB	16k	31250	91.345	512	200.0m
LAMB	32k	15625	91.475	1024	101.2m
LAMB	64k/32k	8599	90.584	1024	76.19m

You Y. et al. Large batch optimization for deep learning: Training BERT in 76 minutes //arXiv preprint arXiv:1904.00962. 2019.

Memory constraints

Large models **do not fit** to a GPU memory;

A rule of thumb is that for **M parameters we need 12M bytes**

12 = 4 bytes x 3 optimizer states

Activations take **(0.1 - 10)** x number of parameters.

Without offloading/checkpointing maximum is **2 billion** on a V100 GPU.

Optimizer parallelism

DeepSpeed framework (https://github.com/microsoft/DeepSpeed) proposed ZeRO:

The idea is to split the optimizer state and gradients between nodes

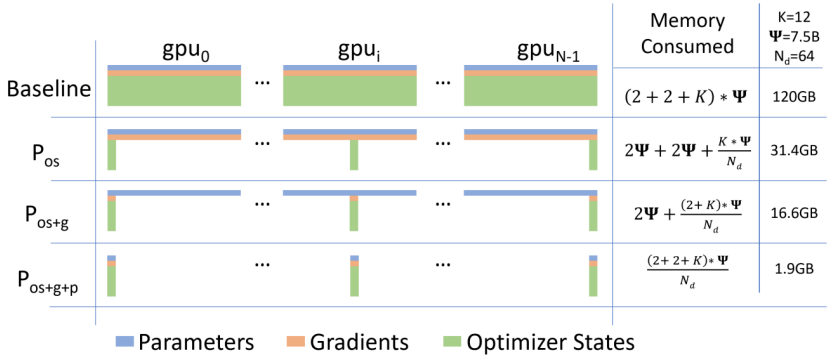


Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

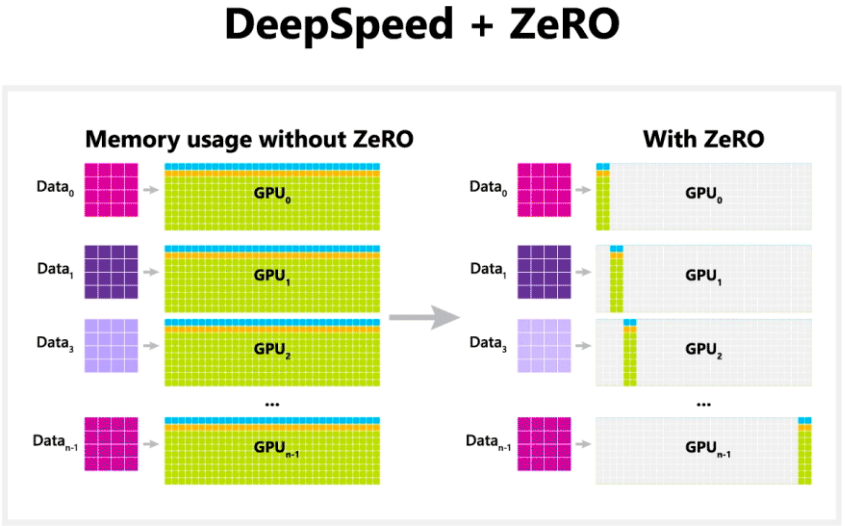
Rajbhandari S. et al. Zero: Memory optimizations toward training trillion parameter models //SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. – IEEE, 2020. – C. 1-16.

ZERO idea explained

Each node stores only its weights;

It computes the gradient only for its weights

When parameters are required for forward or backward, they are received from broadcast



Rajbhandari S. et al. Zero: Memory optimizations toward training trillion parameter models //SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. – IEEE, 2020. – C. 1-16.

What else?

Zero reduces the computational cost, but induces communication (i.e., we need to broadcast and wait)

Alternatives: split the model **vertically** or **horizontally**

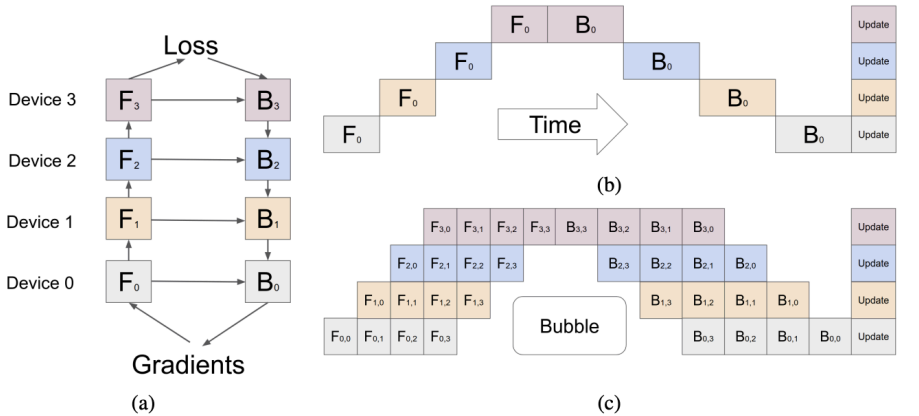
One of the promising ideas is pipeline parallelism.

Pipeline parallelism

The batch is split into micro batches

The model is split by layers

The G-Pipe approach interleaves computations with communication



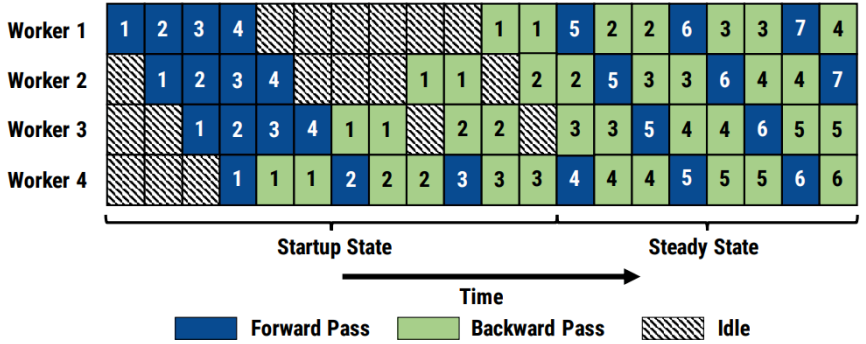
Huang Y. et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism //Advances in neural informa processing systems. – 2019. – T. 32. – C. 103-112.

Pipeline parallelism: is that optimal?

Pipedream: another scheduler.

It takes into account:

- forward/backward time on layer l
- size of activations;
- Size of parameters

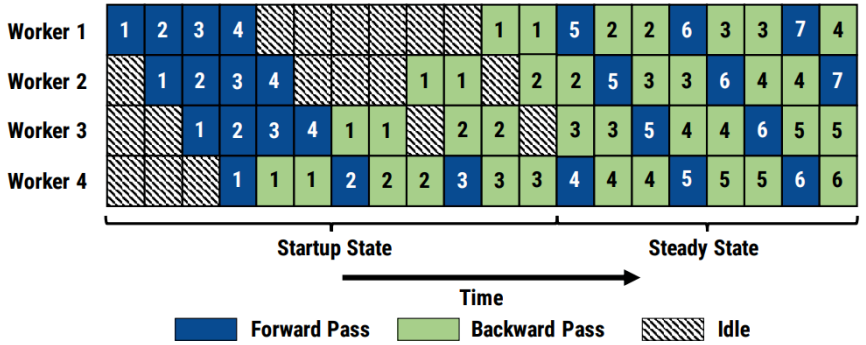


Narayanan D. et al. PipeDream: generalized pipeline parallelism for DNN training //Proceedings of the 27th ACM Symposium on Operating Systems Principles. – 2019. – С. 1-15.

Pipeline parallelism: is that optimal?

Pipedream uses dynamical programming to recursively split the computation between workers

The allocation problem is NP-complete (A. Shilova, thesis).

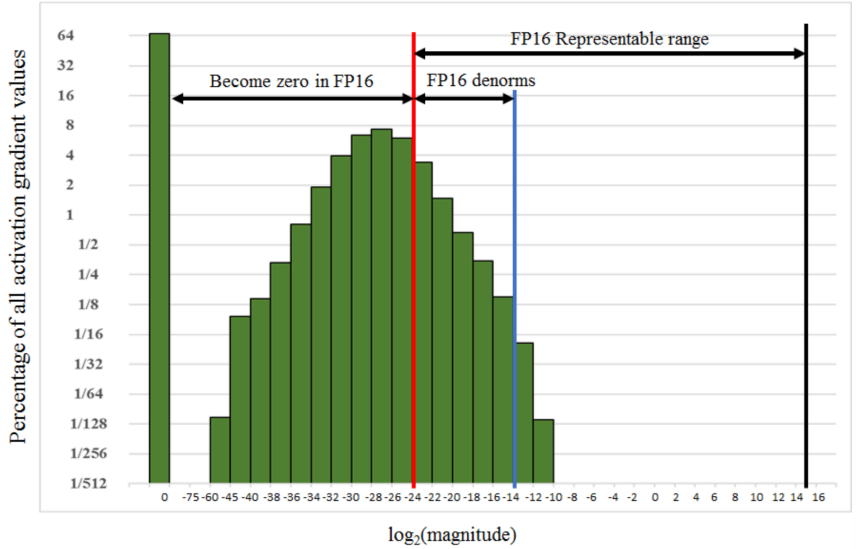


Narayanan D. et al. PipeDream: generalized pipeline parallelism for DNN training //Proceedings of the 27th ACM Symposium on Operating Systems Principles. – 2019. – С. 1-15.

Low-precision optimization

Another way to speedup computation is to use low-precision computations

Using FP16 to store optimizer states leads to instabilities: underflow / overflow (norm distribution)



Low-precision optimization (from DALLE paper)

«The most challenging part was to use FP16 training»

- Scale resblocks (128 scales) + many tricks
- Not all parameters are used
- Underflow when dividing by M(!)

8-bit optimizer

Very promising idea to use 8-bit nonlinear (!) quantization

- Maintains 32-bit performance at a fraction of memory footprint
- 4 (Momentum) and 8 (Adam) bytes / weight -> 1 and 2 bytes / weight

Dettmers T. et al. 8-bit Optimizers via Block-wise Quantization //arXiv preprint arXiv:2110.02861. – 2021.

8-bit optimizer: idea

- Non-linear quantization
- Dynamic tree quantization: dynamic exponent and fraction
- Blocks of weights are quantized / normalized independently

$$N = \max(T), \quad T_i^Q = \arg\min | Q_j^{\text{map}} - \frac{T_i}{N} |$$

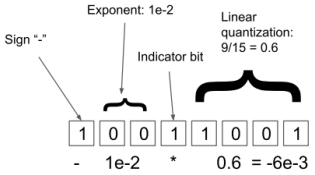


Figure 2: Dynamic tree quantization.

Dettmers T. et al. 8-bit Optimizers via Block-wise Quantization //arXiv preprint arXiv:2110.02861. – 2021.

PowerSGD

To minimize communication, a simple idea is very useful: low-rank approximation

Each layer gradient is a matrix

We approximate such matrix by low-rank matrix using randomized SVD

It minimizes communications!

Algorithm 1 Rank-r POWERSGD compression

1: The update vector Δ_w is treated as a list of tensors corresponding to individual model parameters. Vector-shaped parameters (biases) are aggregated uncompressed. Other parameters are reshaped into matrices. The functions below operate on such matrices independently. For each matrix $M \in \mathbb{R}^{n \times m}$, a corresponding $Q \in \mathbb{R}^{m \times r}$ is initialized from an i.i.d. standard normal distribution.
2: **function** COMPRESS+AGGREGATE(update matrix $M \in \mathbb{R}^{n \times m}$, previous $Q \in \mathbb{R}^{m \times r}$)
3: $P \leftarrow MQ$
4: $P \leftarrow$ ALL REDUCE MEAN(P) ▷ Now, $P = \frac{1}{W}(M_1 + \ldots + M_W)Q$
5: $\hat{P} \leftarrow$ ORTHOGONALIZE(P) ▷ Orthonormal columns
6: $Q \leftarrow M^\top \hat{P}$
7: $Q \leftarrow$ ALL REDUCE MEAN(Q) ▷ Now, $Q = \frac{1}{W}(M_1 + \ldots + M_W)^\top \hat{P}$
8: **return** the compressed representation $(\hat{P}, Q)$.
9: **end function**
10: **function** DECOMPRESS($\hat{P} \in \mathbb{R}^{n \times r}, Q \in \mathbb{R}^{m \times r}$)
11: **return** $\hat{P}Q^\top$
12: **end function**

Vogels T., Karinireddy S. P., Jaggi M. PowerSGD: Practical low-rank gradient compression for distributed optimizat
Advances In Neural Information Processing Systems 32 (Nips 2019). – 2019. – Т. 32. – №. CONF.

Checkpointing

How can we deal with memory constraints?

What if even batch size = 1 does not fit?

The answer is **checkpointing:**

store some activations and recompute the rest

Checkpointing

How can we deal with memory constraints?

We record the timing for backward and forward for each block

We have memory constraint

We solve **dynamic programming task** (A. Shilova, O. Beaumont, Lionel Eyraud-Dubois)

Checkpointing

How can we deal with memory constraints?

We record the timing for backward and forward for each block

We have memory constraint

We solve **dynamic programming task** (A. Shilova, O. Beaumont, Lionel Eyraud-Dubois)

```
chk_gpu.compute_sequence(mem_limit=500*1024*1024)
print(chk_gpu.sequence)
print("Duration", chk_gpu.get_expected_makespan(), "Memory usage", rotor.memory.MemSize(chk_gpu.get_expected_memory()))
```

```
[CF_0, Fe_1, CF_2, Fe_3, CF_4, Fe_5, CF_6, Fe_7, CF_8, Fe_9, CF_10, Fe_11, CF_12, Fe_13, CF_14, Fe_15, CF_16, Fe_17, CF_18, Fe_19, CF_20, Fe_21, Fe_22, Fe_23, L, B_23, B_2
Duration 641.2084645456862 Memory usage 345.0MiB
```

Offloading

If the model does not fit to a single GPU, we can use **offloading** (implemented in ZERO-Infinity)

Different variants:
we can offload activations;
we can offload weights

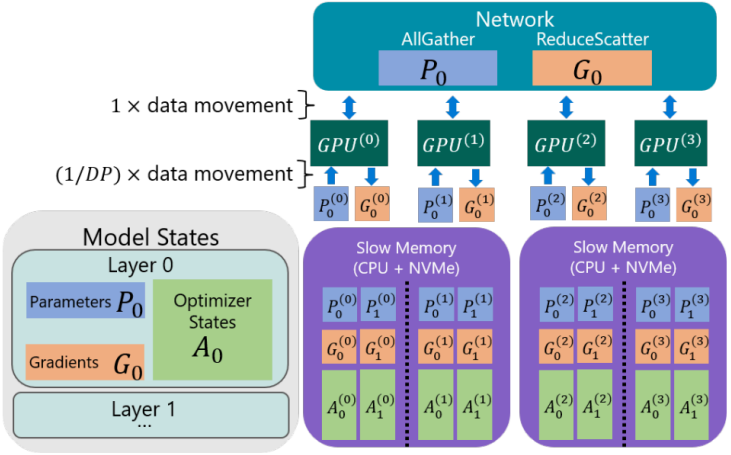


Figure 4: A snapshot of ZeRO-Infinity training a model with two layers on four data parallel (DP) ranks. Communication for the backward pass of the first layer is depicted. Partitioned parameters are moved from slow memory to GPU and then collected to form the full layer. After gradients are computed, they are aggregated, re-partitoned, and then offloaded to slow memory. Layers are denoted with subscripts and DP ranks are denoted with superscripts. For example, $P_0^{(2)}$ is the portion of layer 0's parameters owned by $GPU^{(2)}$.

Activations offloading

Dynamical programming (moving from one layer to another and determining the constraints on this step) gives an optimal solution for a sequential model

Picture from A. Shilova thesis

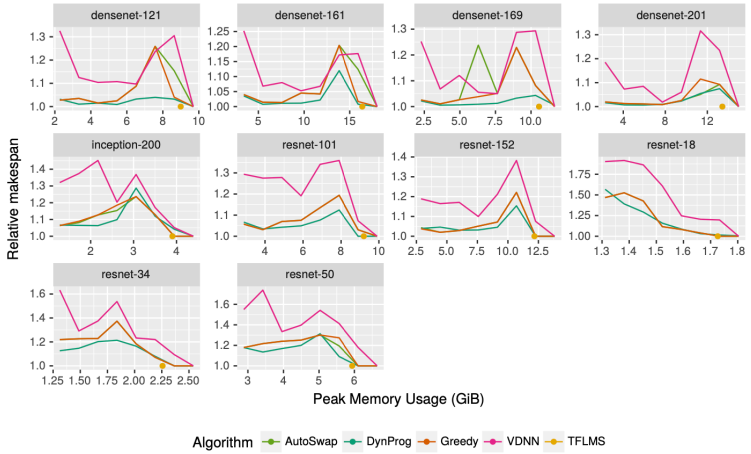


Figure 3.5: Relative makespan (with respect to the lower bound) obtained by different algorithms for different memory limits. Experimental results are provided for image size 224 and batch size 32 (top), image size 500 and batch size 16 (bottom).

What can be done

The existing software allows to reach 30-40% of the peak GPU performance

In order to get a 10x speedup, we need to reduce the number of flops

This can be done by approximating the inference (quantization, partial model updates)

This would also require new optimization methods that are robust to such scenarios
(remember quantization & large batch)

Our result (1): few-bit backward

- We replace the derivatives of the activations with low-bit approximation
- 15-20% less memory
- Already used in training

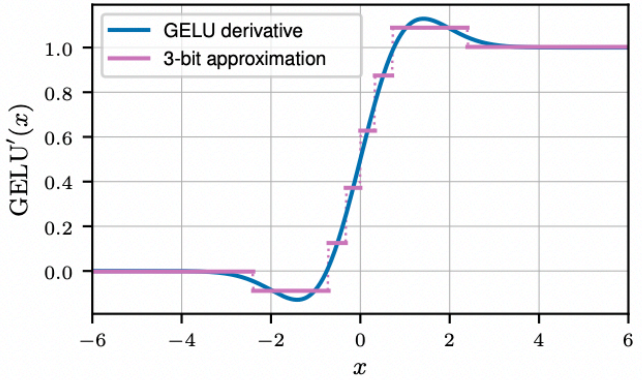


Figure 1. Optimized 3-bit piecewise-constant approximation of the derivative of the GELU activation function.

Few-Bit Backward: Quantized Gradients of Activation Functions for Memory Footprint Reduction

Georgii Novikov [1] Daniel Bershatsky [1] Julia Gusak [1] Alex Shonenkov [2] Denis Dimitrov [2 3] Ivan Oseledets [1 4]

Skoltech
Skolkovo Institute of Science and Technology

Programming deep learning from scratch: NNTile (C++)

- Model problem: 1 forward and backward step of the transformer-like model
- Memory required: 76.25 gigabytes (float32)
- Total amount of flops

Time, s: 31.0794 GFLOP/s: 13868 GFLOP/s/GPU: 13868

1. Time, s: 16.2025 GFLOP/s: 26601.4 GFLOP/s/GPU: 13300.7
2. Time, s: 11.5464 GFLOP/s: 37328.4 GFLOP/s/GPU: 12442.8
3. Time, s: 9.87117 GFLOP/s: 43663.4 GFLOP/s/GPU: 10915.85
4. Time, s: 9.87117 GFLOP/s: 43663.4 GFLOP/s/GPU: 10915.85

Idea is to replace matrices with tiles, and asyncronously process them

Custom tensor layers

- We can replace fully connected layers by tensor-train matrix decomposition (Oseledets, 2009)
- For training, it reduces the number of parameters by 1.5x without significant reduction of accuracy
- Tensors are efficient representation of multidimensional data

APPROXIMATION OF PROBABILITY DISTRIBUTIONS FROM SAMPLES USING TENSORS

We are given samples $x_1, \ldots, x_N$ from the probability distribution

$$p(x) \approx q_\theta(x)$$

$$q_\theta(x) = \langle \alpha_\theta, \Phi(x) \rangle = \sum_{k=1}^{K} \alpha_{\theta,k} f_k(x)$$

Tensor-product basis: $\Phi(x) = f(x_1) \otimes \ldots \otimes f_d(x_d), \quad f_k(x) \in \mathbb{R}^{m_k}$

We put tensor-train constraints on α, which is a d-dimensional tensor!

LOSS FUNCTION

As a loss function, we use $\mathscr{L}(p(x) - q_\theta(x))^2 dx = \int q_\theta^2 dx - 2E_{x\sim p} q_\theta(x) + \text{const}$

All these terms are computable.

SQUARED TT-MODEL

TT-format for the density is not positive;

We also propose to use squared TT model

$$\hat{q} = q_\theta^2(x)$$

It happens, that the complexity of the basic operations (sampling, loss evaluation, etc.) does not grow significantly with respect to the ranks.

WHY TT IS GOOD?

- Sampling is cheap
- Likelihood is available
- Optimization can be done by Riemannian optimization

Table 1: Comparison of the capabilities of different density estimation models. *FFJORD does not use true log-likelihood i the training process and instead uses its unbiased estimate.

Method	Exact Sampling	Tractable LL	No middle-man Training	Computation of CDF
FFJORD	✓	✓*	✓*	✗
Normalizing Flows	✓	✓	✓	✗
GANs	✓	✗	✗	✗
VAEs	✓	✗	✓	✗
Autoregressive	✓	✓	✓	✗
Energy-based	✗	✗	✗	✗
TTDE (ours)	✓	✓	✓	✓

EXPERIMENTS

- Sampling is cheap
- Likelihood is available

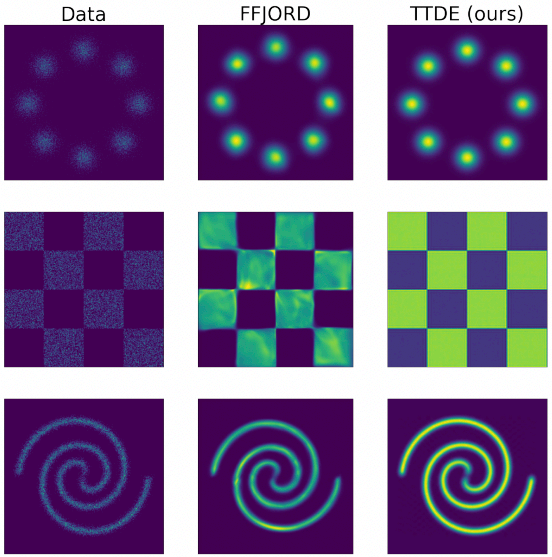


Figure 1: Comparison of TTDE and FFJORD models on 2-dimensional toy distributions.

zation

	Random init.	Rank-1 init.
Adam	5	11
Riemannian	12	32

Table 2: Experiment with mixture of 7 Gaussians in 3D with additional dimensions containing only noise. We report the maximum dimensionality for which approximation of the density converges to the true one for different initialization settings and optimization methods used.

EXPERIMENTS

- Sampling is cheap
- Likelihood is available
- Optimization can be done by Riemannian optimization

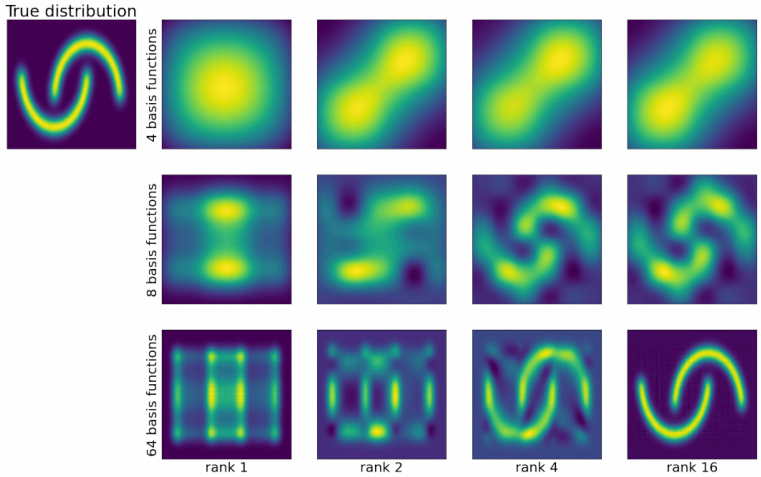


Figure 2: Approximations of “two moons” distribution by TTDE for different basis function set sizes and TT-ranks.

EXPERIMENTS

- Sampling is cheap
- Likelihood is available
- Optimization can be done by Riemannian optimization

	POWER	GAS	HEPMASS	MINIBOONE	BSDS300
Dataset dimensionality	6	8	21	43	64
Gaussians	-7.74	-3.58	-27.93	-37.24	96.67
MADE	-3.08	3.56	-20.98	-15.59	148.85
Real NVP	0.17	8.33	-18.71	-13.84	153.28
Glow	0.17	8.15	-18.92	-11.35	155.07
FFJORD	0.46	8.59	**-14.92**	**-10.43**	**157.40**
Squared TTDE (ours)	**0.46**	**8.93**	-21.34^*	-28.77^*	143.30

Skoltech
Skolkovo Institute of Science and Technology

EXPERIMENTS

- Sampling is cheap
- Likelihood is available
-

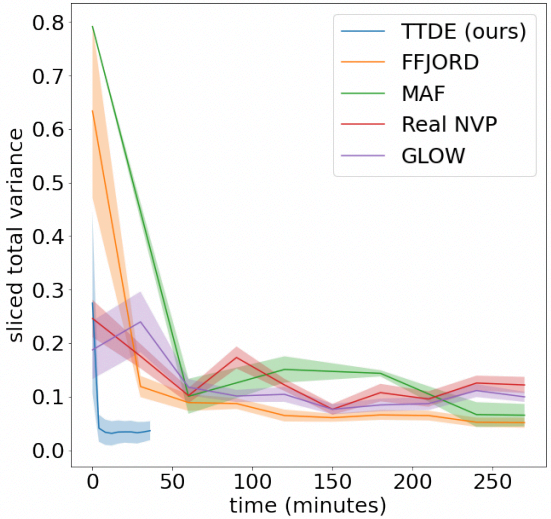


Figure 4: Dependence of the sliced total variation w.r.t. the training time for models trained on 6-dimensional UCI POWER dataset.

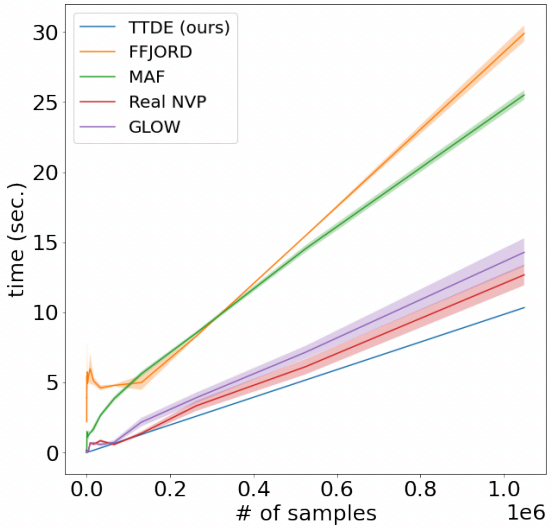


Figure 5: Dependence of the sampling time w.r.t. the number of samples to be generated for 6-dimensional space for models trained on UCI POWER dataset. Our model outperforms its competitors and shows 2.6, 2.5, 1.4 and 1.2 times speedups compared to FFJORD, MAF, GLOW and Real NVP respectively.

EXPERIMENTS

- Sampling is cheap
- Likelihood is available
-

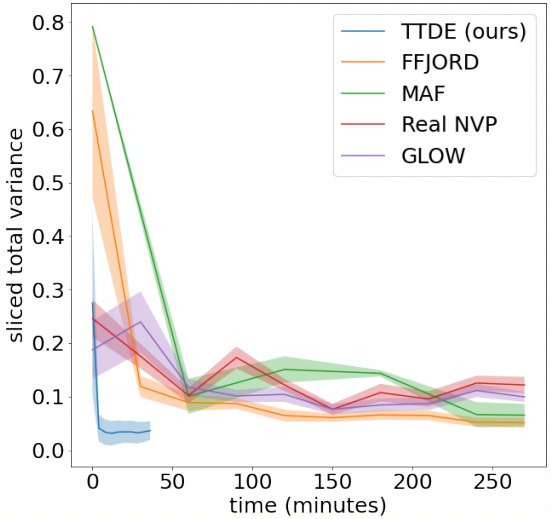


Figure 4: Dependence of the sliced total variation w.r.t. the training time for models trained on 6-dimensional UCI POWER dataset.

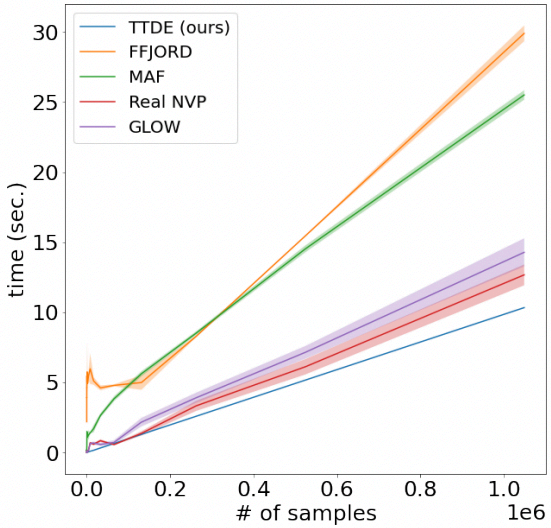


Figure 5: Dependence of the sampling time w.r.t. the number of samples to be generated for 6-dimensional space for models trained on UCI POWER dataset. Our model outperforms its competitors and shows 2.6, 2.5, 1.4 and 1.2 times speedups compared to FFJORD, MAF, GLOW and Real NVP respectively.

REFERENCE

G. Novikov, M. Panov, I. Oseledets Tensor-train density estimation, UAI, 2021.