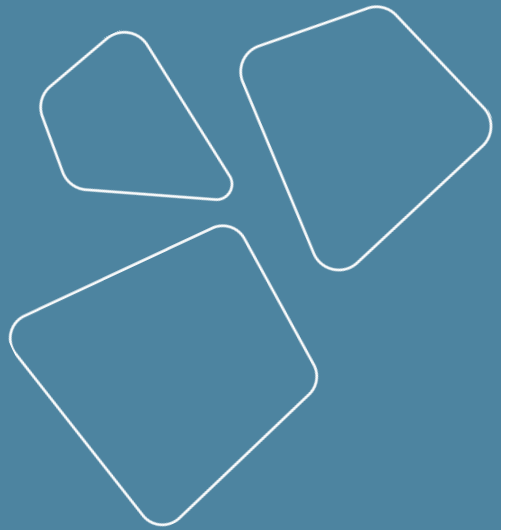
Ускорение обучения. Подготовка к деплою

Astafurov Eugene

MIPT, MSU, fall 2023

Timeline



- Mixed Precision Training и тензорные ядра
- TensorRT

Часть 1: Mixed Precision

Ускоряем обучение

girafe ai

01

TLDR

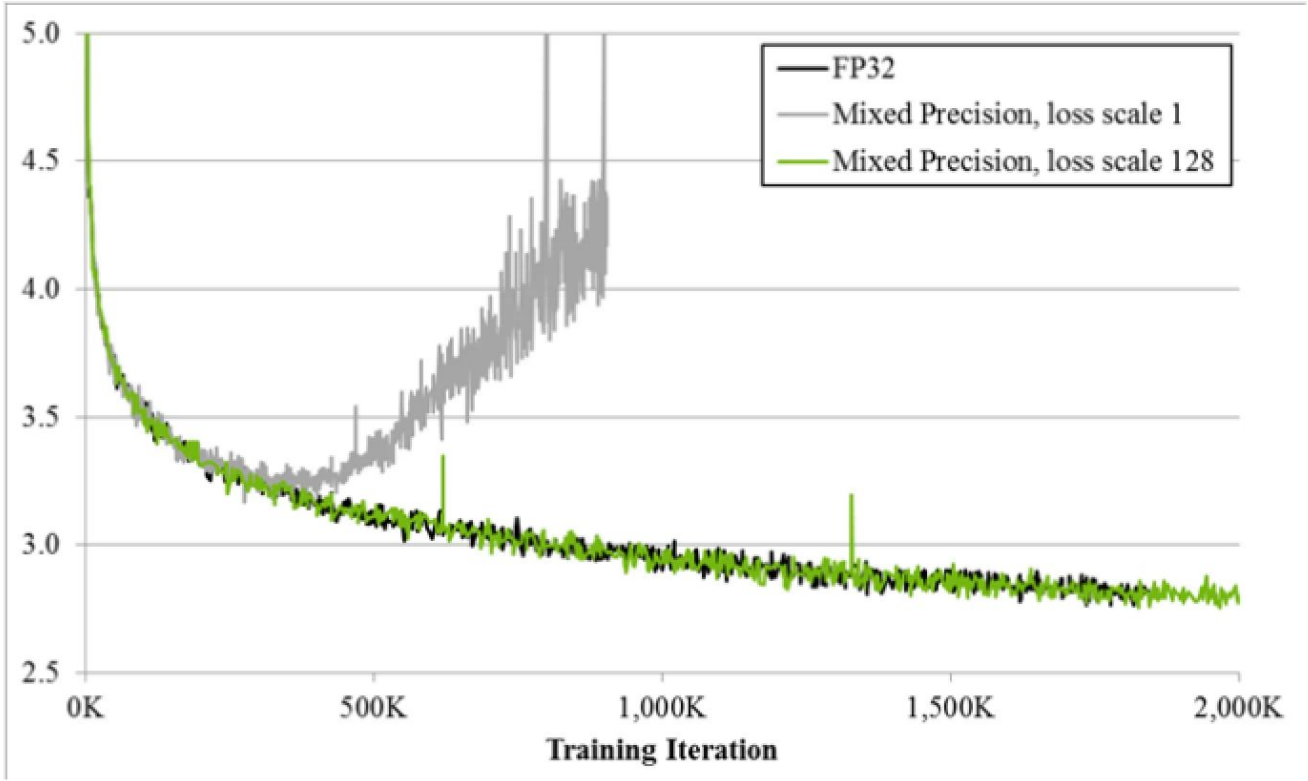
Делаем:

- Заставляем модель использовать FP16 там, где это выгодно
- Добавляем Loss Scale для борьбы с затуханием градиента

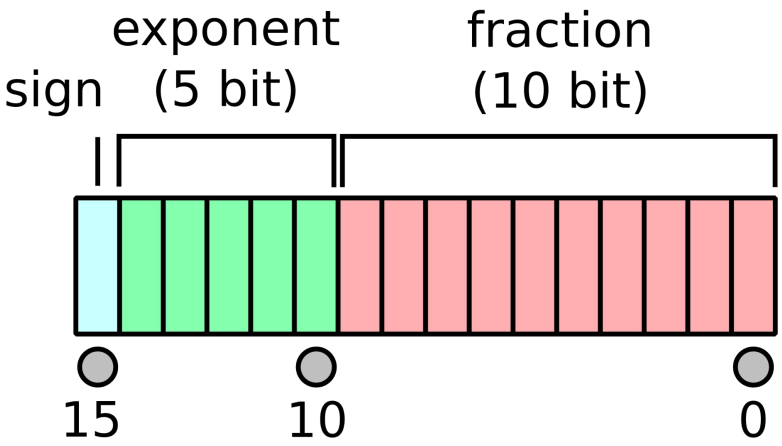
Получаем:

- Более быстрое обучение, ускорение до 8x раз (нет)
- Уменьшение количества необходимой видеопамяти
  - То есть можем использовать батчи большего размера
  - Меньше оверхед на память

Пример


5.0
4.5
4.0
3.5
3.0
2.5
0K
500K
1,000K
1,500K
2,000K
Training Iteration
FP32
Mixed Precision, loss scale 1
Mixed Precision, loss scale 128

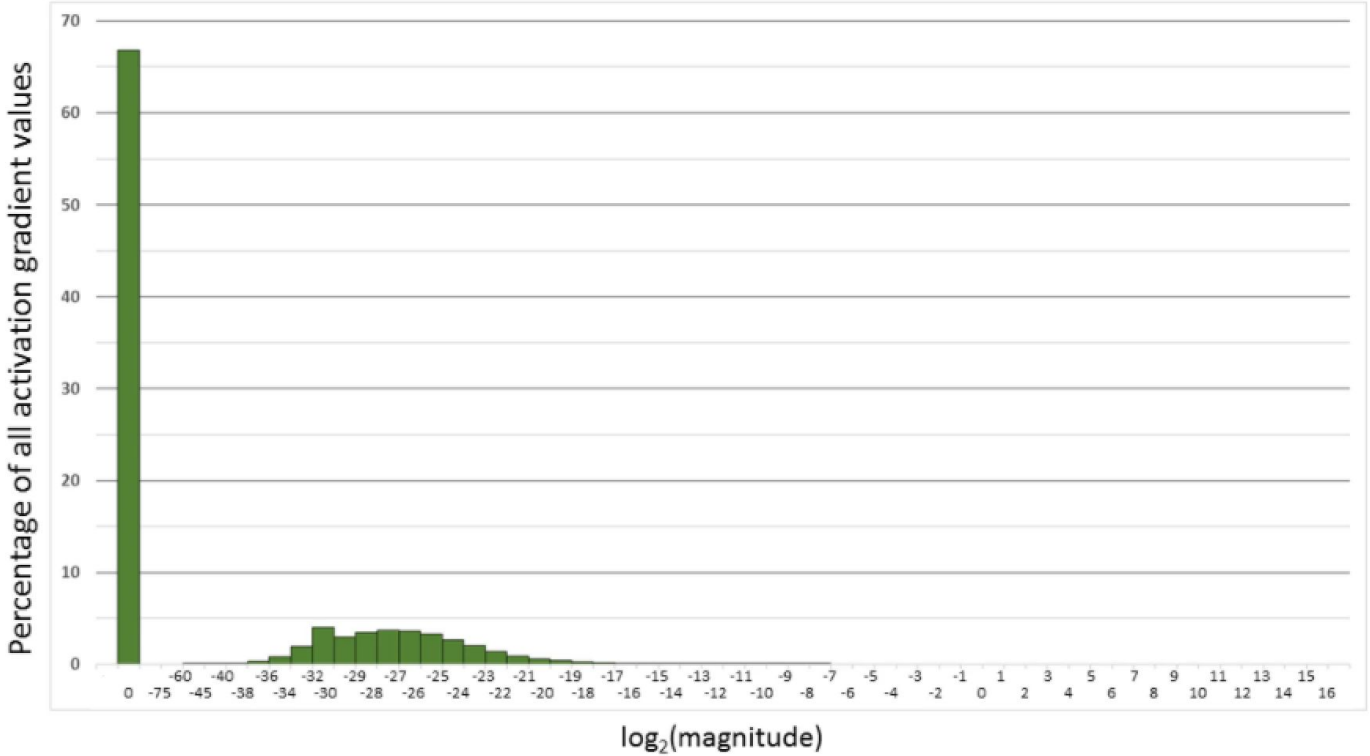
Half Precision



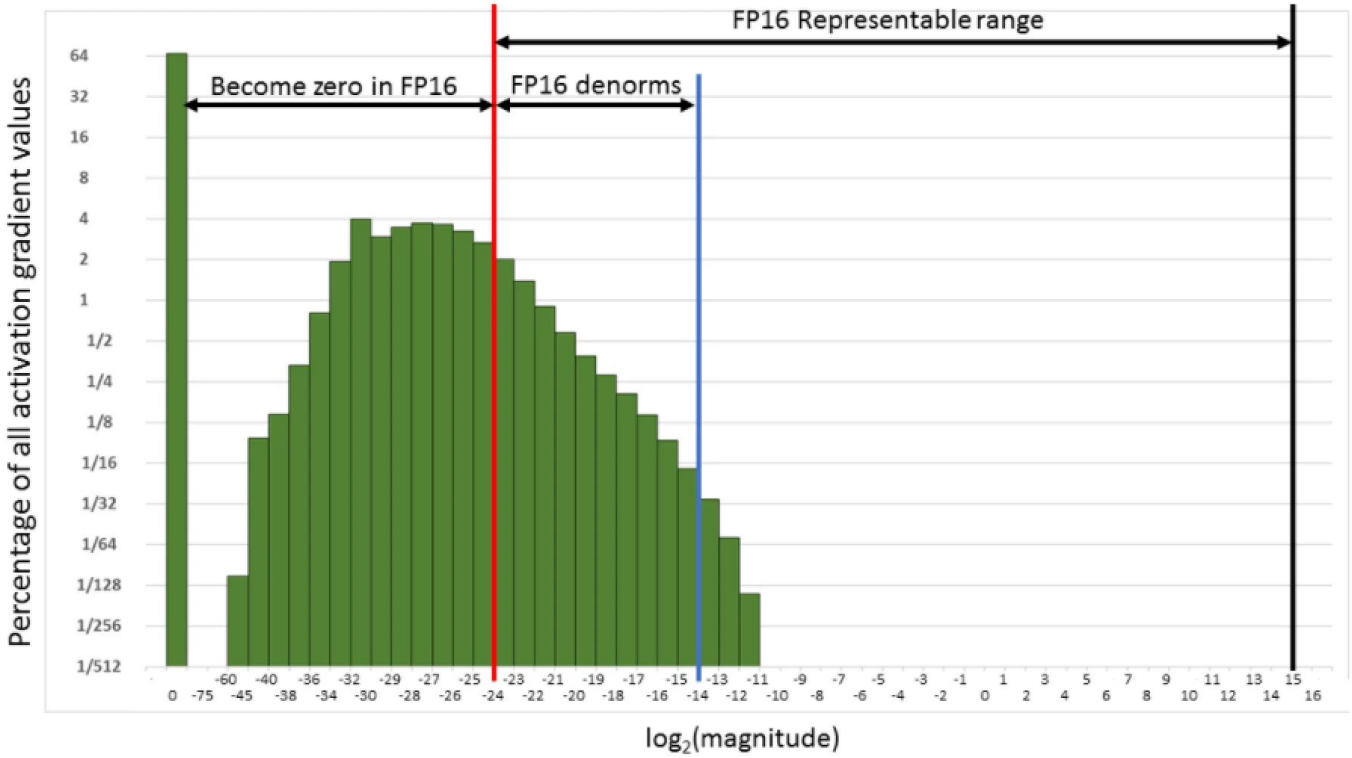
- Normalized:
  - от **2^(-14)** до **2^(15)**
- Denormalized:
  - от **2^(-24)** до **2^(-15)**

- Максимальное нормализованное:
  - **65504**
- Минимальное нормализованное:
  - **~0.00006**
- Минимальное **де**нормализованное:
  - **6e-8**

Магнитуды градиентов FP32



Магнитуды градиентов FP32



Loss Scaling

1. Maintain a primary copy of weights in FP32.
2. For each iteration:
   - Make an FP16 copy of the weights.
   - Forward propagation (FP16 weights and activations).
   - Multiply the resulting loss with the scaling factor S.
   - Backward propagation (FP16 weights, activations, and their gradients).
   - Multiply the weight gradient with 1/S.
   - Complete the weight update (including gradient clipping, etc.).

Scaling Factor

Статический подход:

- Выберем S так, что его произведение с максимальным абсолютным значением градиента было < 65504

Динамический подход:

- ???????

Scaling Factor

1. Maintain a primary copy of weights in FP32.
2. Initialize S to a large value.
3. For each iteration:
    a. Make an FP16 copy of the weights.
    b. Forward propagation (FP16 weights and activations).
    c. Multiply the resulting loss with the scaling factor S.
    d. Backward propagation (FP16 weights, activations, and their gradients).
    e. If there is an Inf or NaN in weight gradients:
        i. Reduce S.
        ii. Skip the weight update and move to the next iteration.
    f. Multiply the weight gradient with 1/S.
    g. Complete the weight update (including gradient clipping, etc.).
    h. If there hasn't been an Inf or NaN in the last N iterations, increase S.

AMP - Automatic Mixed Precision

Steps:

- Converting the model to use the **float16** data type **where possible**.
- **Keeping float32 master weights** to accumulate per-iteration weight updates.
- Using **loss scaling** to preserve small gradient values.

Тензорные ядра и AMP

1. **Satisfy** Tensor Core shape **constraints**
2. **Increase arithmetic intensity**
3. **Decrease** fraction of work in **non-Tensor Core operations**

1. Satisfy Constraints

Matmul:

1. **FP16** - Gemm`s M, N, K кратны **8**
2. **INT8** - Gemm`s M, N, K кратны **16**

Conv:

1. **FP16** - in_channels, out_channels кратны **8**
2. **INT8** - in_channels, out_channels кратны **16**

1. Satisfy Constraints

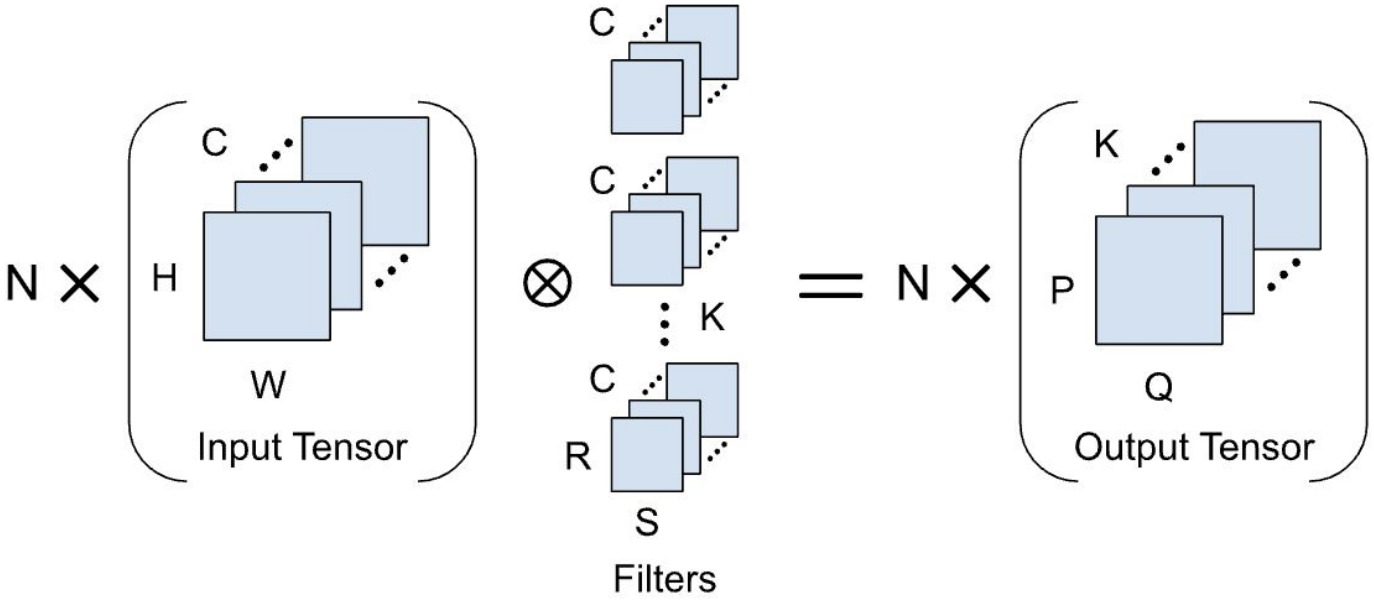
Linear Gemm

Table 1. Mapping of inputs, outputs, and batch size to GEMM parameters M, N, K.

Computation Phase	M	N	K
Forward Propagation	Number of outputs	Batch size	Number of inputs
Activation Gradient	Number of inputs	Batch size	Number of outputs
Weight Gradient	Number of inputs	Number of outputs	Batch size

1. Satisfy Constraints

Conv Gemm



2. Increase Arithmetic Intensity

Model architecture:

- Prefer dense math operations.
  - For example, vanilla convolutions have much higher arithmetic intensity than depth-wise separable convolutions.

- Prefer wider layers when possible.

3. Decrease Non-Tensor Work

Как считается ускорение:

1 / (x + ((1-x) / y)

- x - доля времени в Tensor-Core операциях
- y - ускорение от использования тензорных ядер

Например, если *половина* времени обучения тратится на операции в тензорных ядрах, а в этих операциях ускорение равно *5*, тогда общее ускорение от использования смешанной точности будет равно *1 / (0.5 + (0.5 / 5)) = 1.67*

Allow / Deny / Infer Lists

- **AllowList**:
  - Convolutions
  - Fully-connected layers
- **DenyList**:
  - Large reductions
  - Cross entropy loss
  - L1 Loss
  - Exponential
- **InferList**:
  - Element-wise operations (add, multiply by a constant)

Тут: https://github.com/NVIDIA/apex/blob/master/apex/amp/lists/functional_overrides.py

Hardware requirements

1. Наличие Тензорных Ядер :)

Примеры ускорений

Model Script	Framework	Data Set	FP32 Accuracy	Mixed Precision Accuracy	FP32 Throughput	Mixed Precision Throughput	Speed-up
BERT Q&A	TensorFlow	SQuAD	90.83 Top 1%	90.99 Top 1%	66.65 sentences/sec	129.16 sentences/sec	1.94
SSD w/RN50	TensorFlow	COCO 2017	0.268 mAP	0.269 mAP	569 images/sec	752 images/sec	1.32
GNMT	PyTorch	WMT16 English to German	24.16 BLEU	24.22 BLEU	314,831 tokens/sec	738,521 tokens/sec	2.35
Neural Collaborative Filter	PyTorch	MovieLens 20M	0.959 HR	0.960 HR	55,004,590 samples/sec	99,332,230 samples/sec	1.81
U-Net Industrial	TensorFlow	DAGM 2007	0.965-0.988	0.960-0.988	445 images/sec	491 images/sec	1.10
ResNet-50 v1.5	MXNet	ImageNet	76.67 Top 1%	76.49 Top 1%	2,957 images/sec	10,263 images/sec	3.47
Tacotron 2 / WaveGlow 1.0	PyTorch	LJ Speech Dataset	0.3629/-6.1087	0.3645/-6.0258	10,843 tok/s 257,687 smp/s	12,742 tok/s 500,375 smp/s	1.18/1.94

Performance Limitations

Пусть:

- *T_mem* - время доступов к памяти
- *T_math* - время выполнения арифметических операций

Суммарное время: *max(T_mem, T_max)*.

Performance Limitations

Оценим:

- *T_mem = #bytes / BW_mem*
- *T_math = #ops / BW_math*

, тут *#bytes* - количество необходимых байт в памяти, *#ops* - количество мат операций, *BW_mem* - Memory Bandwidth, а *BW_math* - Math Bandwidth.

T_math > T_mem ⇔ *#ops / BW_math > #bytes / BW_mem* ⇔

⇔ *#ops / #bytes > BW_math / BW_mem*.

#ops / #bytes -- это и есть арифметическая интенсивность

Performance Limitations

Рассмотрим Nvidia V100:

- BW_mem_onchip = 3.1 Tb/s
- BW_mem_offchip = 900 GB/s
- BW_math = 125 FP16 TFLOPS

=> **арифметическая интенсивность между 40 и 140** (в зависимости от источника данных для операции).

Performance Limitations

Рассмотрим конкретный пример с полносвязанным слоем:

- in = 4096, out = 1024, bs = 512

- FLOP = 2 * bs * in * out ~= 4 * 10**9 flop (4 GFLOP), (2 так как mul-acc операция)
- Bytes (weight) = in * out * 2, (2 байта тк fp16)
- Bytes (inputs) = bs * in * 2
- Bytes (out) = bs * out * 2
- Bytes (total) = in * out * 2 + bs * 2 * (in + out) ~= 0.01 * 10**9

#ops / #bytes = 400 => для nvidia V100 **ТОЧНО** ограничено арифметикой

Performance Limitations

Рассмотрим пример с ReLU идущим за этим линейным слоем:

- out = 1024, bs = 512

- FLOP = bs * out ~= 0.5 * 10**6 (MFLOP)
- bytes (input and output) = bs * out * 2 + bs * out * 2 ~= 2 * 10**6 (MB)

=> **#ops / #bytes = 0.25** => на V100 операция ограничена памятью

Performance Limitations

Попробуем взять тот же самый полносвязанный слой и оценить границу когда операция становится ограничена памятью: in = 4096, out = 1024

- #ops / #bytes = 40
- 40 = (2 * x * 1024 * 4096) / (1024 * 4096 * 2 + x * 2 * (1024 + 4096)
- x = bs ~= 42

Часть 2: TensorRT

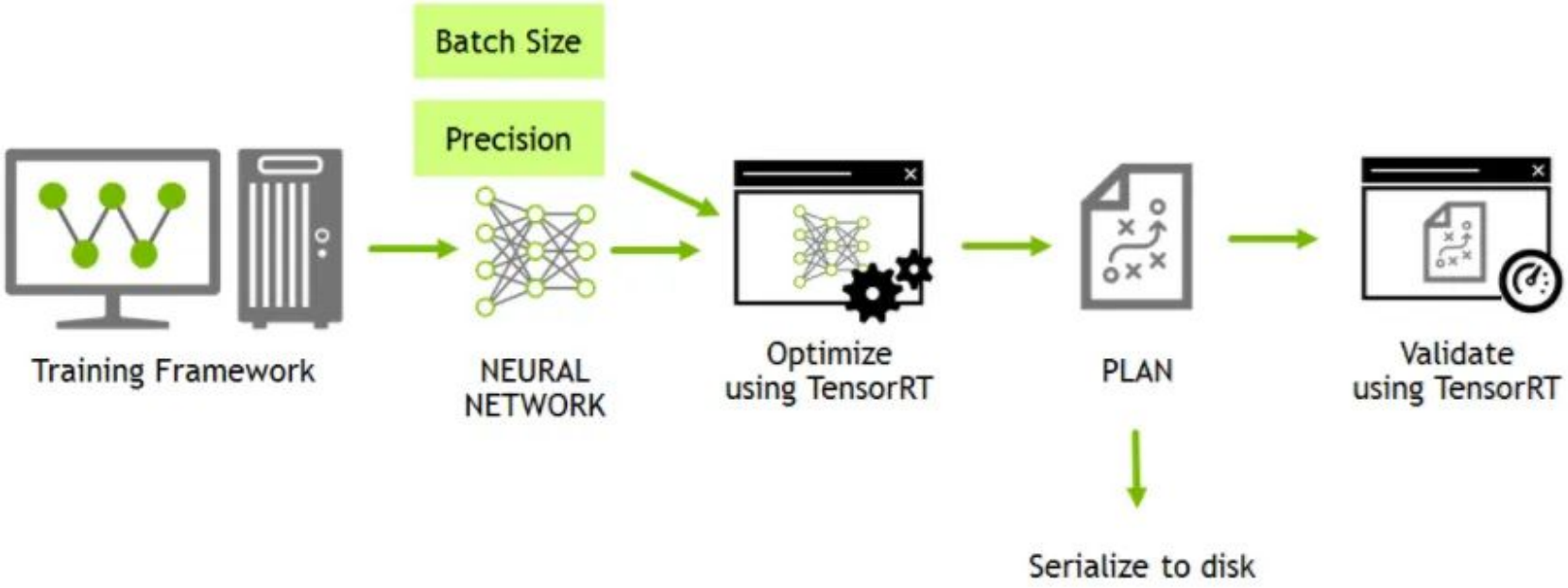
girafe ai

02

Train vs Infer

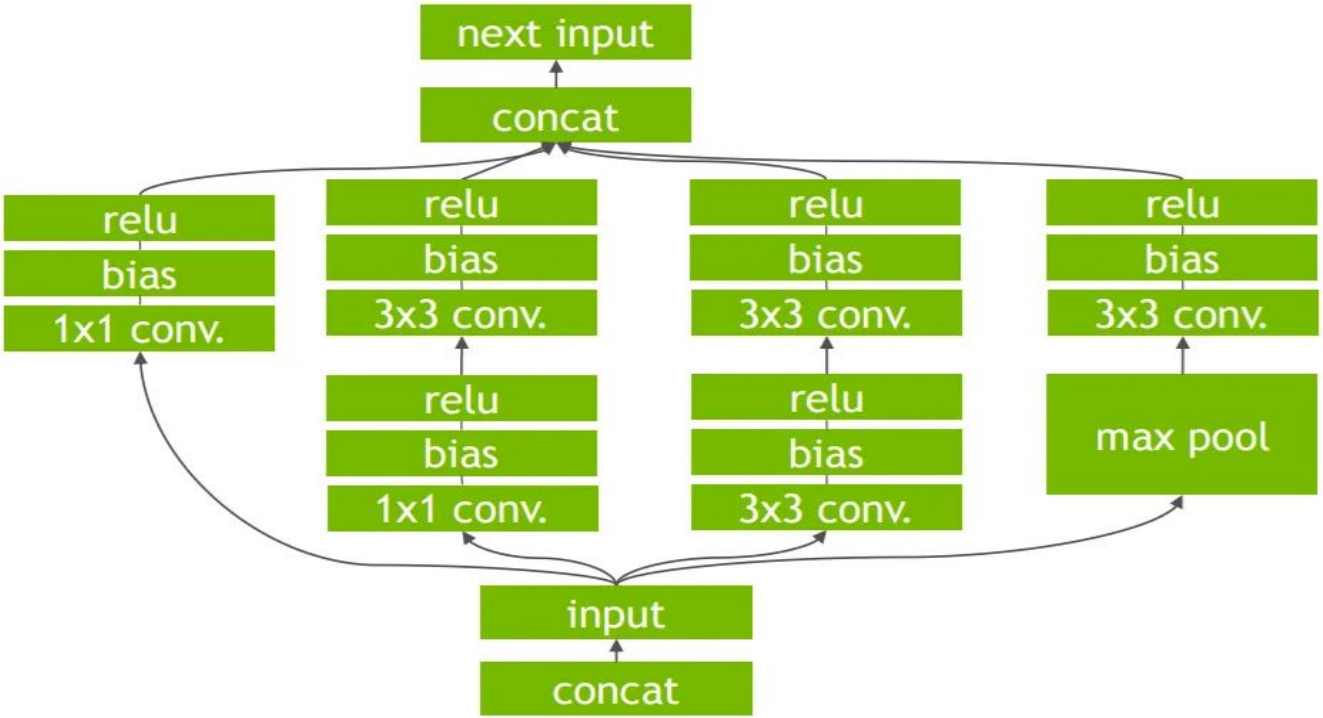
- Железо
  - Много моделей деплоится на CPU
  - Выгодно учить на DGX с A100/V100 с SXM а не PCI-E
    - Но DGX-ы супер прожорливые, особенно в простое
      - 80W idle против 30W idle
    - Обычно не инферим с большим батчем, следовательно не нужно много прожорливой видеопамяти
- Тренировка - оффлайн, инференс - реалтайм
- Нужна высокопроизводительная инфра для обработки запросов
- Как минимум мы не считаем градиенты в инференсе
  - Хотя знаю одно исключение (grad-cam)
- Экономим электричество при инференсе
  - Электричество дорогое
  - В ДЦ вас могут рассчитывать по пику электропотребления
  - У видеокарт нулевой КПД => все электричество переходит в тепло, тепло сложно рассеивать
- Невероятно замороченная оптимизация для деплоя, так как даже 1% к метрике производительности на большом объеме данных может дать прирост в gpu-cpu-годы или даже столетия
- Большая дисперсия софтины для инеренса. Надо заботиться об отказоустойчивости. Нужны балансировщики нагрузки…
- …..
- …..

TensorRT (aka TRT)



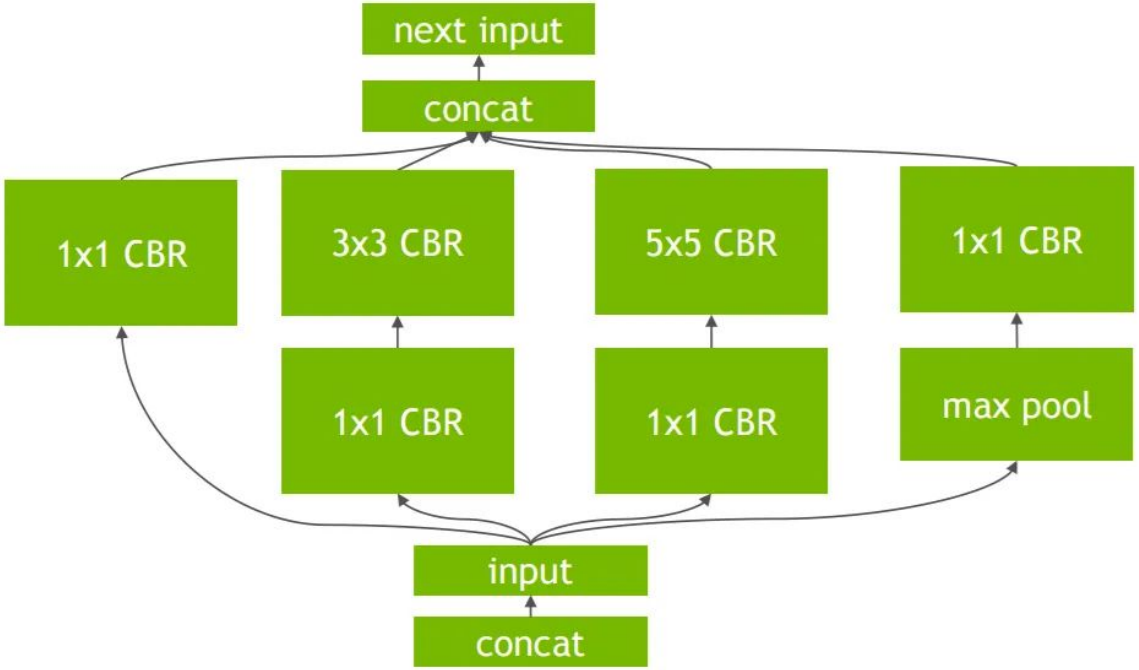
TRT’s Layer Fusion

до

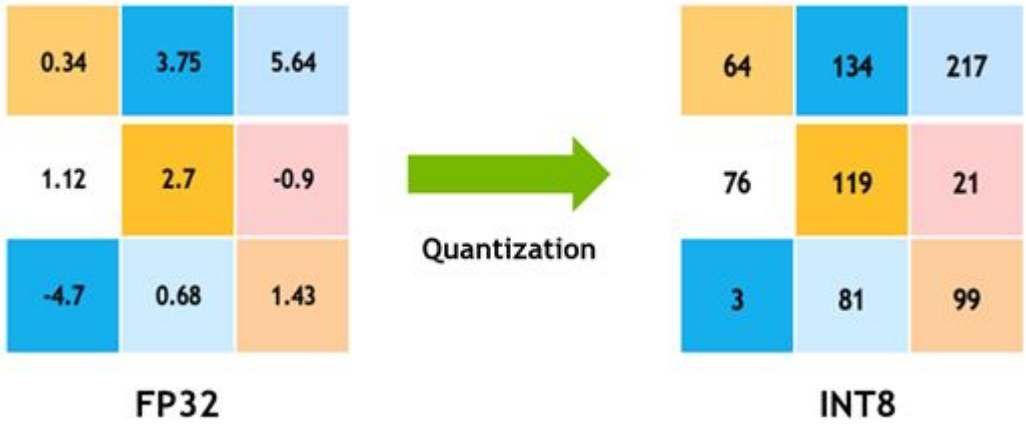


TRT's Layer Fusion

Vertical Fusion



TRT’s Static Quantization

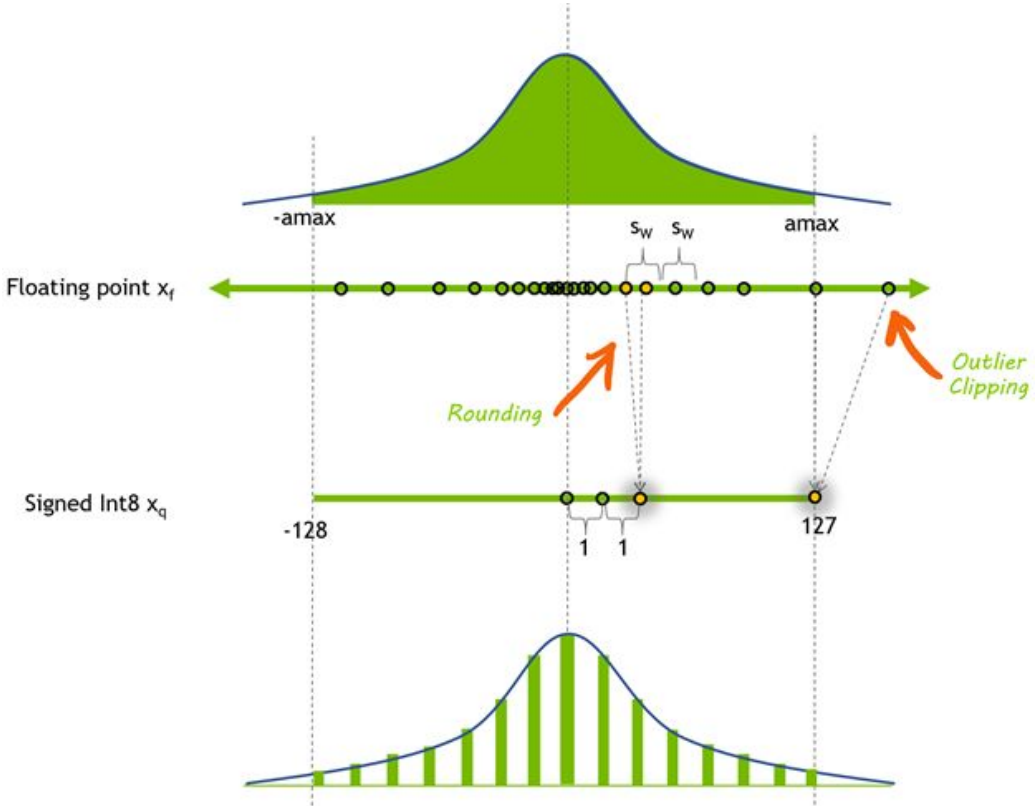


TRT’s Static Quantization

- INT8: [-127, 128]
- x_f: FP32 Tensor
- x_q: INT8 Tensor

Symmetric: $x\_q = Clip(Round(x\_f / scale))$
, где clip - throw outliers (-128,127)

$scale = 2 * amax / 256$



Не работает? QAT!

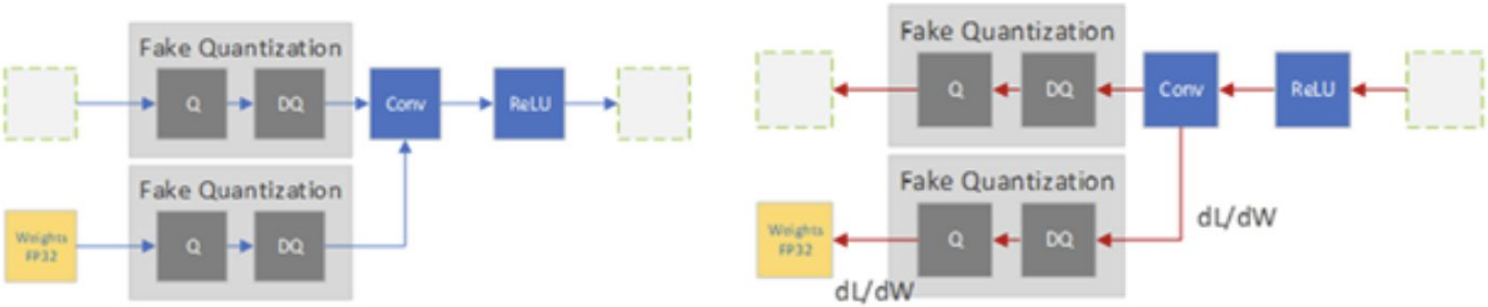


Figure 2. QAT fake-quantization operators in the training forward-pass (left) *and backward-pass* (right)

Доп литература

- Accelerating Sparse Deep NN: https://arxiv.org/pdf/2104.08378.pdf
- Sparsity in INT8 for TRT: https://developer.nvidia.com/blog/sparsity-in-int8-training-workflow-and-best-practices-for-tensorrt-acceleration/
- Q/DQ Layer-Placement Reccomendations: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#qdq-placement-recs
- Dynamic shapes in TRT: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#work_dynamic_shapes
- Custom Layers in TRT: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#extending
- Deep Dive in TRT: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#work

Доп литература

- TensorRT-LLM: https://github.com/NVIDIA/TensorRT-LLM
- ONNX-Docs: https://onnx.ai/onnx/

Спасибо за внимание!

Жду вопросов и обсуждений