Deep Learning Optimisé - Jean Zay

Best Practices and State Of The Art

cnrs

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

Fast.ai tips and engineering

“An AI speed test shows clever coders can still beat tech giants like Google and Intel.” DAWNBench competition 2018

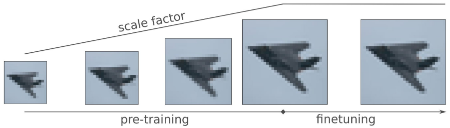
👏 OneCycle lr scheduler
+ lr finder

Popularizes the works of [Leslie N. Smith]

Thanks to Global Average Pooling

Test Rectangular Validation Technique

Progressive image resizing



Dynamic batch size

ML Perf - Reference for AI Supercomputing

MLPerf

Fair and useful benchmarks for measuring training and inference performance of ML hardware, software, and services.

Industry standard

- Hardware
- Framework
- SOTA

ML Perf - Benchmarks

Training

Speech Recognition
RNN-T

NLP
BERT

Recommender
DLRM

Reinforcement Learning
MiniGo

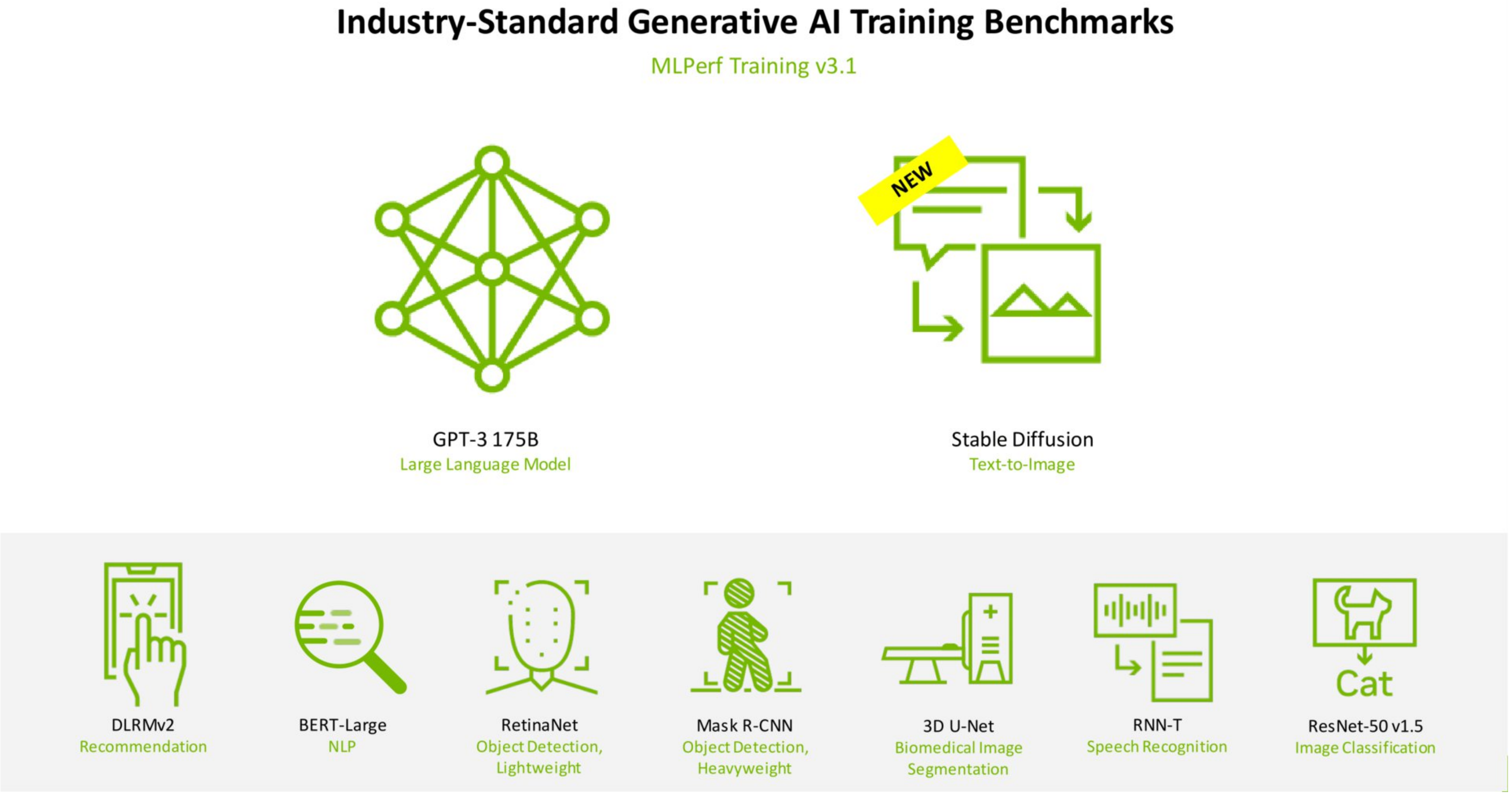
Biomedical Image Segmentation
UNet-3D

Object Detection (Light weight)
SSD

Object Detection (Heavy weight)
Mask R-CNN

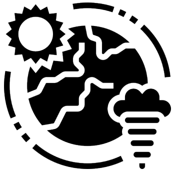
Image Classification
ResNet-50 v1.5

ML Perf - Benchmarks



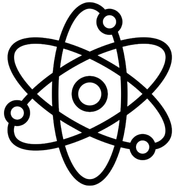
ML Perf - Benchmarks

Training HPC



Climate segmentation
DeepCAM

Cosmology parameter prediction
CosmoFlow



Quantum molecular modeling
DimNet++

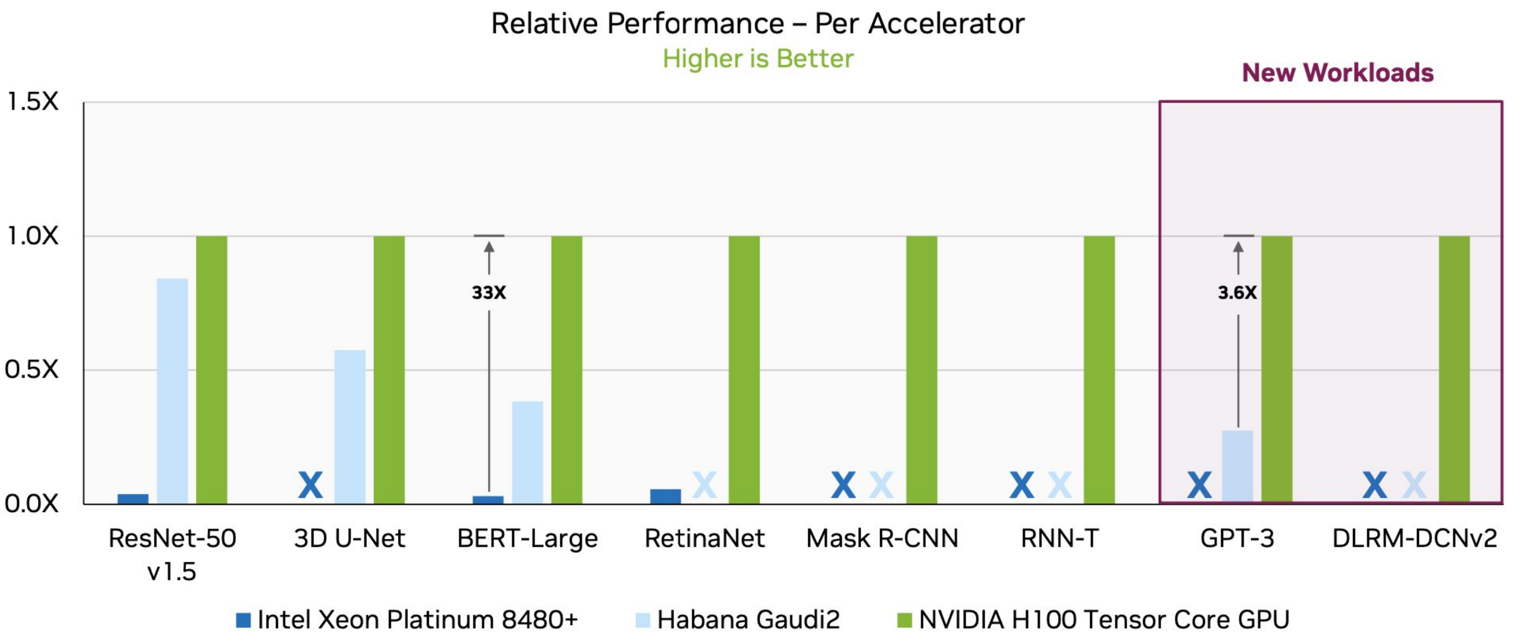
Inference :

- Datacenter
- Edge
- Mobile
- Tiny

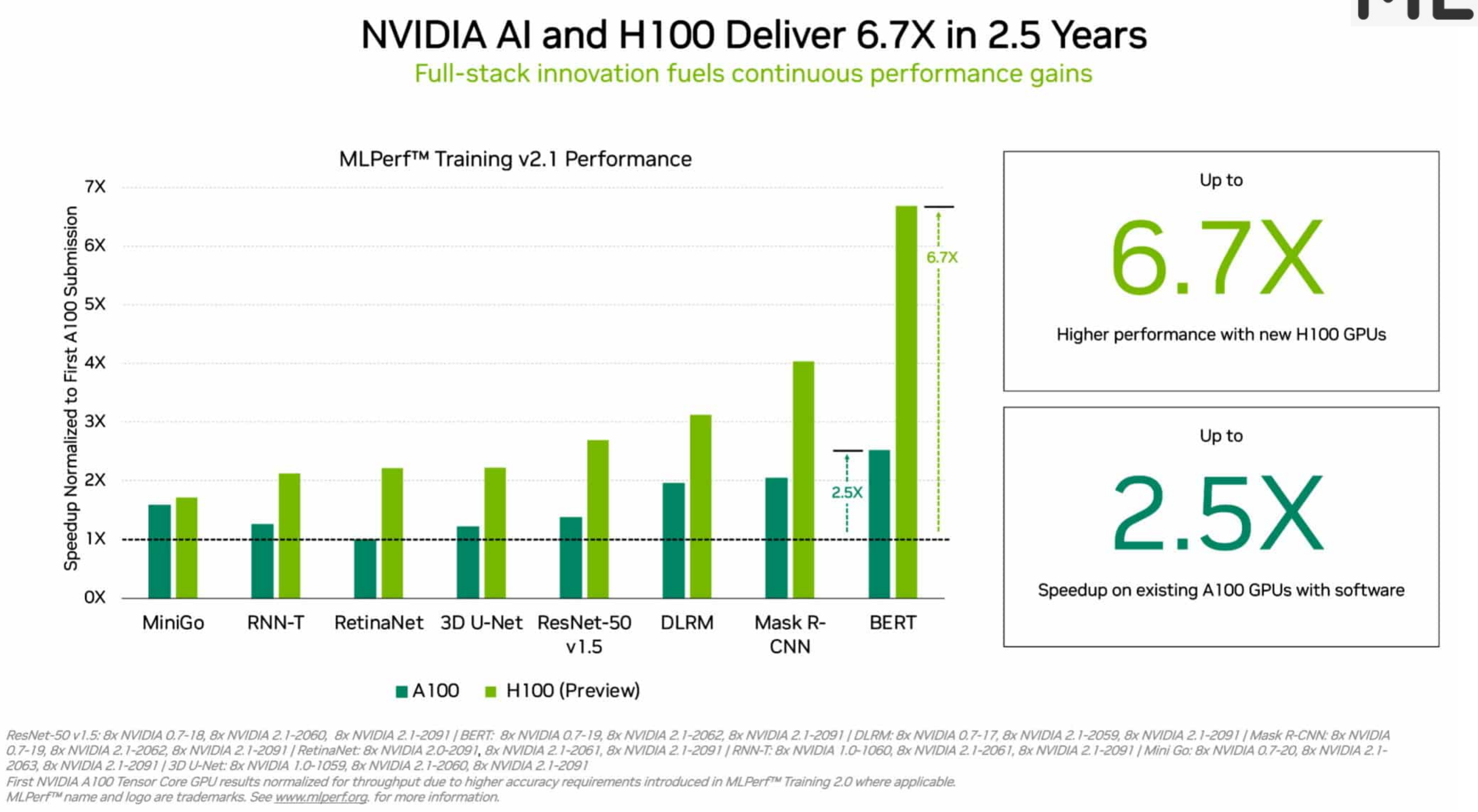
ML Perf - Leadership

NVIDIA H100 GPU Extends AI Training Leadership

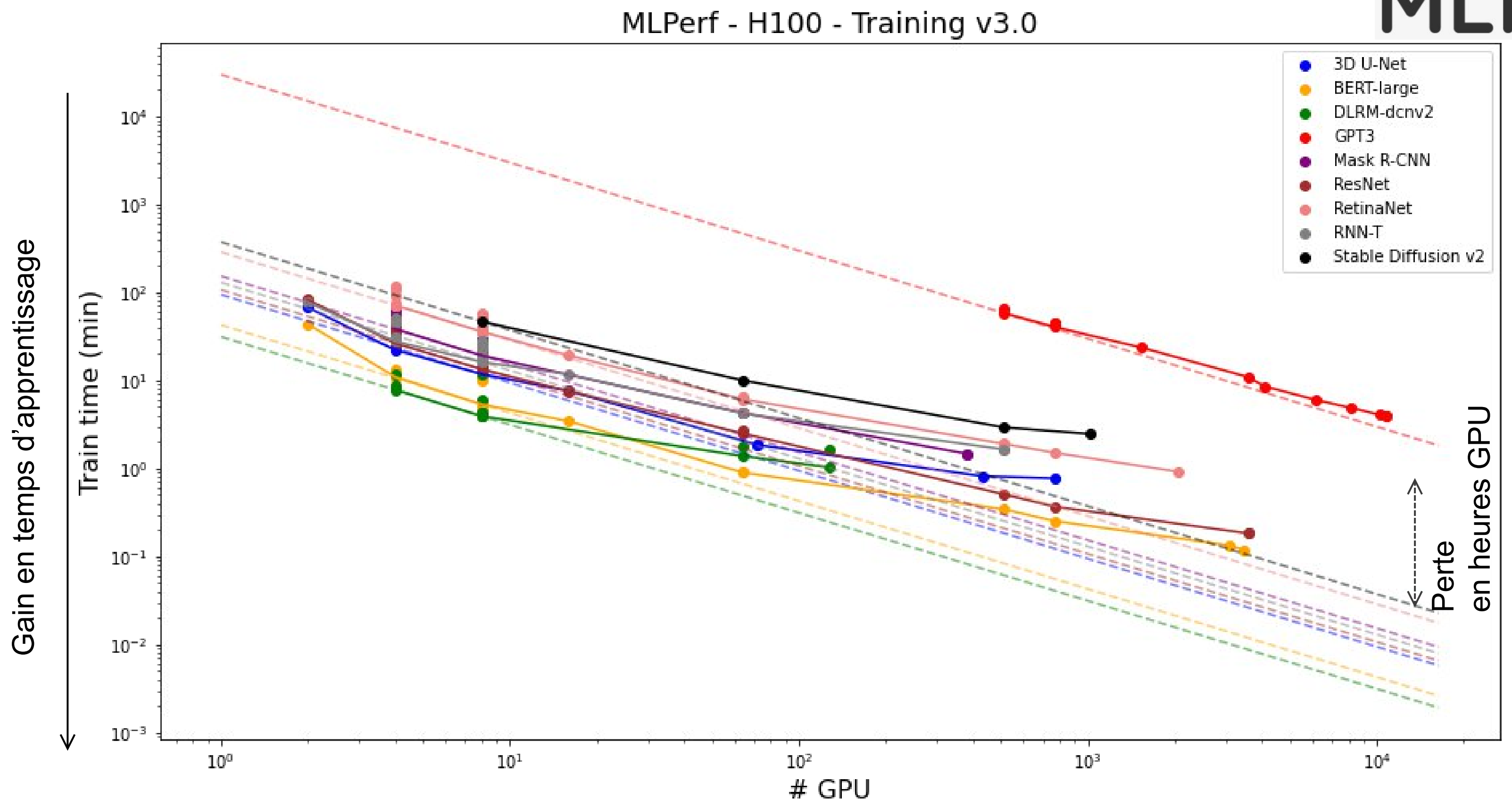
Fastest and most versatile AI accelerator



ML Perf - Evolution



ML Perf - Scaling



AlgoPerf: Training Algorithms benchmark

Speeding up training requires new competitive benchmarks to detect robust improvment :

Hyperparameter	AdamW	NadamW	Heavy Ball	Nesterov
Base LR	Log [1e−4 ,1e−2]	Log [1e−4 ,1e−2]	Log [1e−1 ,10]	Log [1e−1 ,10]
Weight decay	Log [5e−3 ,1]	Log [5e−3 ,1]	Log [1e−7 ,1e−5]	Log [1e−7 ,1e−5]
1 - β_1[Default]	0.1	0.1	0.1	0.1
1 - β_1[Tuned]	Log [2e−2 ,0.5]	Log [4e−3 ,0.1]	Log [5e−3 ,0.3]	Log [5e−3 ,0.3]
1 - β_2	0.999	0.999	NA	NA
Schedule	warmup + cosine decay	warmup + cosine decay	warmup + linear decay	warmup + linear decay
Warmup	5%	5%	5%	5%
Decay factor	NA	NA	{1e−2 ,1e−3}	{1e−2 ,1e−3}
Decay steps	NA	NA	0.9	0.9
Label smoothing	{0.1, 0.2}	{0.1, 0.2}	{0.1, 0.2}	{0.1, 0.2}
Dropout (Tied)	{0.0, 0.1}	{0.0, 0.1}	{0.0, 0.1}	{0.0, 0.1}

Hyperparameter	LAMB	Adafactor	SAM(w. Adam)	Distributed Shampoo
Base LR	Log [1e−4 ,1e−2]	Log [1e−4 ,1e−2]	Log [1e−4 ,1e−2]	Log [1e−4 ,1e−2]
Weight decay	Log [1e−3 ,1]	Log [1e−3 ,1]	Log [1e−2 ,0.2]	Log [5e−3 ,1]
1 - β_1[Default]	0.1	0.1	0.1	0.1
1 - β_1[Tuned]	Log [2e−2 ,0.5]	Log [1e−2 ,0.45]	Log [5e−2 ,0.43]	Log [1e−2 ,0.15]
1 - β_2	0.999	0.999	0.999	0.999
ρ	NA	NA	{0.01, 0.02, 0.05}	NA
Schedule	warmup + cosine decay	warmup + cosine decay	warmup + cosine decay	warmup + cosine decay
Warmup	5%	5%	5%	5%
Decay factor	NA	NA	NA	NA
Decay steps	NA	NA	NA	NA
Label smoothing	{0.1, 0.2}	{0.1, 0.2}	{0.1, 0.2}	{0.1, 0.2}
Dropout (Tied)	{0.0, 0.1}	{0.0, 0.1}	{0.0, 0.1}	{0.0, 0.1}

- **Activation function:** Most base workloads employed ReLU as the activation function and we explored alternative activation functions such as GELU, SiLU, or TanH.
- **Pre-LN vs Post-LN:** For Transformer-based models, the base workload was usually the PRE-LAYER NORM (PRE-LN) (Xiong et al., 2020) version. We changed these to POST-LAYER NORM (POST-LN, see Figure 3).
- **Attention temperature:** For the WMT TRANSFORMER, we modified the attention layers to compute Softmax $\left(\frac{cXW^Q(XW^K)^\top}{\sqrt{D/H}}\right)$ where c is a constant scalar denoting the attention temperature. The default self-attention implementation sets $c = 1$. In order to artificially induce instabilities similar to those faced by larger versions of these models, we set $c = 1.6$.

Initialization scales: For the DLRM model, changing the scale of the initial weights of the embedding layer resulted in a variant. For the RESNET model, changing the initial batch normalization layer scale weights resulted in a workload variant.

Normalization layer: We changed the type of normalization layer employed in the model. Common changes included interchanging batch normalization with layer normalization, as well as instance normalization with layer normalization.

Width, depth, and channels: We explored changing model width, depth, and number of channels, as applicable.

Input pipeline: On LIBRISPEECH, we found changing SPECAUGMENT strength to be an effective strategy.

Residual connection structure and scaling: For the DLRM model, we created a variant with additional residual connections. For DEEPSPEECH, we removed residual connections from the model.

Pooling layer type: Changing the pooling layer type from *global average pooling* to *max average pooling* resulted in a variant for the VIT workload.

DataPerf – Data-Centric benchmark

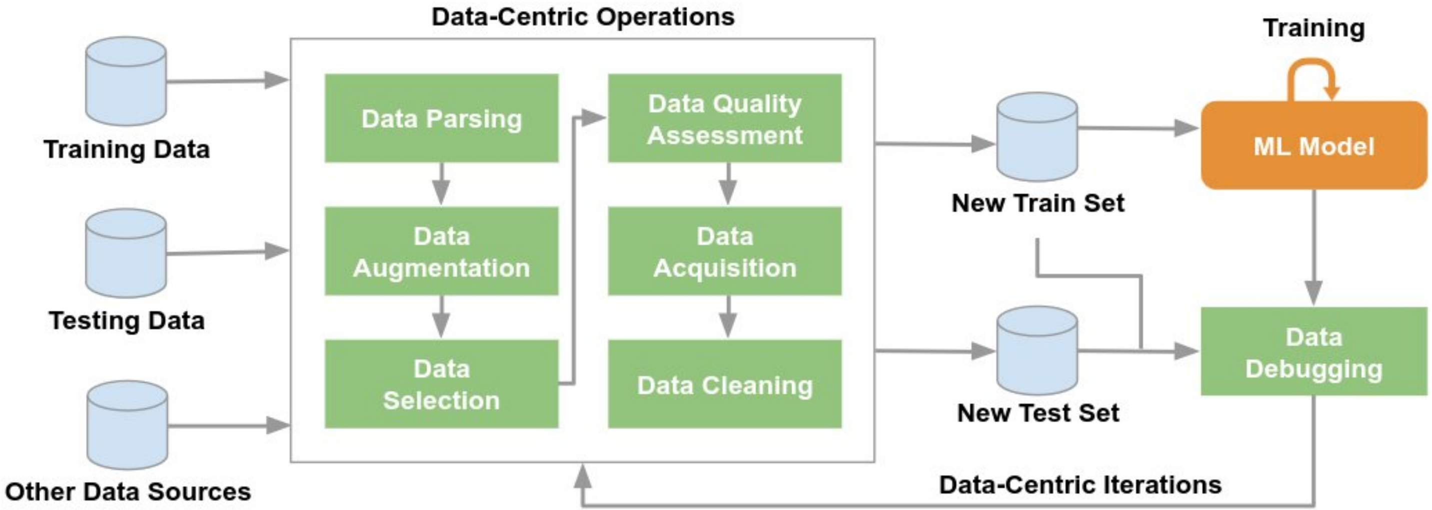


Figure 1: Typical benchmarks are model-centric, and therefore focus on the model design and training stages of the ML pipeline (shown in orange). However, to develop high-quality ML applications, users often employ a collection of data-centric operations to improve data quality and repeated data-centric iterations to refine these operations. DataPerf aims to benchmark all major stages of such a data-centric pipeline (shown in green) to improve ML data quality.

Performance Tuning Guide

PyTorch

- Enable asynchronous data loading and augmentation

```
torch.utils.data.DataLoader
num_workers > 0
pin_memory=True
```

- Disable gradient calculation for validation or inference

```
with torch.no_grad():
    val_outputs = model(val_images)
    val_loss = criterion(val_outputs, val_labels)
```

- Use mixed precision and AMP

```
from torch.cuda.amp import autocast, GradScaler
with autocast():
```

- Use efficient data-parallel backend

```
torch.nn.parallel.DistributedDataParallel
```

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Performance Tuning Guide

PyTorch

- Disable bias for convolutions directly followed by a batch norm

```
nn.Conv2d(..., bias=False, ....)
Models available from torchvision already
implement this optimization.
```

- Enable channels_last memory format for computer vision models

```
x = x.to(memory_format=torch.channels_last)
```

- Disable debugging APIs

```
anomaly detection: torch.autograd.detect_anomaly or torch.autograd.set_detect_anomaly(True)
profiler related: torch.autograd.profiler.emit_nvtx, torch.autograd.profiler.profile
autograd gradcheck: torch.autograd.gradcheck or torch.autograd.gradgradcheck
```

- Create tensors directly on the target device

```
torch.rand(size).cuda()
torch.rand(size, device='cuda')
```

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Performance Tuning Guide

PyTorch

- Fuse pointwise operations
  Pointwise operations (elementwise addition, multiplication, math functions - sin(), cos(), sigmoid() etc.) can be fused into a single kernel to amortize memory access time and kernel launch time. PyTorch JIT can fuse kernels automatically.

```
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

- Enable cuDNN auto-tuner
  For convolutional networks

```
torch.backends.cudnn.benchmark = True
```

- Avoid unnecessary CPU-GPU synchronization

```
print(cuda_tensor)
cuda_tensor.item()
memory copies: tensor.cuda(), cuda_tensor.cpu() and equivalent tensor.to(device) calls
cuda_tensor.nonzero()
python control flow e.g. if (cuda_tensor != 0).all()
```

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Performance Tuning Guide

PyTorch

- ## Load-balance workload in a distributed setting
  The core idea is to distribute workload over all workers as uniformly as possible within each global batch. For example Transformer solves imbalance by forming batches with approximately constant number of tokens (and variable number of sequences in a batch), other models solve imbalance by bucketing samples with similar sequence length or even by sorting dataset by sequence length.

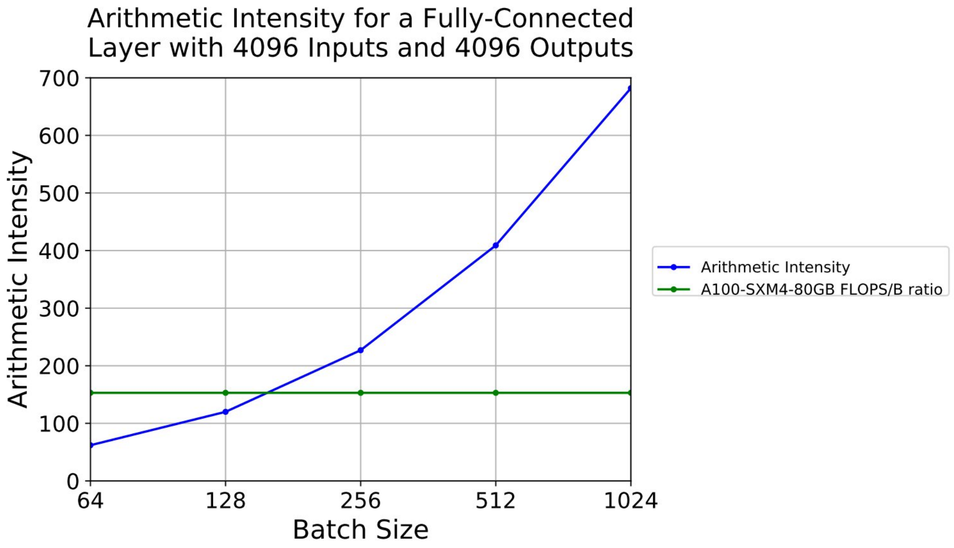
- ## Preallocate memory in case of variable input length
  For Speech Recognition or NLP, preexecute a forward and a backward pass with a generated batch of inputs with maximum sequence length (either corresponding to max length in the training dataset or to some predefined threshold). This step preallocates buffers of maximum size, which can be reused in subsequent training iterations.

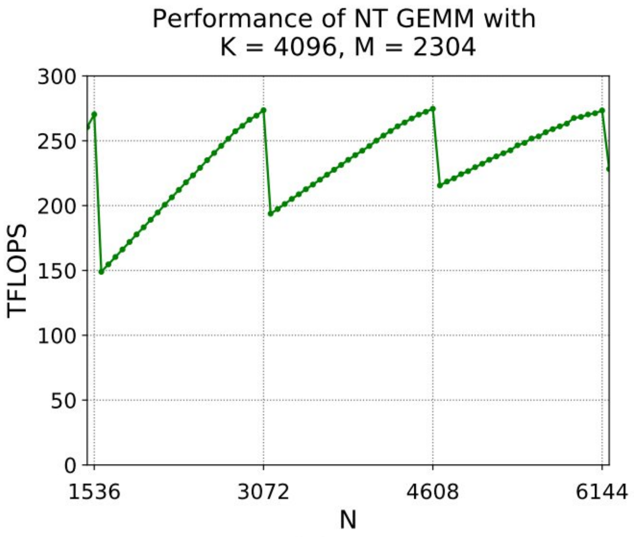
- ## Match the order of layers in constructors and during the execution if using DistributedDataParallel``(find_unused_parameters=True)
  To maximize the amount of overlap, the order in model constructors should roughly match the order during the execution. If the order doesn't match, then all-reduce for the entire bucket waits for the gradient which is the last to arrive.
  With find_unused_parameters=False it's not necessary to reorder layers or parameters to achieve optimal performance.

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Deep Learning Performance Documentation

$$\text{Arithmetic Intensity} = \frac{\text{number of FLOPS}}{\text{number of byte accesses}} = \frac{2 \cdot (M \cdot N \cdot K)}{2 \cdot (M \cdot K + N \cdot K + M \cdot N)} = \frac{M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}$$





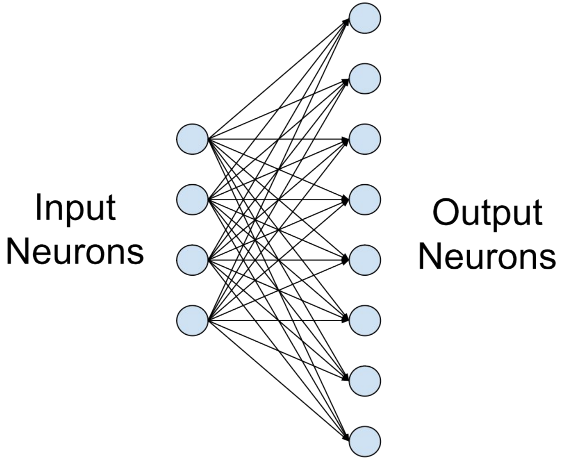
Wave Quantization effect

https://docs.nvidia.com/deeplearning/performance/index.html

Linear/Fully-Connected Layers User's Guide

The following quick start checklist provides specific tips for **fully-connected layers**.

- Choose the batch size and the number of inputs and outputs to be **divisible by 4 (TF32) / 8 (FP16) / 16 (INT8)** to run efficiently on **Tensor Cores**. For best efficiency on **A100**, choose these parameters to be **divisible by 32 (TF32) / 64 (FP16) / 128 (INT8)** .
- Especially when ones are small, choosing the batch size and the number of inputs and outputs to be **divisible by at least 64** and **ideally 256** can streamline tiling and reduce overhead.
- **Larger values** for batch size and the number of inputs and outputs **improve** parallelization and efficiency.
- As a rough guideline, choose batch sizes and neuron counts **greater than 128** to avoid being limited by memory bandwidth.

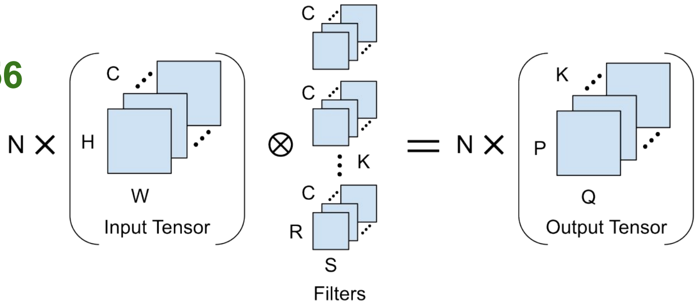


https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html

Convolutional Layers User's Guide

The following quick start checklist provides specific tips for **convolutional layers**.

- Choose the number of **input and output channels** to be divisible **by 8 (for FP16) or 4 (for TF32)** to run efficiently on **Tensor Cores**. For the **first convolutional layer** in most CNNs with **3-channel** images, **padding to 4 channels** is sufficient if a stride of 2 is used.
- Choose parameters to be divisible by **at least 64** and **ideally 256** to enable efficient tiling and reduce overhead.
- **Larger values** for size-related parameters can improve parallelization.
- When the **size of the input is the same** in each iteration, **autotuning** is an efficient method to ensure the selection of the ideal algorithm for each convolution in the network. `torch.backends.cudnn.benchmark = True`.
- Choose tensor layouts in memory to avoid transposing input and output data. We recommend using the **NHWC format** where possible.

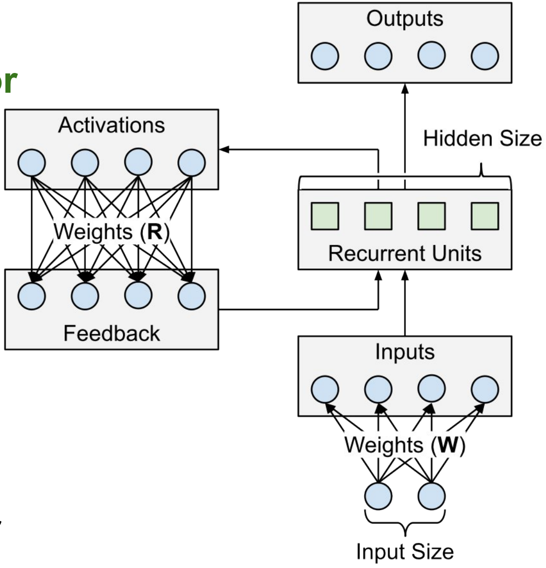


https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html

Recurrent Layers User's Guide

The following quick start checklist provides specific tips for **recurrent layers**.

- **Recurrent operations can be parallelized**. We recommend using NVIDIA® cuDNN implementations, which do this automatically.
- When using the **standard implementation**, minibatch size and hidden sizes should be:
  - **Divisible by 8 (for FP16)** or **4 (for TF32)** to run efficiently on **Tensor Cores**.
  - **Divisible by at least 64** and **ideally 256** to improve tiling efficiency.
  - **Greater than 128 (minibatch size)** or **256 (hidden sizes)** to be limited by computation rate rather than memory bandwidth.
- When using the **persistent implementation** (available for FP16 data only):
  - **Hidden sizes** should be **divisible by 32** to run efficiently on Tensor Cores. Better tiling efficiency may be achieved **by larger multiples of 2, up to 256**.
  - **Minibatch size** should be **divisible by 8** to run efficiently on Tensor Cores...
- **Try increasing parameters for better efficiency**.



https://docs.nvidia.com/deeplearning/performance/dl-performance-recurrent/index.html

Memory-Limited Layers User's Guide

The following quick start checklist provides specific tips **for layers whose performance is limited by memory accesses (Batch Normalization, Activations, Pooling, ...)**.

- Explore the available implementations of each layer in the **NVIDIA cuDNN API** Reference or your framework. Often the best way to improve performance is to choose **a more efficient implementation**.
- **Be aware of the number of memory accesses** required for each layer. Performance of a memory-bound calculation is simply based on the number of inputs, outputs, and weights that need to be loaded and/or stored per pass. We don't have recommended parameter tweaks for these layers.
- **Be aware of the impact of each layer** on the overall training step performance. **Memory-bound layers** are most likely to take a significant amount of time in small networks where there are no large and computation-heavy layers to dominate performance.

https://docs.nvidia.com/deeplearning/performance/dl-performance-memory-limited/index.html

Hugging Face

Efficient Training on Multiple GPUs

If training a model on a single GPU is too slow or if the model's weights do not fit in a single GPU's memory, transitioning to a multi-GPU setup may be a viable option. Prior to making this transition, thoroughly explore all the strategies covered in the Methods and tools for efficient training on a single GPU as they are universally applicable to model training on any number of GPUs. Once you have employed those strategies and found them insufficient for your case on a single GPU, consider moving to multiple GPUs.

Transitioning from a single GPU to multiple GPUs requires the introduction of some form of parallelism, as the workload must be distributed across the resources. Multiple techniques can be employed to achieve parallelism, such as data parallelism, tensor parallelism, and pipeline parallelism. It's important to note that there isn't a one-size-fits-all solution, and the optimal settings depend on the specific hardware configuration you are using.

This guide offers an in-depth overview of individual types of parallelism, as well as guidance on ways to combine techniques and choosing an appropriate approach. For step-by-step tutorials on distributed training, please refer to the 🤗 Accelerate documentation.

> While the main concepts discussed in this guide are likely applicable across frameworks, here we focus on PyTorch-based implementations.

Before diving deeper into the specifics of each technique, let's go over the rough decision process when training large models on a large infrastructure.

Scalability strategy

Begin by estimating how much vRAM is required to train your model. For models hosted on the 🤗 Hub, use our Model Memory