



# Deep Learning Optimisé - Jean Zay

---

Good Practice and State Of The Art




INSTITUT DU  
DÉVELOPPEMENT ET DES  
RESSOURCES EN  
INFORMATIQUE  
SCIENTIFIQUE



# Fast.ai tips and engineering

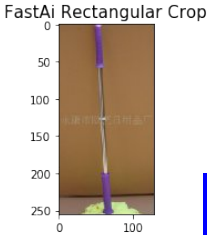


“An AI speed test shows clever coders can still beat tech giants like Google and Intel.” DAWNBench competition 2018

 OneCycle lr scheduler  
+ lr finder

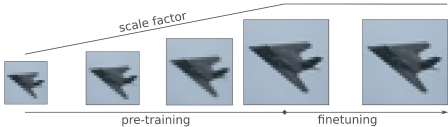
➔ Popularizes the works of  
[Leslie N. Smith](#)

Thanks to Global Average Pooling



**Test Rectangular  
Validation Technique**

**Progressive image  
resizing**



**Dynamic batch size**



# ML Perf - Référence pour le Supercomputing en IA



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

Industry standard

- Hardware
- Framework
- SOTA



## 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

## Industry-Standard Generative AI Training Benchmarks

MLPerf Training v3.1



GPT-3 175B  
Large Language Model



Stable Diffusion  
Text-to-Image



DLRMV2  
Recommendation



BERT-Large  
NLP



RetinaNet  
Object Detection,  
Lightweight



Mask R-CNN  
Object Detection,  
Heavyweight



3D U-Net  
Biomedical Image  
Segmentation



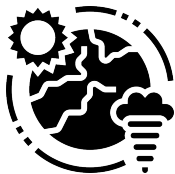
RNN-T  
Speech Recognition



Cat  
ResNet-50 v1.5  
Image Classification



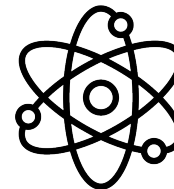
## Training HPC



Climate segmentation  
**DeepCAM**



Cosmology parameter prediction  
**CosmoFlow**



Quantum molecular modeling  
**DimNet++**

## Inference :

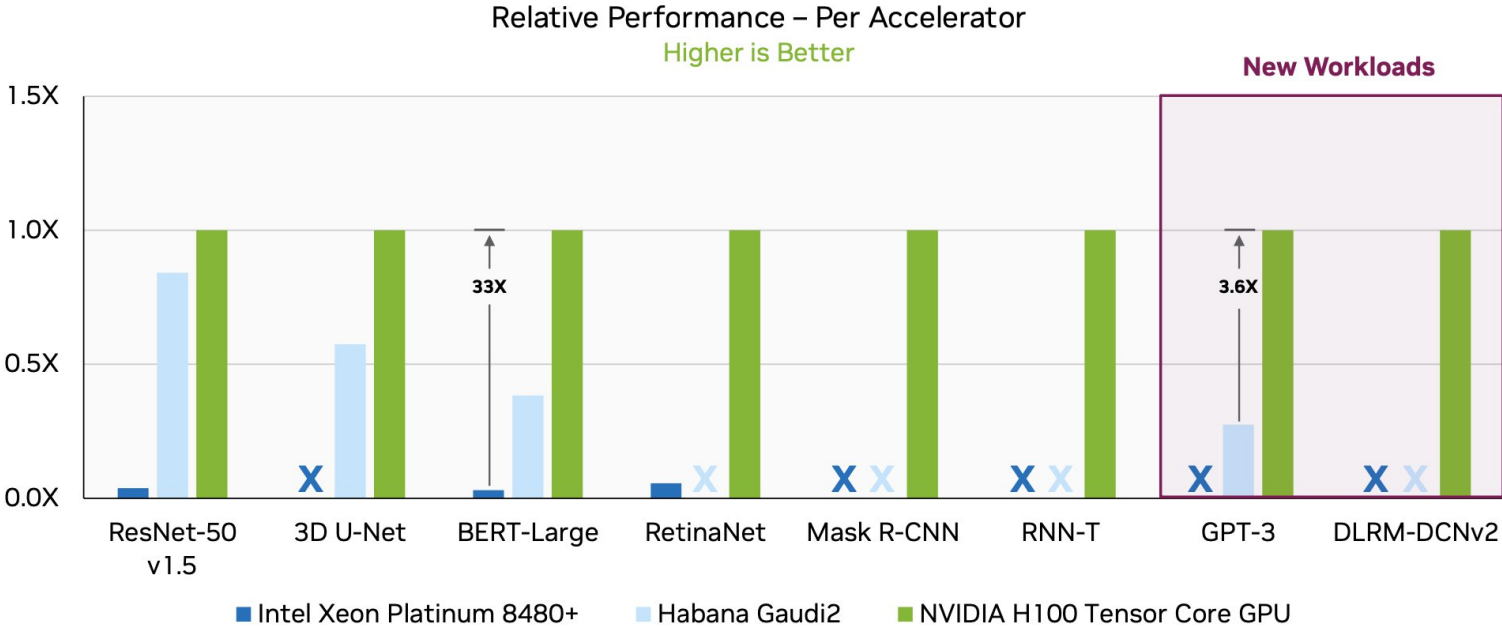


- Datacenter
- Edge
- Mobile
- Tiny



## NVIDIA H100 GPU Extends AI Training Leadership

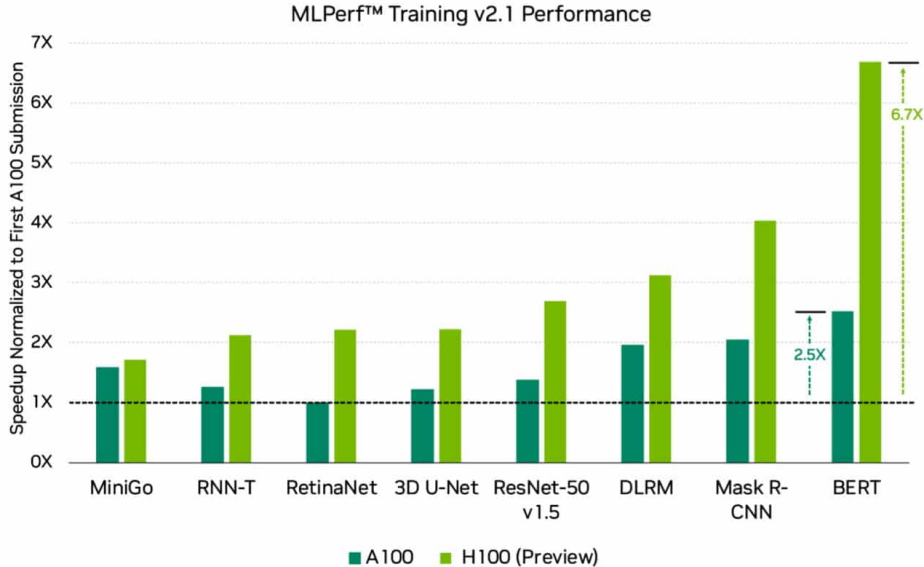
Fastest and most versatile AI accelerator





## NVIDIA AI and H100 Deliver 6.7X in 2.5 Years

Full-stack innovation fuels continuous performance gains



Up to  
**6.7X**  
Higher performance with new H100 GPUs

Up to  
**2.5X**  
Speedup on existing A100 GPUs with software

ResNet-50 v1.5: 8x NVIDIA 0.7-18, 8x NVIDIA 2.1-2060, 8x NVIDIA 2.1-2091 | BERT: 8x NVIDIA 0.7-19, 8x NVIDIA 2.1-2062, 8x NVIDIA 2.1-2091 | DLRM: 8x NVIDIA 0.7-17, 8x NVIDIA 2.1-2059, 8x NVIDIA 2.1-2091 | Mask R-CNN: 8x NVIDIA 0.7-19, 8x NVIDIA 2.1-2062, 8x NVIDIA 2.1-2091 | RetinaNet: 8x NVIDIA 2.0-2091, 8x NVIDIA 2.1-2061, 8x NVIDIA 2.1-2091 | RNN-T: 8x NVIDIA 1.0-1060, 8x NVIDIA 2.1-2061, 8x NVIDIA 2.1-2091 | Mini Go: 8x NVIDIA 0.7-20, 8x NVIDIA 2.1-2063, 8x NVIDIA 2.1-2091 | 3D U-Net: 8x NVIDIA 1.0-1059, 8x NVIDIA 2.1-2060, 8x NVIDIA 2.1-2091

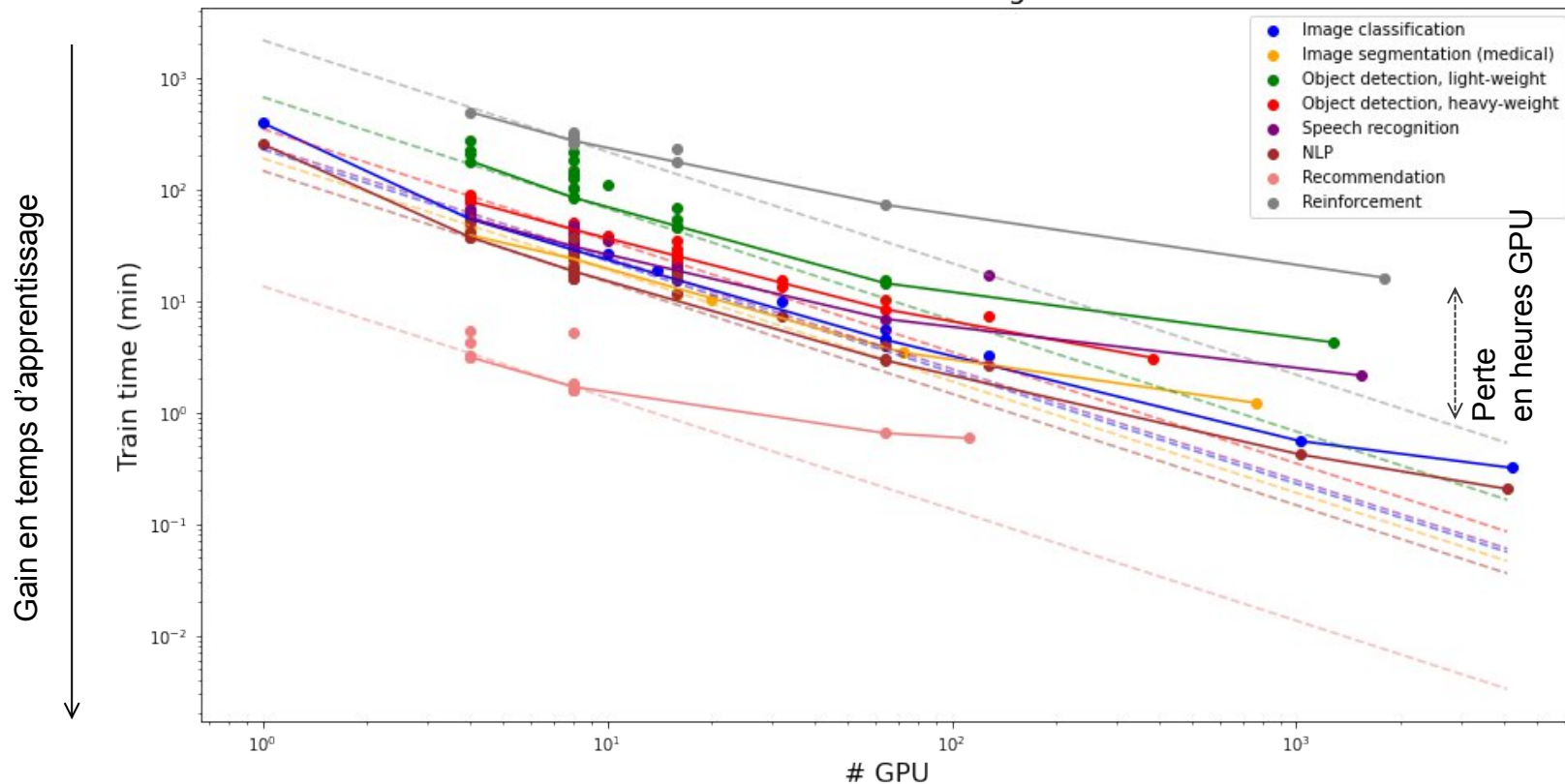
First NVIDIA A100 Tensor Core GPU results normalized for throughput due to higher accuracy requirements introduced in MLPerf™ Training 2.0 where applicable.  
MLPerf™ name and logo are trademarks. See [www.mlperf.org](https://www.mlperf.org) for more information.



# ML Perf - Scaling



MLPerf - A100 - Training v2.0



- 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
```

- 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')
```

- 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()
```

- 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

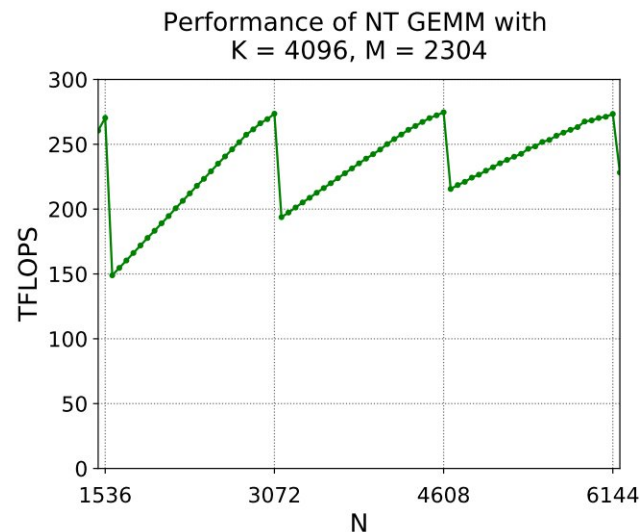
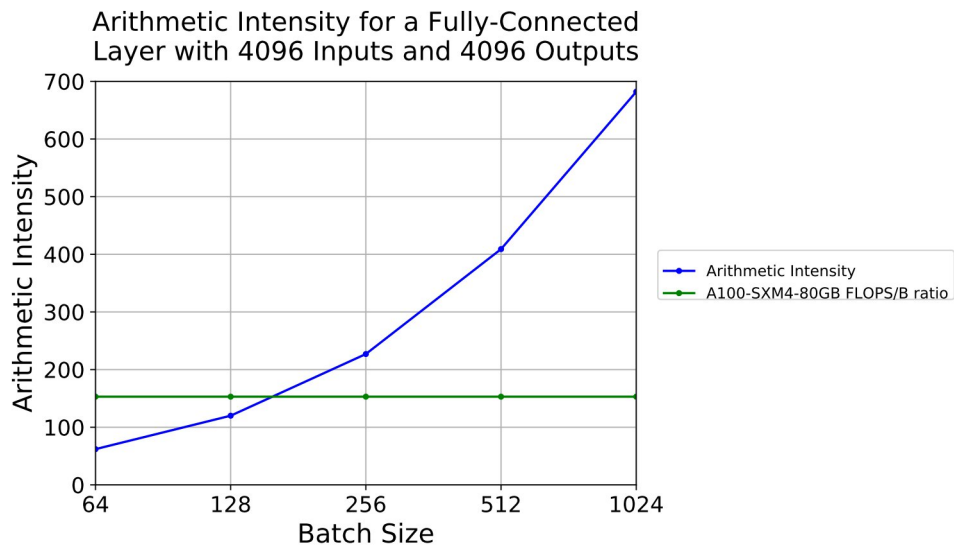
For Speech Recognition or NLP, **preexecute** a forward and a backward pass with a **generated batch of inputs with maximum sequence length** (either corresponding to max length in the training dataset or to some predefined threshold). This step **preallocates buffers** of maximum size, which can be 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.

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



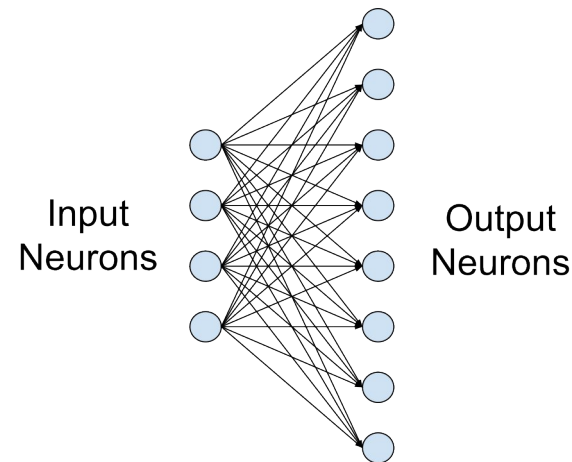
Wave Quantization effect

# 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.

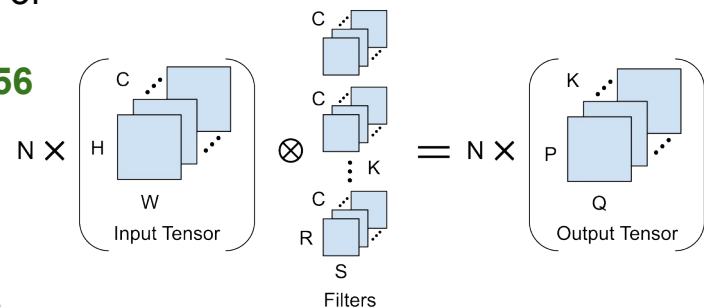


# Convolutional Layers User's Guide



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.



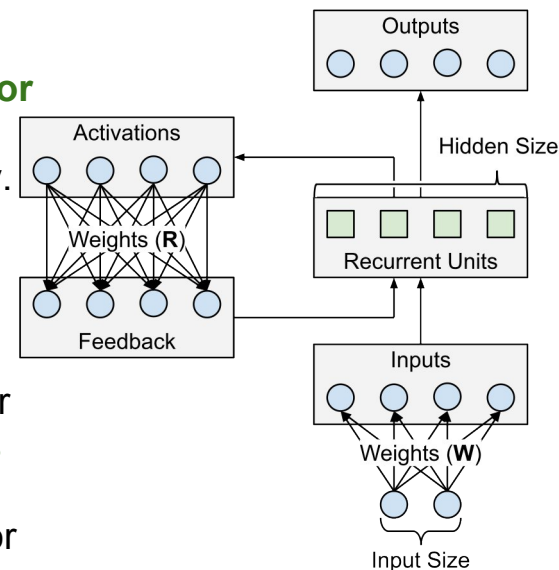


# Recurrent Layers User's Guide



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.**

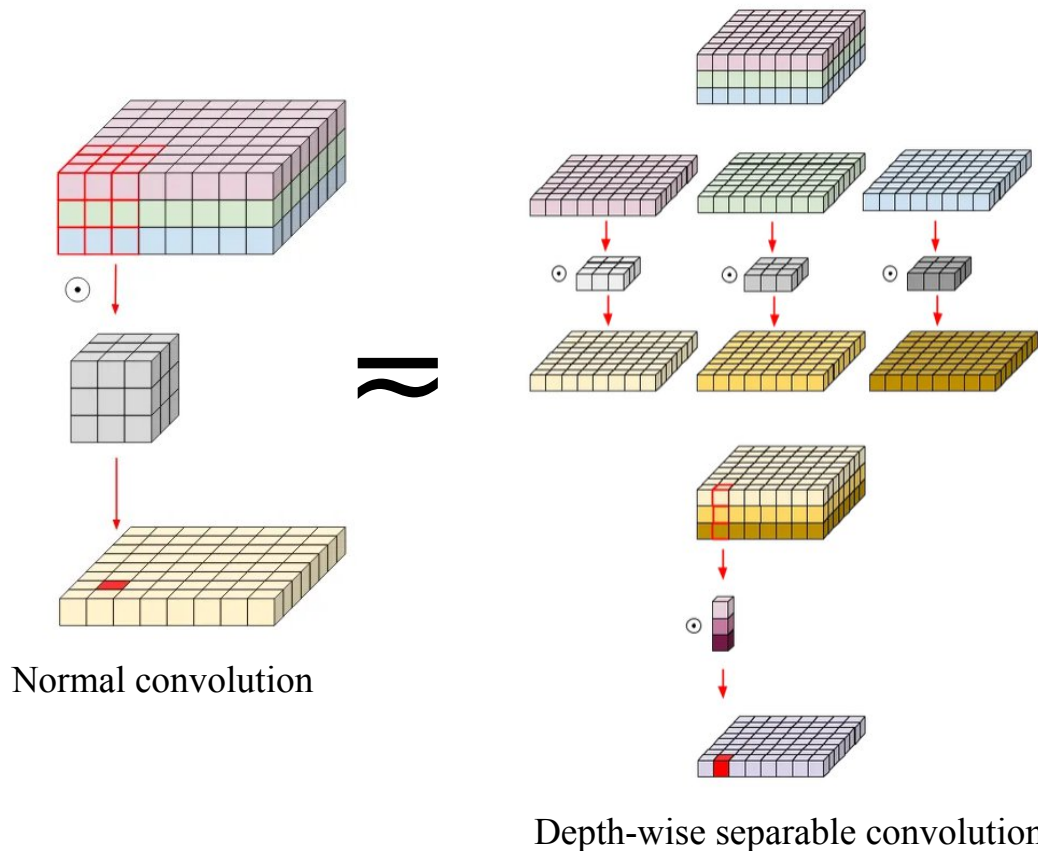




The following quick start checklist provides specific tips **for layers whose performance is limited by memory accesses (Batch Normalization, Activations, Pooling, ...)**.

- 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.

# Memory-Limited Layers Example



MobileNet  
EfficientNet



**Decreasing of  
Arithmetic Intensity.**

**GPU unadapted !!**



## The AI community building the future.

Build, train and deploy state of the art models powered by the reference open source in machine learning.



104,839

### • Hub

Host Git-based models, datasets and Spaces on the Hugging Face Hub.

### • Hub Python Library

Client library for the HF Hub: manage repositories from your Python runtime.

### • Inference API

Use more than 50k models through our public inference API, with scalability built-in.

### • Accelerate

Easily train and use PyTorch models with multi-GPU, TPU, mixed-precision.

### • Tokenizers

Fast tokenizers, optimized for both research and production.

### • Datasets-server

API to access the contents, metadata and basic statistics of all Hugging Face Hub datasets.

### • timm

State-of-the-art computer vision models, layers, optimizers, training/evaluation, and utilities.

### • Transformers

State-of-the-art ML for PyTorch, TensorFlow, and JAX.

### • Datasets

Access and share datasets for computer vision, audio, and NLP tasks.

### • Huggingface.js

A collection of JS libraries to interact with Hugging Face, with TS types included.

### • Inference Endpoints

Easily deploy your model to production on dedicated, fully managed infrastructure.

### • Optimum

Fast training and inference of HF Transformers with easy to use hardware optimization tools.

### • Evaluate

Evaluate and report model performance easier and more standardized.

### • Simulate

Create and share simulation environments for intelligent agents and synthetic data generation.

### • Safetensors

Simple, safe way to store and distribute neural networks weights safely and quickly.

### • Diffusers

State-of-the-art diffusion models for image and audio generation in PyTorch.

### • Gradio

Build machine learning demos and other web apps, in just a few lines of Python.

### • Transformers.js

Community library to run pretrained models from Transformers in your browser.

### • PEFT

Parameter efficient finetuning methods for large models

### • Optimum Neuron

Train and Deploy Transformers & Diffusers with AWS Trainium and AWS Inferentia.

### • Tasks

All things about ML tasks: demos, use cases, models, datasets, and more!

### • Amazon SageMaker

Train and Deploy Transformer models with Amazon SageMaker and Hugging Face DLCs.

### • AutoTrain

AutoTrain API and UI