

~~Model pipeline~~

Serving & training optimisation

Sprint 4 - Week 8

INFO 9023 - Machine Learning Systems Design

Thomas Vrancken (t.vrancken@uliege.be)

Matthias Pirlet (matthias.pirlet@uliege.be)

2025 Spring

Agenda

What will we talk about today

Lecture

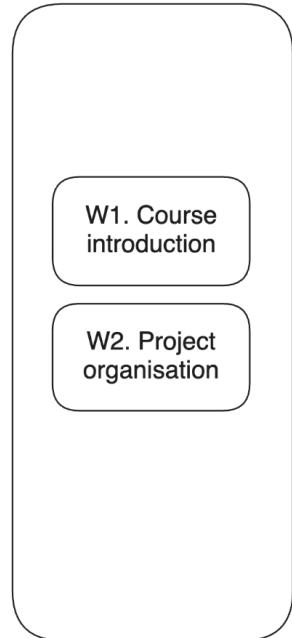
1. Parallel and Distributed Training
2. Model complexity optimisation
3. Model serving optimisation
4. *ML model pipeline* → If we have enough time
5. *ML platforms & orchestrators*

Directed Work

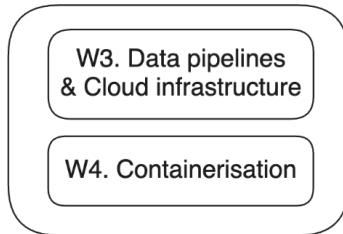
1. ~~Vertex Pipeline~~

Status on our overall course roadmap

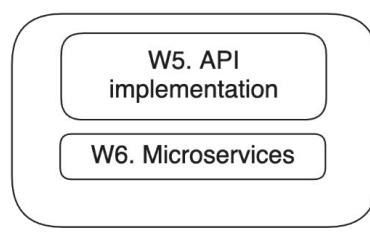
Sprint 1: Project organisation



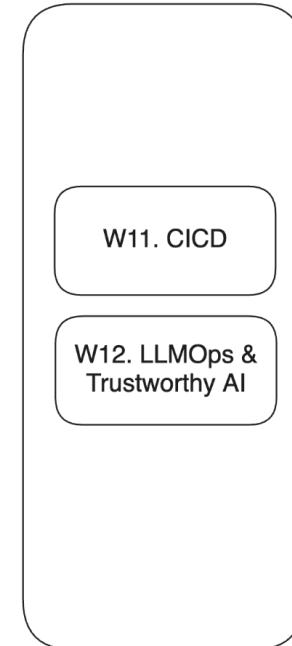
Sprint 2: Cloud & containerisation



Sprint 3: API implementation



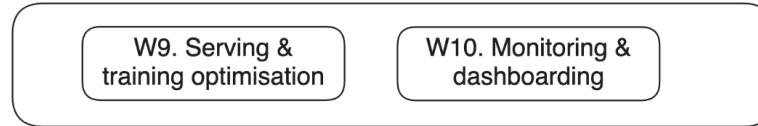
Sprint 6: CICD



Sprint 4: Model deployment



Sprint 5: Optimisation & monitoring



Group projects

Collaboration rules recap

- Final grades for the project are **personal**
- They take into account personal **contribution** to the project

Therefore, make sure that each team members **demonstrate works** on the project

- Split *presentation time* and *answering questions* during MS presentations
- Be active in pushing codes to github

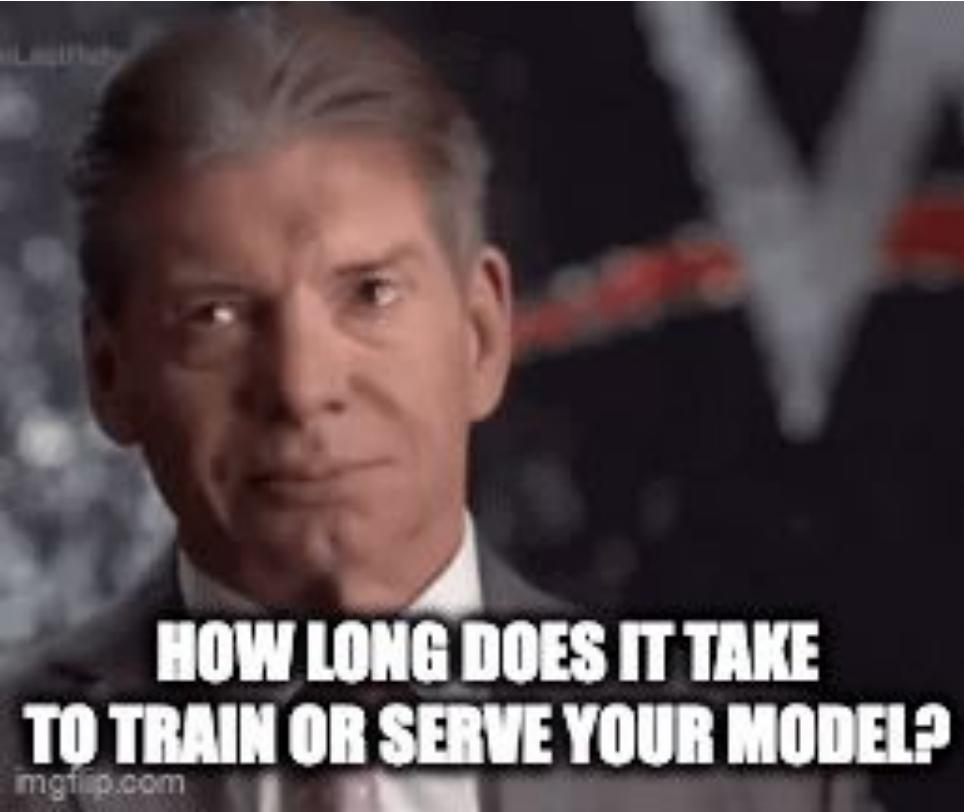
Additional directed work

Collaboration rules recap

Today will be another directed work! You'll get a pass/fail which will count for your final grade.

That means there will be more than the originally announced 5 directed works.

But all directed works will still sum up to 20% of your grade.



Breakout exercise



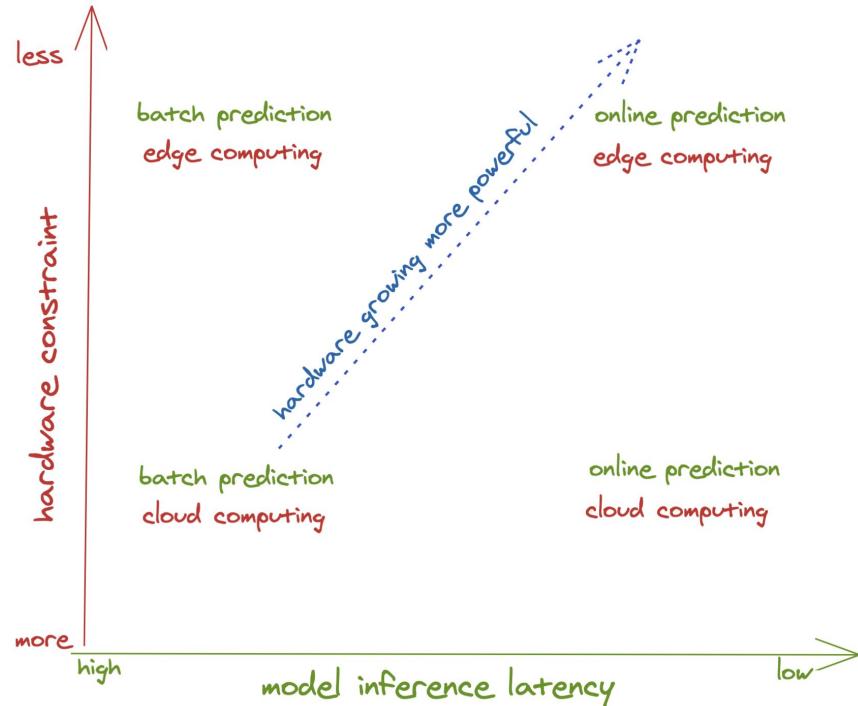
Group of ~4, duration 10 minutes

Identify 2 applications for each quadrant.

How do you determine:

- Batch vs. online prediction
- Edge vs. cloud

Hints: Look at some of the applications on your phone.

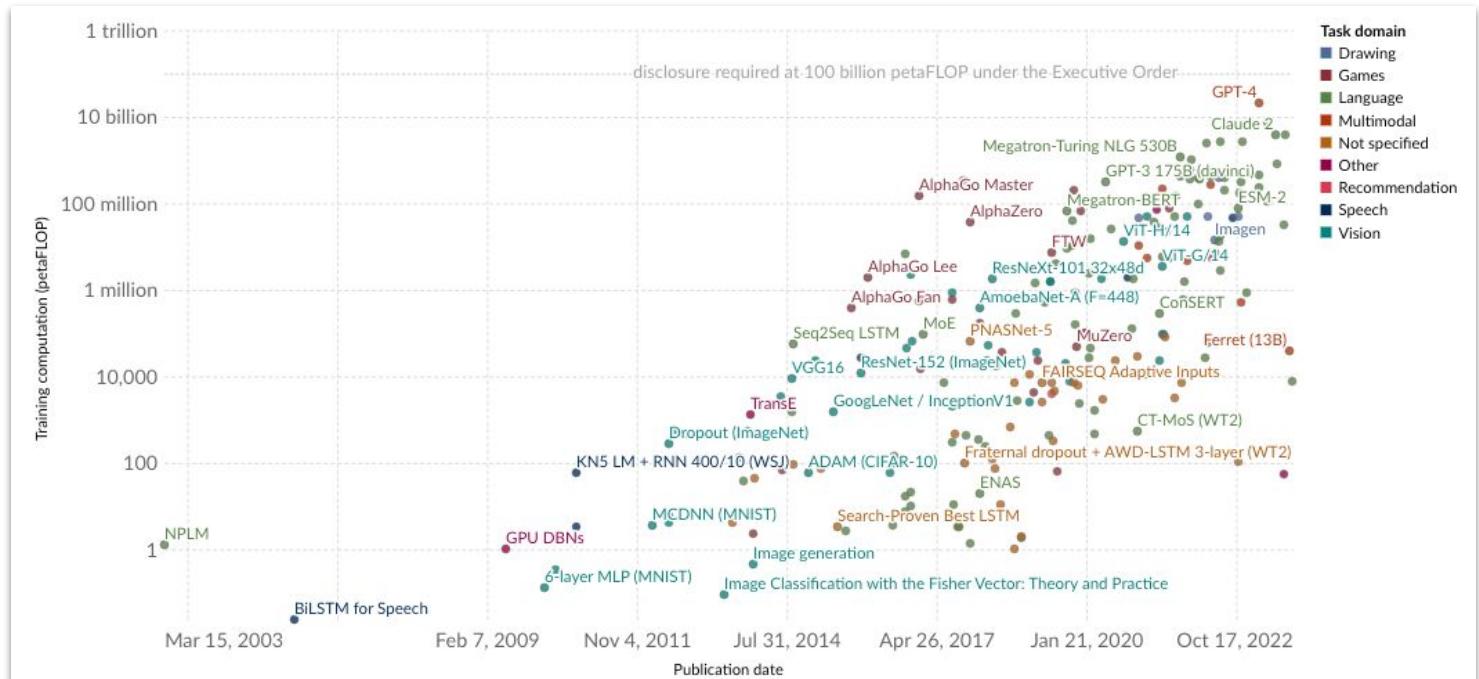


Computational cost of training ML models

Training ML models is often a computationally heavy task



LLMs are pushing the boundaries



Calculate the size of an ML training job

Method 1: Counting operations in the model

How do we represent a single operation?

Floating Point Operation (FLOP): A single operation on a floating variables.

Typically used to calculate computer performance ($\text{FLOP} / \text{seconds} = \text{FLOPS}$) and the total number of operations for a specific computational job.

PetaFLOP: $\sim 10^{15}$ FLOPs.

PetaFLOP-day: The number of FLOP done by a machine doing 1 petaFLOPS if it was running for a full day.

Name	Unit	Value
kiloFLOPS	kFLOPS	10^3
megaFLOPS	MFLOPS	10^6
gigaFLOPS	GFLOPS	10^9
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}
ronnaFLOPS	RFLOPS	10^{27}
quetteFLOPS	QFLOPS	10^{30}

Apple M2 GPU = 3.6 TFLOPS

Calculate the size of an ML training job

Method 1: Counting operations in the model

How do we count the number of operations?

training_compute = (ops_per_forward_pass + ops_per_backward_pass) * n_passes

n_passes = n_epochs * n_examples

Layer	# parameters	# floating point operations
Fully connected layer from N neurons to M neurons	$N \cdot M + M = N \cdot M$ WEIGHTS BIASES NONLINEARITIES	$2 \cdot N \cdot M + M + M = 2 \cdot N \cdot M$

Number of operations in a single forward pass.

Calculate the size of an ML training job

Method 2: GPU time

GPU-days describe the accumulated number of days a single GPU has been used for the training.

If the training lasted 5 days and a total of 4 GPUs were used, that equals 20 GPU-days.

Downside: Dependent of what type of GPUs were used...

Calculate the size of an ML training job

Method 2: GPU time

From GPU-days to FLOP: Let's take [Image GPT](#)

*[..]iGPT-L was trained for roughly **2500 V100-days** [...]*

- **Tensor performance:** We see in the V100 specification that it performs at **125 TeraFLOPS**
- **Usage:** Hard to make full usage of the tensor performance, we assume **30%**
- **Time:** It ran for **2500 days**. And each day is made of **86400 seconds**

SPECIFICATIONS

	V100 PCIe	V100 SXM2	V100S PCIe
GPU Architecture	NVIDIA Volta		
NVIDIA Tensor Cores	640		
NVIDIA CUDA® Cores	5,120		
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	8.2 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	16.4 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS	130 TFLOPS

[NVIDIA V100 specifications](#)

Total number of FLOP to train Image GPT =

$$30\% \times 125\text{e}12 \text{ FLOPS} \times 2500 \text{ days} \times 86400 \text{ sec/day} = 8.1\text{e}21 = 8.1\text{e}6 \text{ PetaFLOP} = 8.1 \text{ ZettaFLOP}$$

Parallel and Distributed Training

Why should we look at parallelisation?

Model training is too slow? You have multiple GPUs? \Rightarrow Use **parallelisation**

An ML model training works as different iterations of updating the model weights/parameters.

How to parallelise this process?



```
# Essential working of ML model training

for step in range(training_steps):
    # Each loops changes the weights to reduce loss,
    # based on a data sample
    weights, loss = update(weights, data)

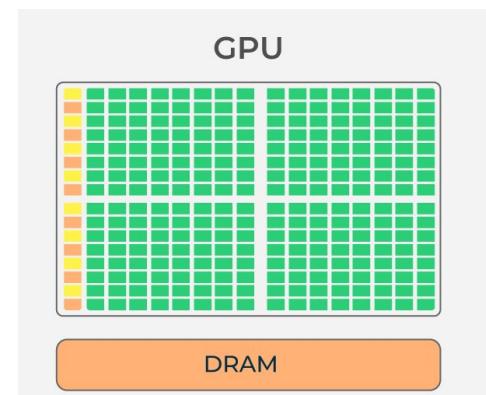
    # If the loss is low enough the training stops
    if loss <= exit_loss:
        break
```

Why should we look at parallelisation?

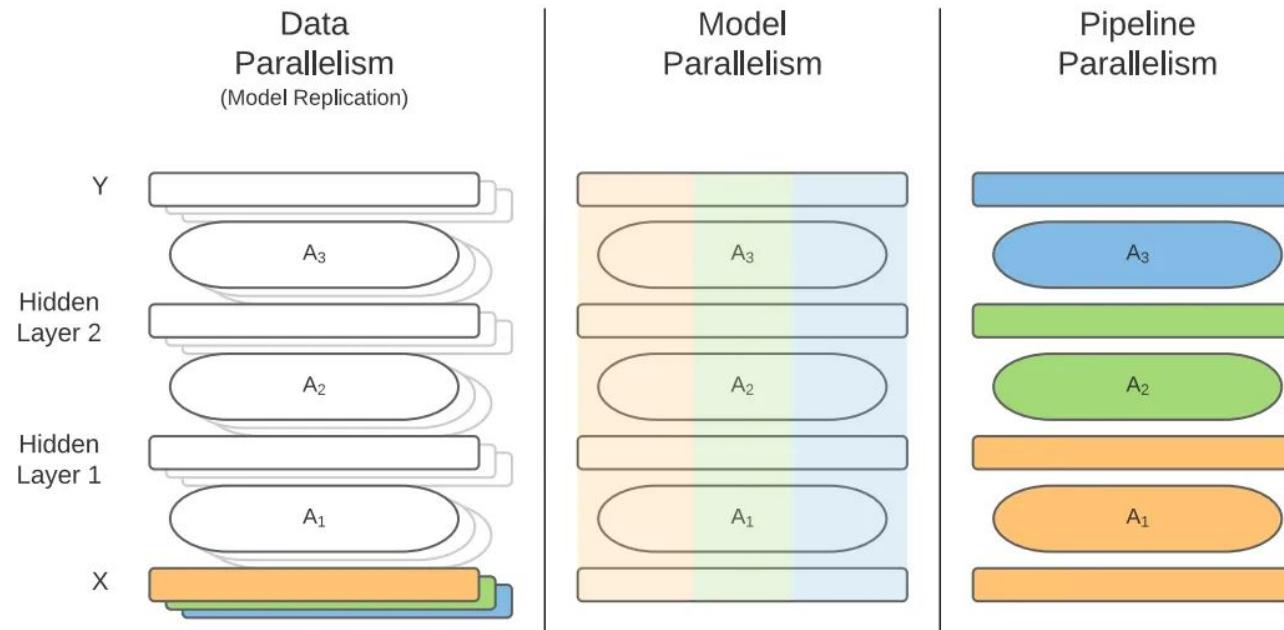
Both due to model architecture and GPU hardware design

A **GPU** (Graphics Processing Unit) is designed for massively **parallel computation**. CPU has a few powerful cores that are designed for general-purpose, Sequential Processing. GPU has many small cores that are designed especially for tasks like graphics rendering and parallel computation.

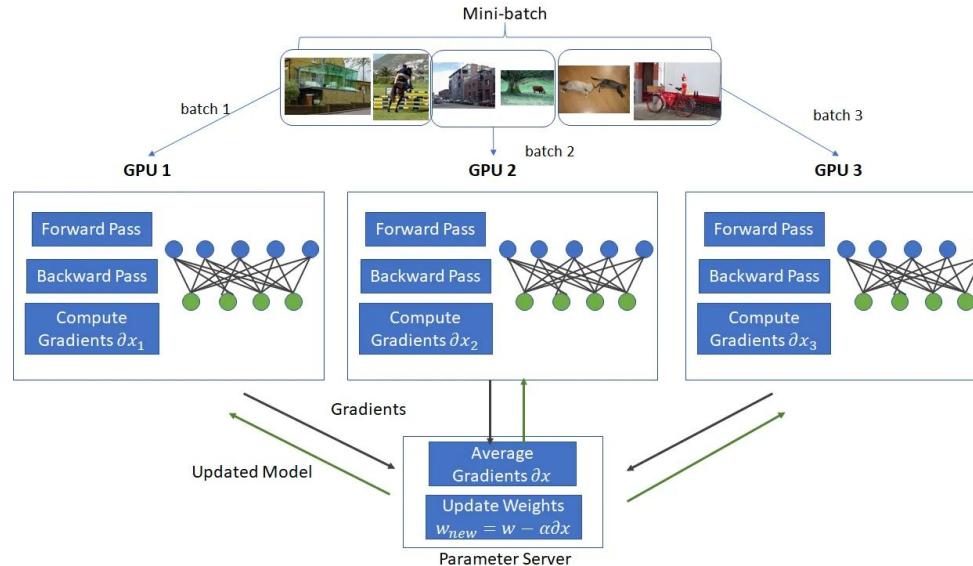
Pros	Cons
<ul style="list-style-type: none">• Parallel Processing• High Throughput• Graphics Rendering	<ul style="list-style-type: none">• Not Versatile (cannot be used for tasks which require processing, such as operating systems or everyday applications)• Power Consumption• Cost



Overview of the different methods for model parallel training

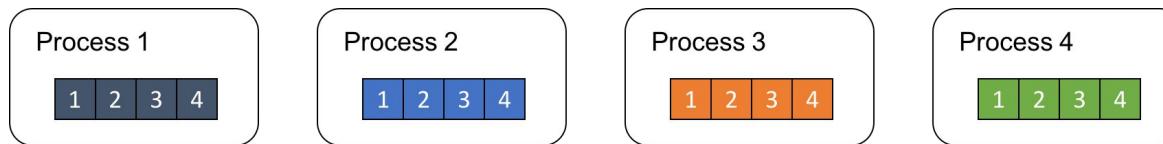


1. Data parallelism

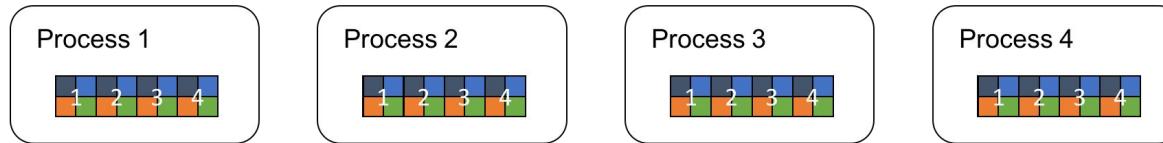


Split a single batch into equal size chunks called **mini-batches (MBS)**, or micro-batches.
Compute the forward and backward passes on all MBS in parallel on **different workers/machines**.
Use (e.g.) an all-reduce algorithm to aggregate the results of each MBS and compute the final weights.

1. Data parallelism



AllReduce



The parameter server then updates the weights for all workers before starting a new step with a new data batch.

1. Data parallelism

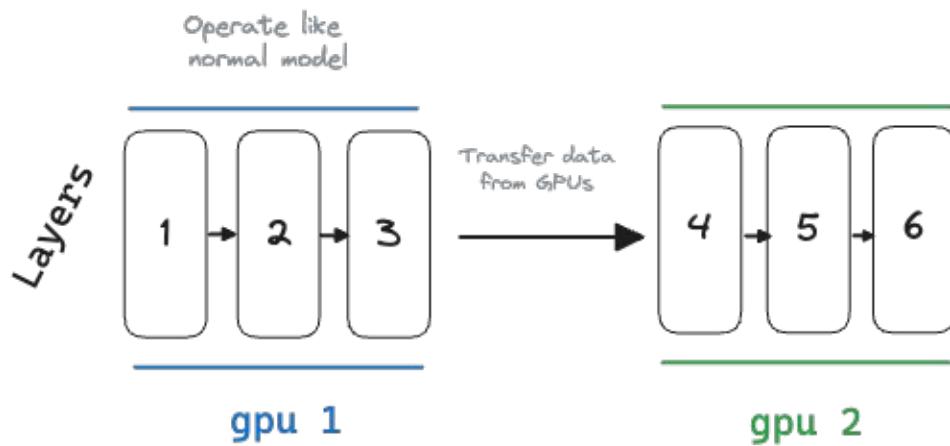
Pros	Cons
<ul style="list-style-type: none">• Easy to implement, usually has available implementations already — DP and DDP in PyTorch. Or MirroredStrategy in Tensorflow.• Fully model-agnostic — works with anything (CNNs, Transformers, GANs, etc.) without modifications.• Easy to predict speed improvements before running (e.g. if we use 4 GPUs, then the convergence will be at most 4x quicker).	<ul style="list-style-type: none">• Requires the model to fit into the memory of a single worker.• Scalable only up to a point (without further tricks and optimizations, such as LAMB) 4 → 8 batch size = Optimisation 1024 → 2048 batch size = Each worker get 2x more compute so no optimisation• Big communication payload if you have a large models with a lot of parameters. You need to send each parameters back and forth to all workers at each batch... ResNet50 = 24M parameters = 1.47GB• Soft requirement for GPUs to be on the same machine or on a local network

2. Naive model parallelism

Naive (or vertical) model parallelism spreads groups of **model layers** across multiple GPUs.

Transmit information at each pass between GPUs.
Because to compute layer l , you typically need all outputs from layer $l-1$, so you need to wait until all of the workers computing layer $l-1$ have finished.

It is almost exclusively used within a single physical cluster due to high bandwidth between devices (GPUs) the motherboard provides, rather than across the network.



2. Naive model parallelism

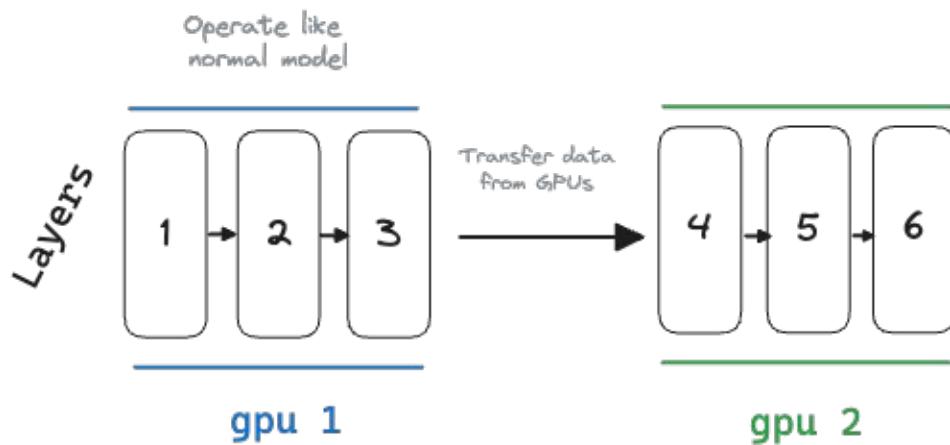
Overhead in copying data:

E.g.

$1 \times 24\text{GB card} = 4 \times 6\text{GB cards using naive MP}$



Slower due to data transfer

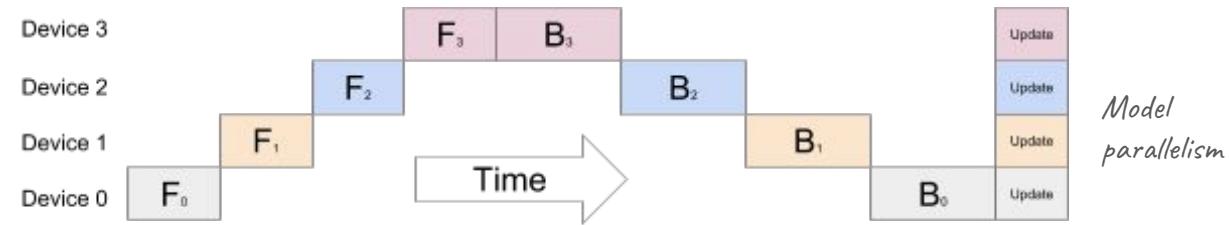


2. Naive model parallelism

Pros	Cons
<ul style="list-style-type: none">• Greatly reduced memory usage, proportional to the number of devices used. Relevant in the era of Large models• Can use NVIDIA's Megatron-LM library	<ul style="list-style-type: none">• Model-dependent — different way to divide the model which is dependent on the model architecture• Expensive communications• Idle time for your GPUs while other groups of layers are being processed• It's hard! Implementing Model Parallelism in your ML program is typically done by hand, requires knowledge of your model and its layers, as well as how they interact with your hardware

2. Naive model parallelism

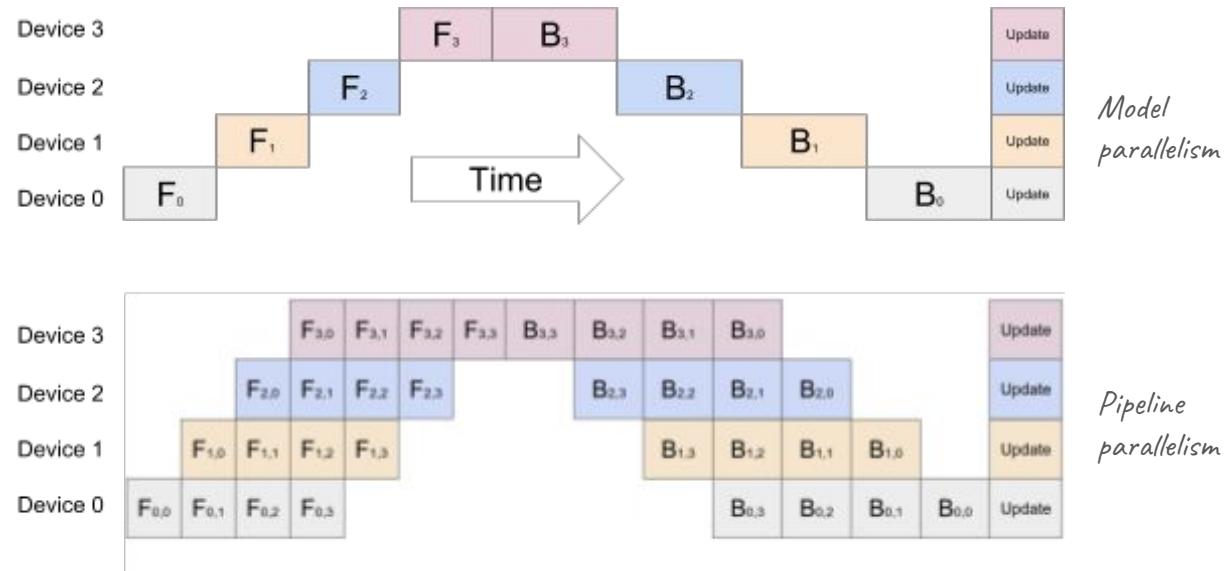
Long idle time of your GPUs.



3. Pipeline parallelism

Solution: Mix of data and model parallelism.

Split data in mini-batches and compute passes for different groups of layers sequentially.

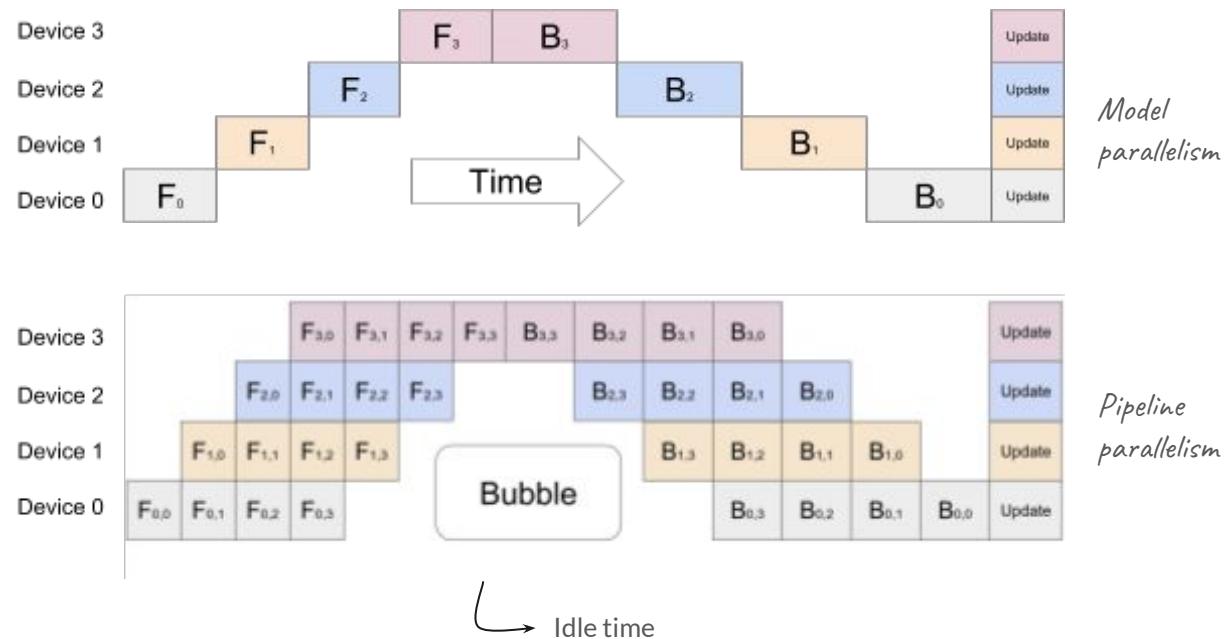


3. Pipeline parallelism

Still some idle time called the **bubble**.

Traditional Pipeline API solutions:

- PyTorch
- FairScale
- DeepSpeed
- Megatron-LM

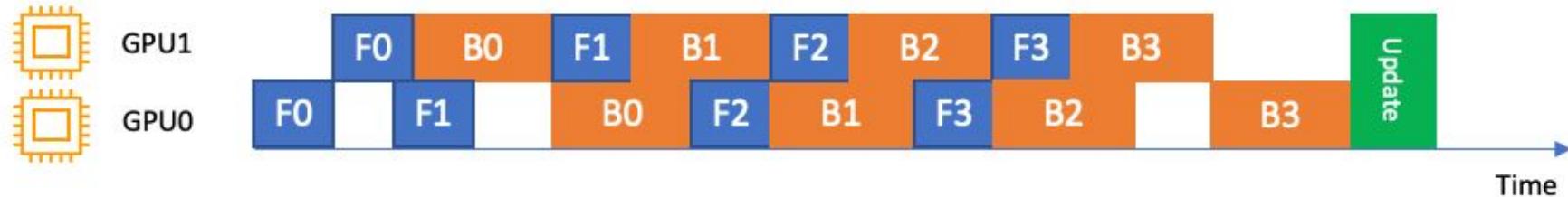


3. Pipeline parallelism

Pros	Cons
<ul style="list-style-type: none">Reduces memory usage and GPU idle time	<ul style="list-style-type: none">More complexity (extra hyperparameter to optimise)Limited framework. PyTorch requires a single Tensor or a tuple of Tensors as the only input and output. No other python variables can be passed.Have to arrange each layer so that the output of one model becomes an input to the other model.

3. Pipeline parallelism

Interleaved Pipeline: Further reduce the bubble (idle) time by prioritising backward passes.



Supported by DeepSpeed, Varuna and SageMaker.

When to use which parallelisation technique?

Technique	Implementation complexity	Model size & structure	Hardware
Data parallelism	Simple. Integrated in frameworks like Pytorch or Tensorflow.	Your model fits into a single GPU (usually small batch size - e.g. 64).	Works on both single-machine and multi-machine configurations.
Model parallelism	More complex.	Model does not fit into a single device.	Recommended on single machine due to high communication costs .
Pipeline parallelism	Complex but supported out-of-the-box by some frameworks	Model does not fit into a single device. Works well for sequential models like CNN and transformers.	Recommended on single machine due to high communication costs . High machine utilization and memory efficiency.

Ray



Ray is a framework that provides a simple, universal API to build distributed ML applications. It is designed to scale Python applications from a single computer to a cluster with ease.

- **Does not provide hardware:** Ray itself doesn't provide the physical compute resources (like CPUs or GPUs).
- **Scaling:** Mostly used when transitioning from a single machine to distributed training.
- **Model integration:** Integrates directly with Pytorch, Huggingface or Scikit-learn.
- **Cloud Integration:** You can deploy Ray easily on Azure, GCP or AWS.

Model complexity optimisation

Difference between large and simple models

Large models

- More features, hidden units in Neural Network, trees for decision trees, parameters, ...
- Capture more complex relationship in the data
- Generalises better
- More memory intensive
 ⇒ Slower!

For accuracy sensitive complex use cases where a lot of training data is available.

Simple models

- Less features, smaller Neural Network, less trees, less parameters, ...
- Fits less well on complex relationship in the data
- For simpler tasks
- Requires less memory

⇒ Faster!

For low-latency or cost efficient models capturing simpler systems.

VS

Clear objective: What are your speed requirements?

- Satisficing metrics: E.g. prediction latency, cost, ... ⇒ Set a threshold
- Optimising metrics: Accuracy, precision, recall, ... ⇒ Optimise

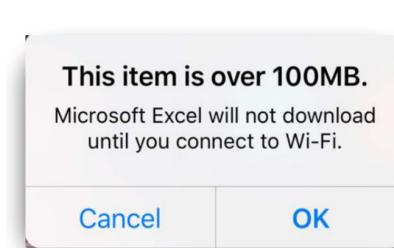
Nevertheless, sometimes you start with a large model which you want to reduce size and complexity.

- Model speed
 - *E.g. not detecting pedestrian in time*
- Model size
 - *E.g. uploading/updating model over network*
- Energy efficiency
 - *E.g. AlphaGo: 1920 CPUs and 280 GPUs*

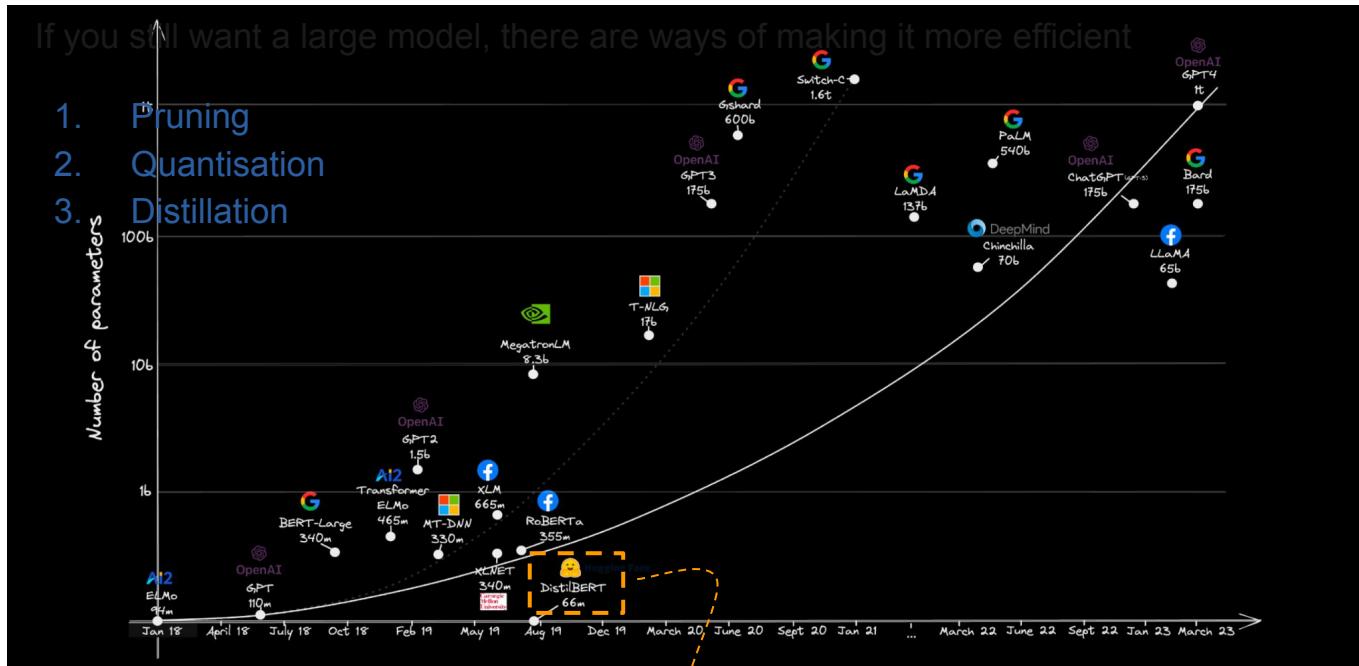
⇒ \$3000 electric bill per game



This image is licensed under CC-BY 2.0



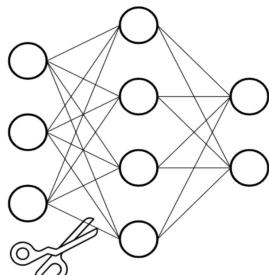
Optimising your model can be done through model optimisation.



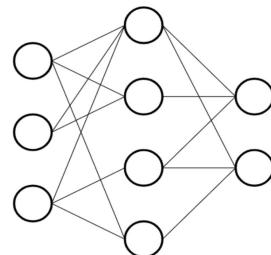
97% of the accuracy for 50% of the size of BERT - its original model

Pruning for model compression.

Pruning: Removing underutilised weight connections in a network to increase inference speed and decrease model storage size.



Before pruning

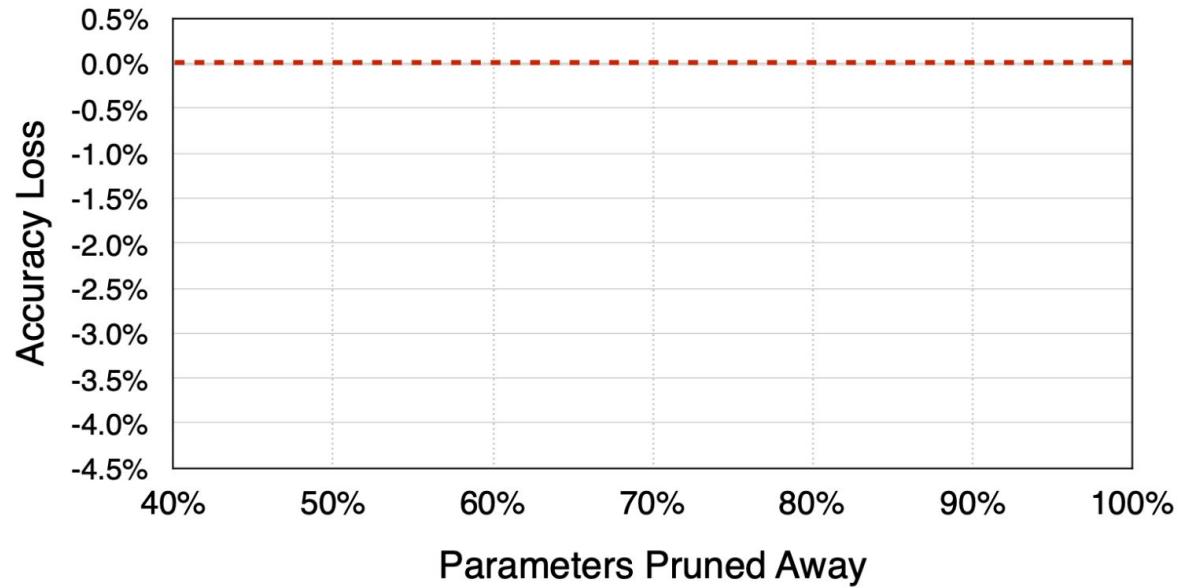


After pruning

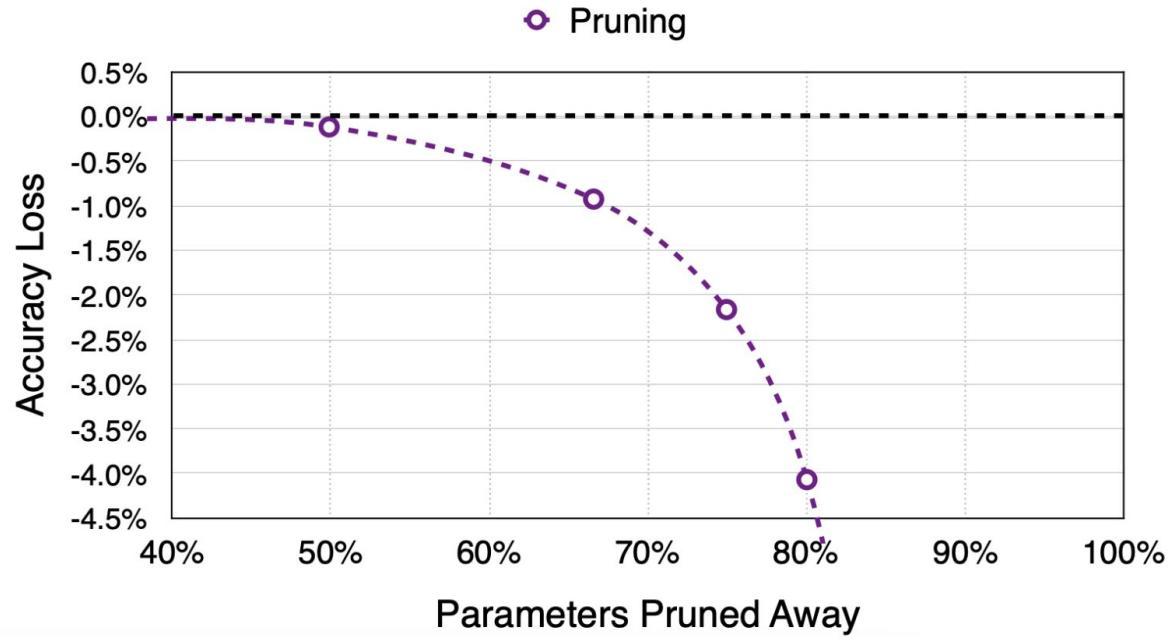
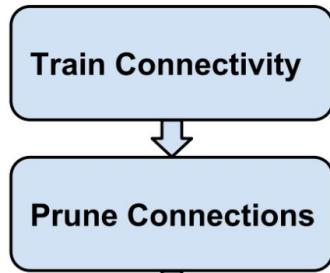
Model	Non-sparse accuracy (float)	Sparse accuracy (float)	Number of non-zero parameters (NNZ) in sparse models
InceptionV3	78.1%, 27.1M parameters	78.0% @ 50% sparsity	13.6M
		76.1% @ 75% sparsity	6.8M
		74.6% @ 87.5% sparsity	3.3M
GNMT EN-DE	26.77 BLEU, 211 M parameters	26.86 BLEU @ 80% sparsity	44 M
		26.52 BLEU @ 85% sparsity	33 M
		26.19 BLEU @ 90% sparsity	22 M
GNMT DE-EN	29.47 BLEU, 211 M parameters	29.50 BLEU @ 80% sparsity	44 M
		29.24 BLEU @ 85% sparsity	33 M
		28.81 BLEU @ 90% sparsity	22 M

Pruning for model compression.

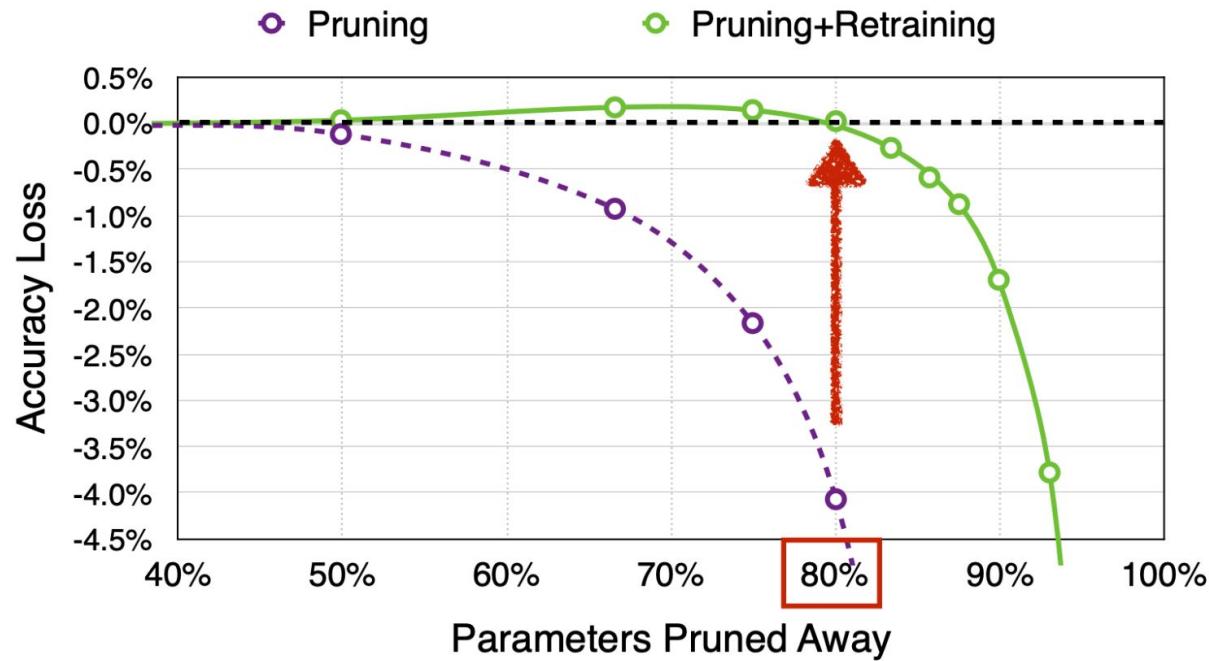
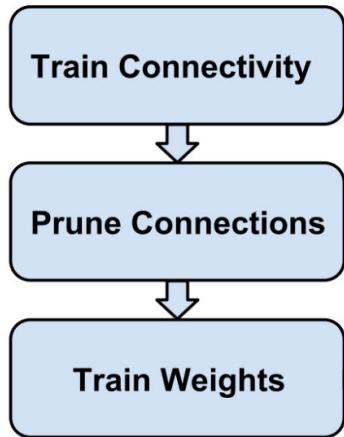
Train Connectivity



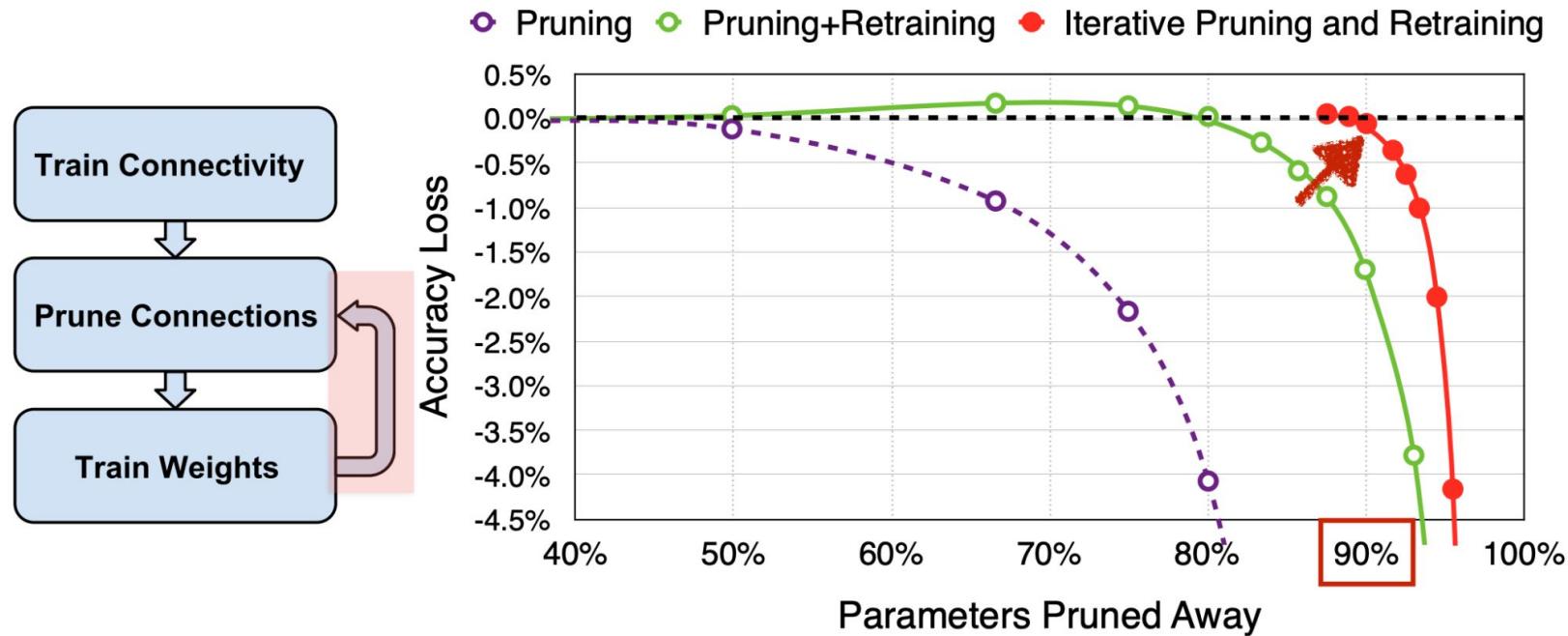
Pruning for model compression.



Pruning for model compression.



Pruning for model compression.



Pruning happens in human brain.

1000 Trillion
Synapses

50 Trillion
Synapses

500 Trillion
Synapses



This image is in the public domain

Newborn



This image is in the public domain

1 year old

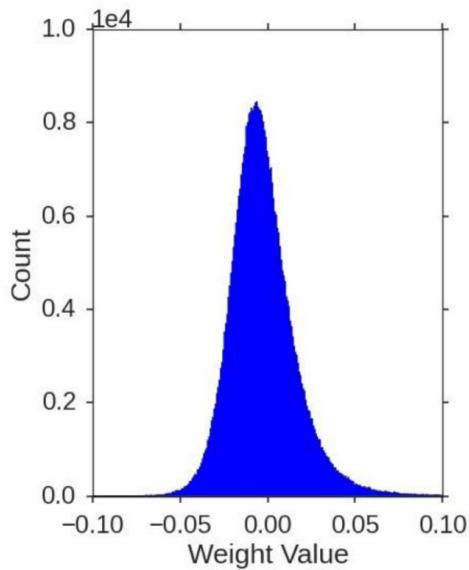


This image is in the public domain

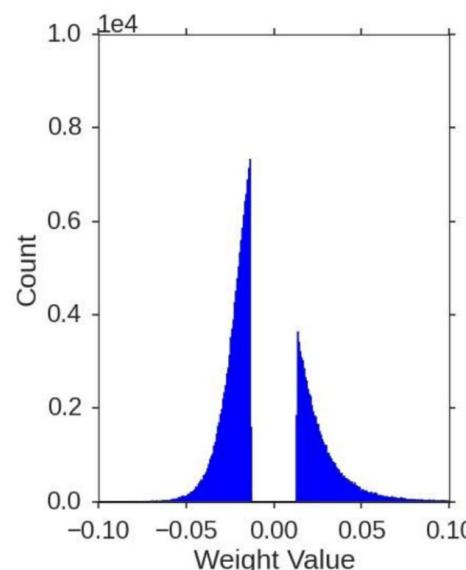
Adolescent

Pruning changes weight distributions.

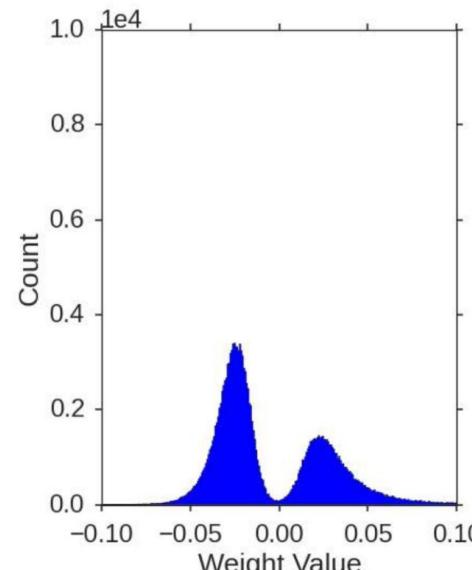
Before Pruning



After Pruning



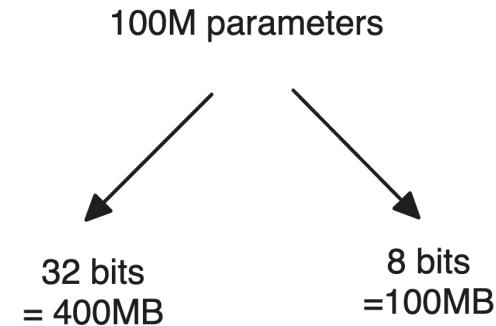
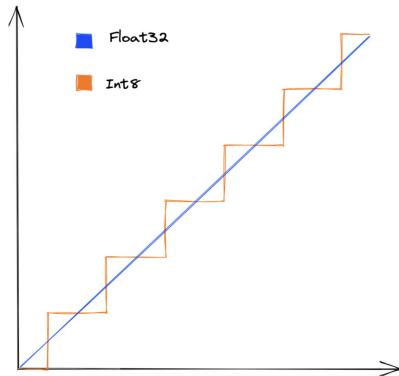
After Retraining



Quantisation

Quantisation: Reduction in precision of the numbers used to represent a models parameters.

E.g. Change the weights of a model from **float32** to **int8**.



Quantisation

- **Post-training quantization:** This involves applying quantization to a fully trained model. The weights and activations are converted from floating-point to lower-precision formats, typically 8-bit integers.
- **Quantization-aware training:** This involves simulating low-precision weights and activations during the training process itself. By incorporating quantization into the training, the model can adapt to the reduced precision and often results in higher accuracy compared to post-training quantization.
- **Dynamic quantization:** This approach quantizes the weights of the model ahead of time but quantizes the activations dynamically at runtime. This is particularly useful for models with recurrent layers, where the input data shape varies from one batch to another.

Quantisation

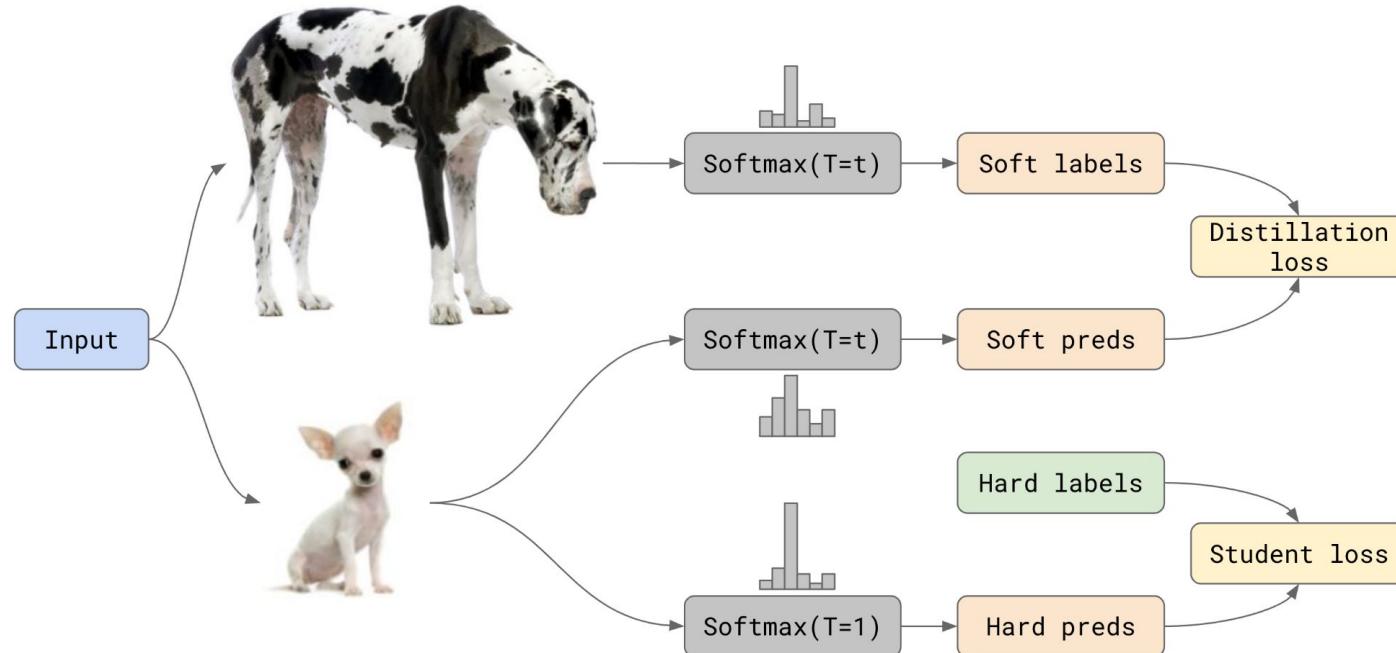
Model	Top-1 Accuracy (Original)	Top-1 Accuracy (Post Training Quantized)	Top-1 Accuracy (Quantization Aware Training)	Latency (Original) (ms)	Latency (Post Training Quantized) (ms)	Latency (Quantization Aware Training) (ms)	Size (Original) (MB)	Size (Optimized) (MB)
MobileNet-v1-1-224	0.709	0.657	0.70	124	112	64	16.9	4.3
MobileNet-v2-1-224	0.719	0.637	0.709	89	98	54	14	3.6
Inception_v3	0.78	0.772	0.775	1130	845	543	95.7	23.9
Resnet_v2_101	0.770	0.768	N/A	3973	2868	N/A	178.3	44.9

TensorFlow Lite [documentation](#)

Quantisation

Pros	Cons
<ul style="list-style-type: none">• Reduce memory footprint• Increase computation speed• Bigger batch size• Computation on 16 bits is faster than on 32 bits	<ul style="list-style-type: none">• Smaller range of values• Values rounded to 0• Need an efficient rounding technique

Knowledge distillation: training a smaller model to mimic a larger one based on high Softmax temperature.



Interesting example: [DistilBERT](#)

Knowledge distillation: training a smaller model to mimic a larger one based on high Softmax temperature.

Pros	Cons
<ul style="list-style-type: none">• Fast to train student network if teacher is pre-trained.• Teacher and student can be completely different architectures.	<ul style="list-style-type: none">• If teacher is not pre-trained, may need more data & time to first train teacher.• Sensitive to applications and model architectures.

Interesting example: [DistilBERT](#)

DistilBERT is a good example of distilled model.

Hugging Face

Models Datasets Spaces Docs Solutions Pricing

distilbert-base-uncased like 183

Fill-Mask PyTorch TensorFlow JAX Rust Safetensors Transformers bookcorpus wikipedia English distilbert exbert

AutoTrain Compatible arxiv:1910.01108 License: apache-2.0

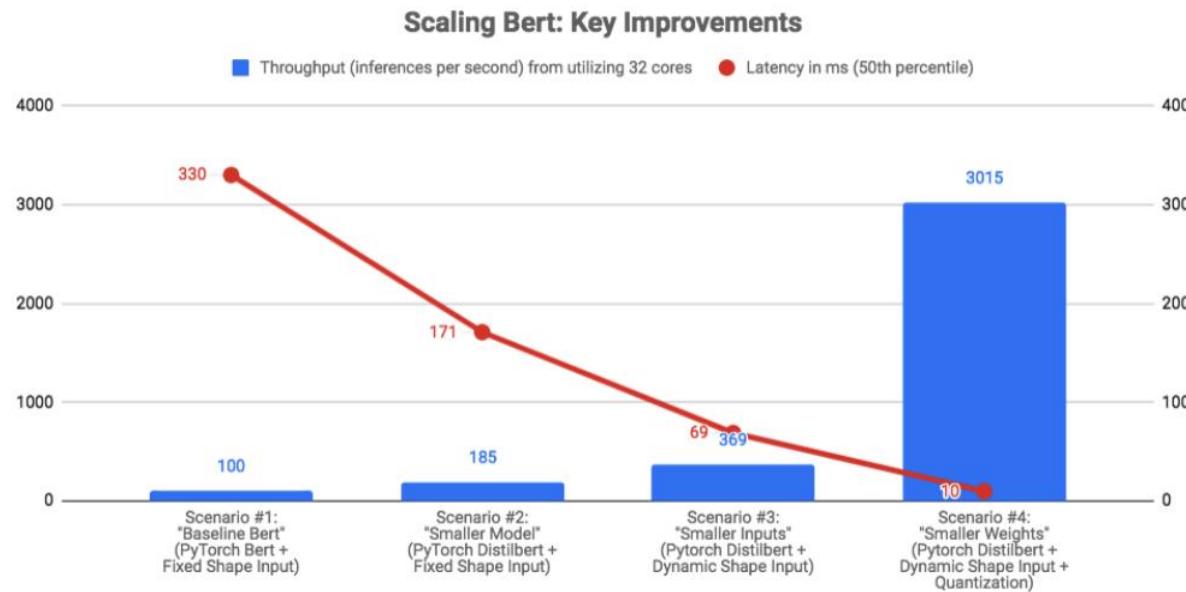
Model card Files and versions Community 7 Edit model card Train Deploy Use in Transformers

DistilBERT base model (uncased)

Downloads last month 9,743,608



Improvement in terms of latency and throughput is significant.

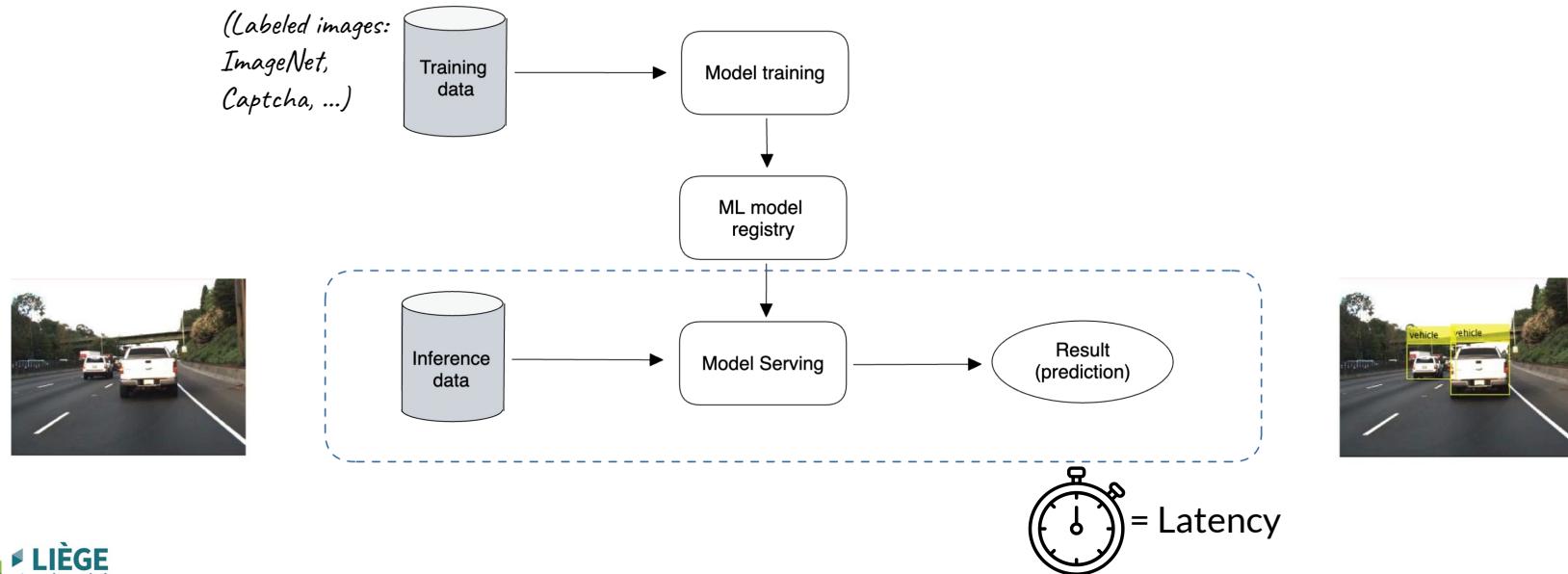


Model serving optimisation

What is model serving and latency?

An ML model is first trained on a (usually labeled) data set. It is then called on **inference** by users during **model serving**.

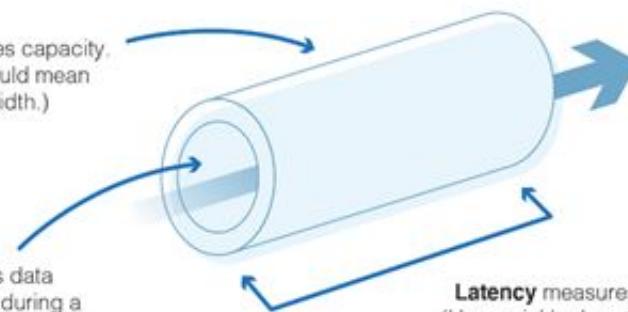
Latency is the delay between an ML model receiving inference data and producing a specific result.



Different serving metrics

Network Latency vs. Throughput vs. Bandwidth

Bandwidth measures capacity.
(A bigger pipe would mean higher bandwidth.)



Throughput measures data transmitted and received during a specific time period. (Throughput is the water running through the pipe.)

Latency measures data speed.
(How quickly does the water in the pipe reach its destination?)

Despite closely related, the performance in each metric is not linear (e.g. can have a low latency but not a great throughput).

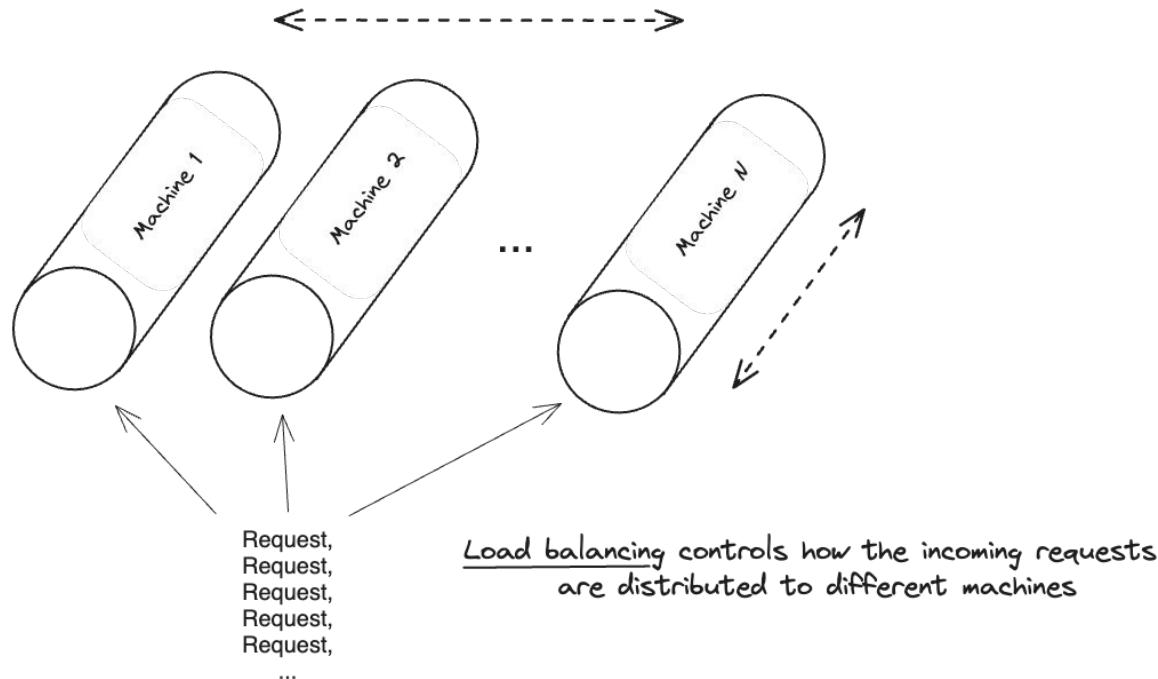
Cloud deployment: Autoscaling

Different Cloud platforms allow for auto scaling and load balancing.

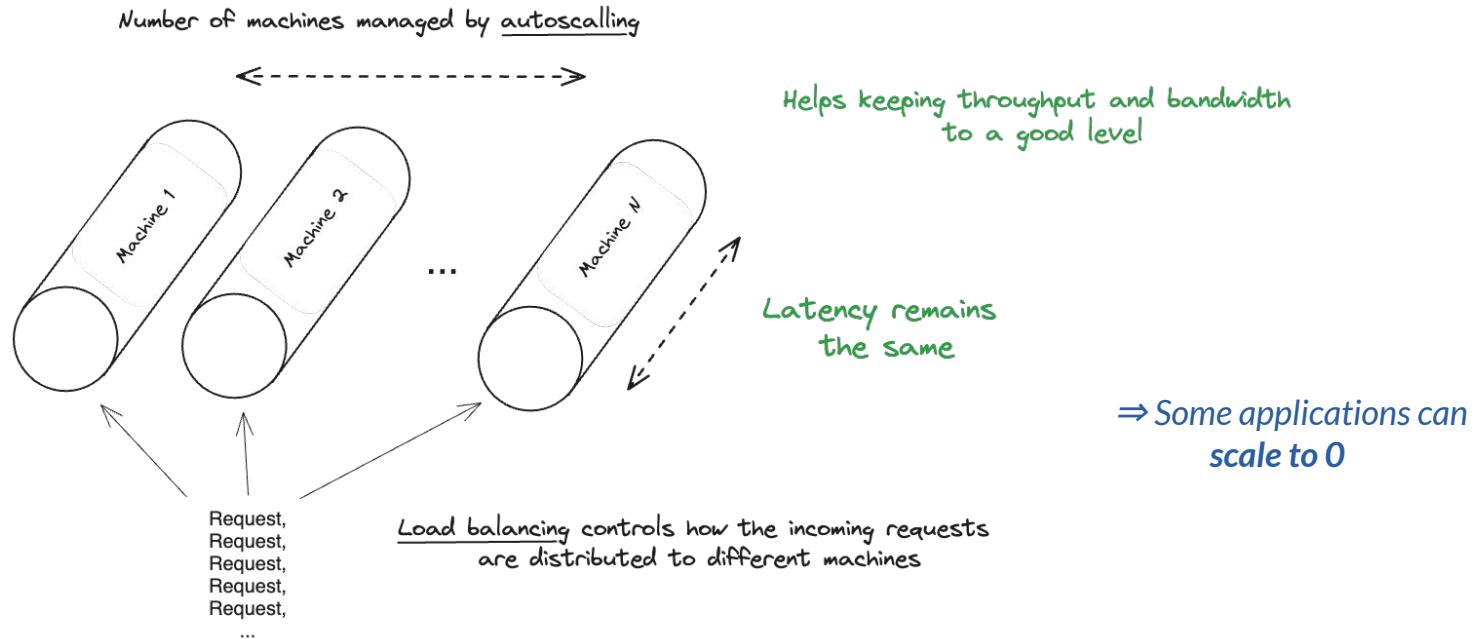
They will spin more machines to handle incoming requests in parallel.

It is detrimental to handling a large amount of incoming requests.

Number of machines managed by autoscaling



Autoscaling will improve bandwidth and throughput (latency of a single request will not be impacted).



Ways of reducing your serving latency without changing your ML model

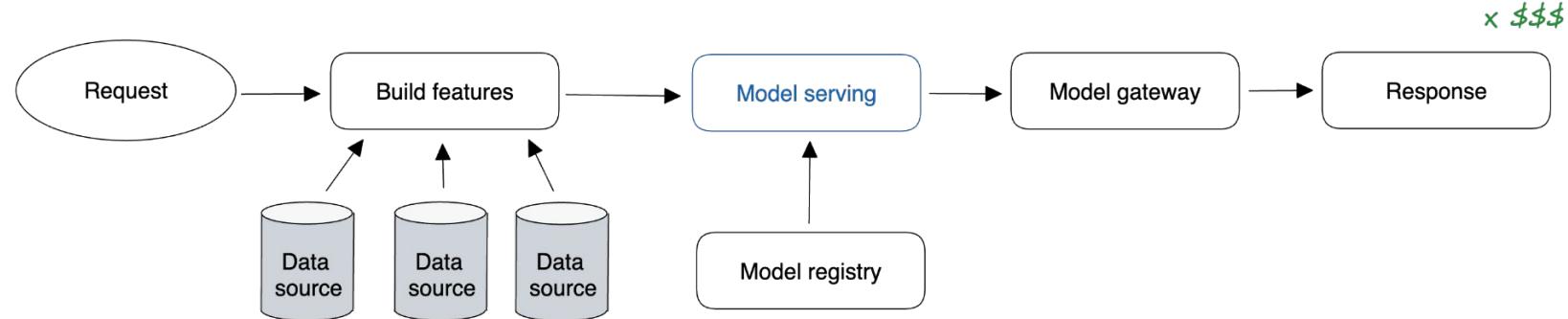
We look at three different methods:

1. Optimise the **pipeline** around your model
2. Optimise the **hardware** used by your model
3. Optimise the **framework** used by your model

Look at different parts of your pipeline.

Price for house:

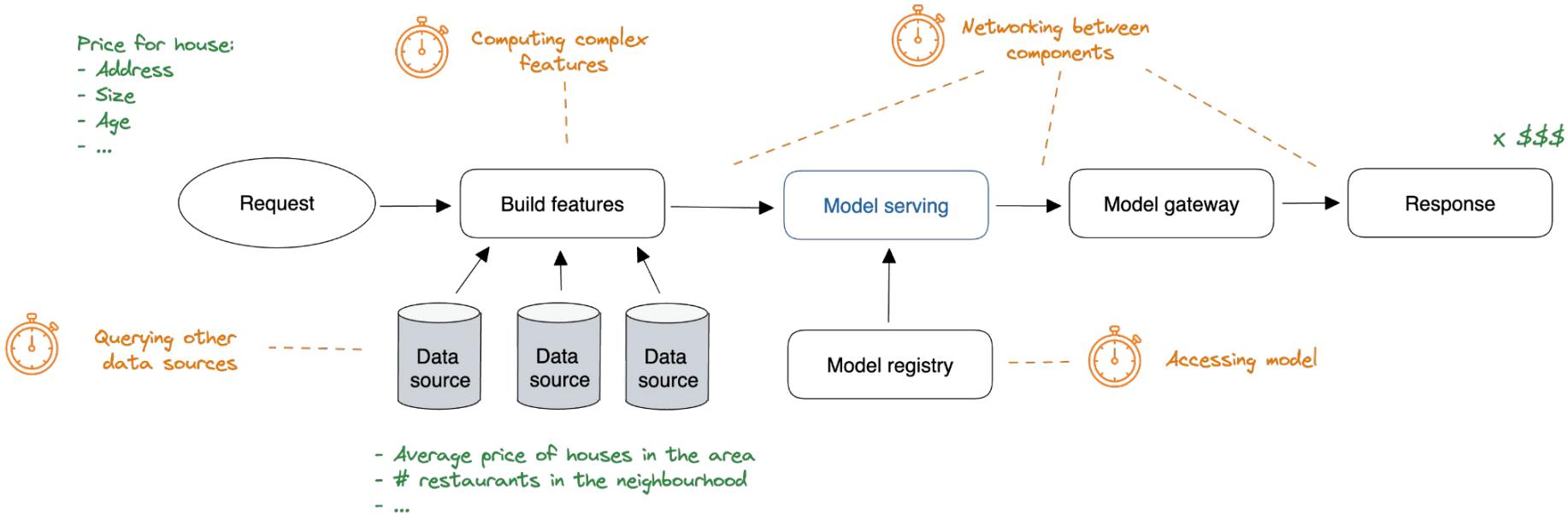
- Address
- Size
- Age
- ...



- Average price of houses in the area
- # restaurants in the neighbourhood
- ...

Look at different parts of your pipeline.

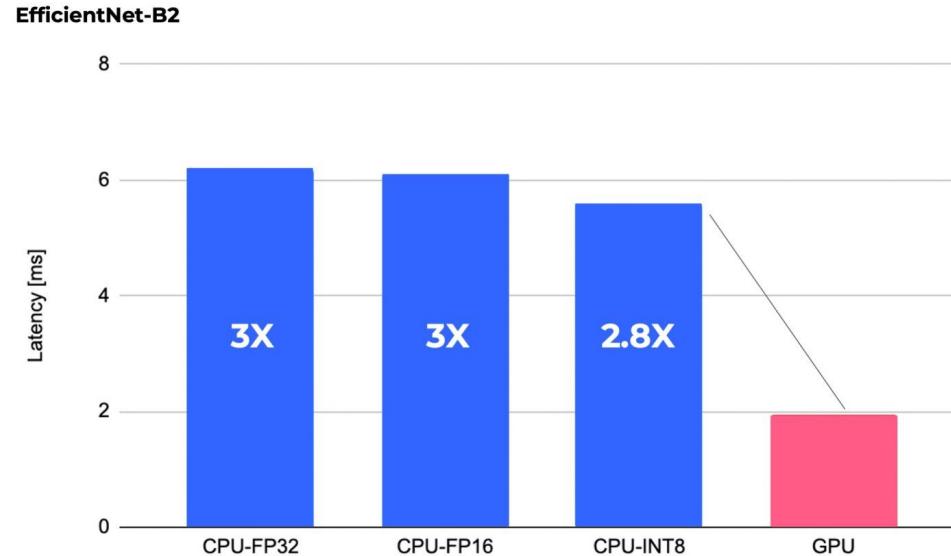
You ML model might not be the bottleneck!



Optimise the hardware used by your ML model.

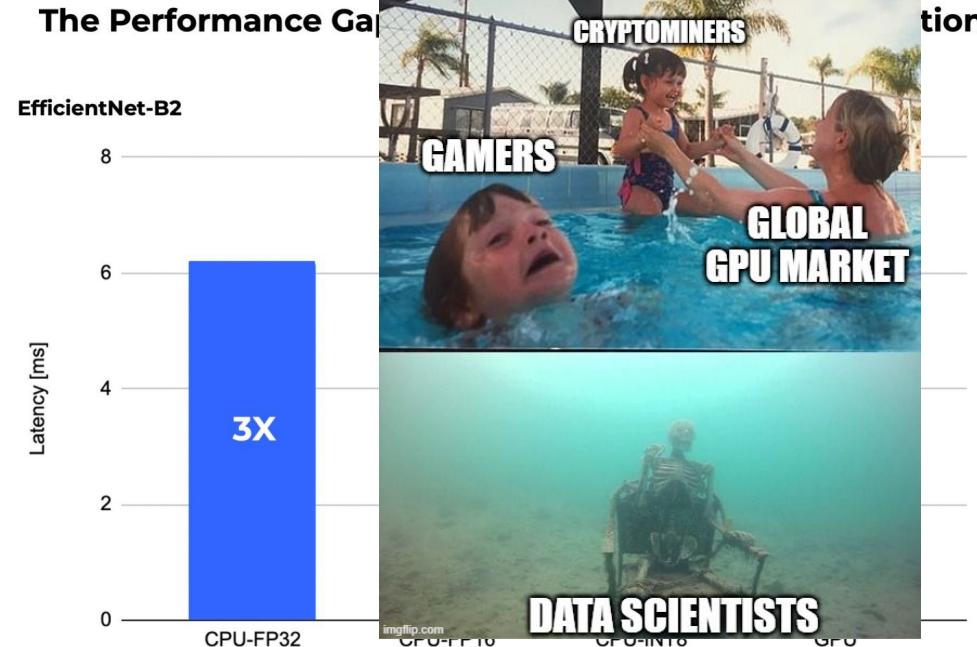
Hardware

The Performance Gap after Compilation and Quantization



Optimise the hardware used by your ML model.

Hardware



Tensor Processing Unit (TPU)

Processor	mm^2	Clock MHz	TDP Watts	Idle Watts	Memory GB/sec	Peak TOPS/chip	
						8b int.	32b FP
CPU: Haswell (18 core)	662	2300	145	41	51	2.6	1.3
GPU: Nvidia K80 (2 / card)	561	560	150	25	160	--	2.8
TPU	<331*	700	75	28	34	91.8	--

*TPU is less than half die size of the Intel Haswell processor

K80 and TPU in 28 nm process; Haswell fabbed in Intel 22 nm process

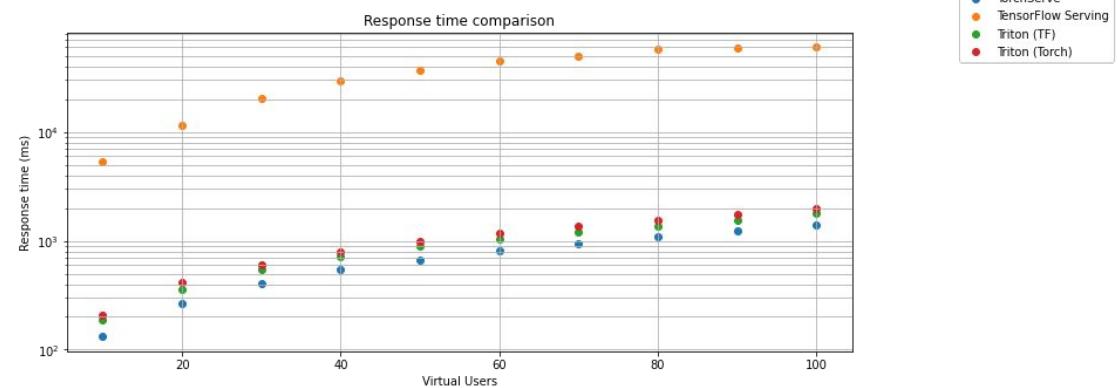
These chips and platforms chosen for comparison because widely deployed in Google data centers

You can wrap your model using different model frameworks.

TorchServe: flexible and easy-to-use tool for serving PyTorch models created by Facebook.

vLMM: Enhance the inference and serving efficiency of large language models (LLMs). Using PagedAttention and Continuous Batching, vLMM optimizes memory management and accelerates inference speeds, supporting distributed inference across multiple GPUs.

Triton™ Inference Server: NVIDIA's Triton Inference Server provides a cloud and edge inferencing solution optimized for both CPUs and GPUs.

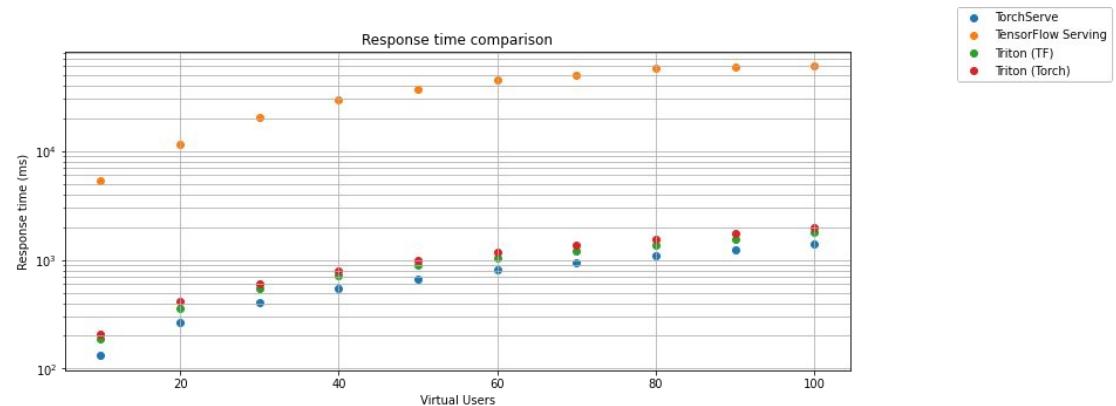


You can wrap your model using different model frameworks.

Triton™ Inference Server

NVIDIA Dynamo: Announced on March 18, 2025, at GTC 2025, NVIDIA Dynamo is an open-source, low-latency inference framework designed to serve generative AI models in distributed environments. It builds upon the success of Triton by introducing a modular architecture tailored for large-scale AI deployments.

Dynamo is designed to be inference engine agnostic (supports TRT-LLM, vLLM, SGLang or others).



NVIDIA Dynamo: Zoom-in

Features.



Disaggregated Serving

Separates LLM context (prefill) and generation (decode) phases across distinct GPUs, enabling tailored model parallelism and independent GPU allocation to increase requests served per GPU.



GPU Planner

Monitors GPU capacity in distributed inference environments and dynamically allocates GPU workers across context and generation phases to resolve bottlenecks and optimize performance.



Smart Router

Routes inference traffic efficiently, minimizing costly recomputation of repeat or overlapping requests to preserve compute resources while ensuring balanced load distribution across large GPU fleets.



NIXL Low-Latency Communication Library

Accelerates data movement in distributed inference settings while simplifying transfer complexities across diverse hardware, including GPUs, CPUs, networks, and storage.

“When serving the open-source DeepSeek-R1 671B reasoning model on NVIDIA GB200 NVL72, NVIDIA Dynamo increased the number of requests served by up to 30x...”

NVIDIA Dynamo: Zoom-in

Installation

The following examples require a few system level packages. Recommended to use Ubuntu 24.04 with a x86_64 CPU. See [support_matrix.md](#)

```
apt-get update  
DEBIAN_FRONTEND=noninteractive apt-get install -yq python3-dev python3-pip python3-venv libucx  
python3 -m venv venv  
source venv/bin/activate  
  
pip install ai-dynamo[all]
```

Running and Interacting with an LLM Locally

To run a model and interact with it locally you can call `dynamo run` with a hugging face model. `dynamo run` supports several backends including: `mistralrs`, `sgllang`, `vllm`, and `tensorrtllm`.

Example Command

```
dynamo run out=vllm deepseek-ai/DeepSeek-R1-Distill-Llama-8B
```

```
? User > Hello, how are you?
```

```
✓ User · Hello, how are you?
```

```
Okay, so I'm trying to figure out how to respond to the user's greeting. They said, "Hello, ho
```

LLM Serving

Dynamo provides a simple way to spin up a local set of inference components including:

- **OpenAI Compatible Frontend** – High performance OpenAI compatible http api server written in Rust.
- **Basic and Kv Aware Router** – Route and load balance traffic to a set of workers.
- **Workers** – Set of pre-configured LLM serving engines.

To run a minimal configuration you can use a pre-configured example.

Start Dynamo Distributed Runtime Services

First start the Dynamo Distributed Runtime services:

```
docker compose -f deploy/docker-compose.yml up -d
```

Start Dynamo LLM Serving Components

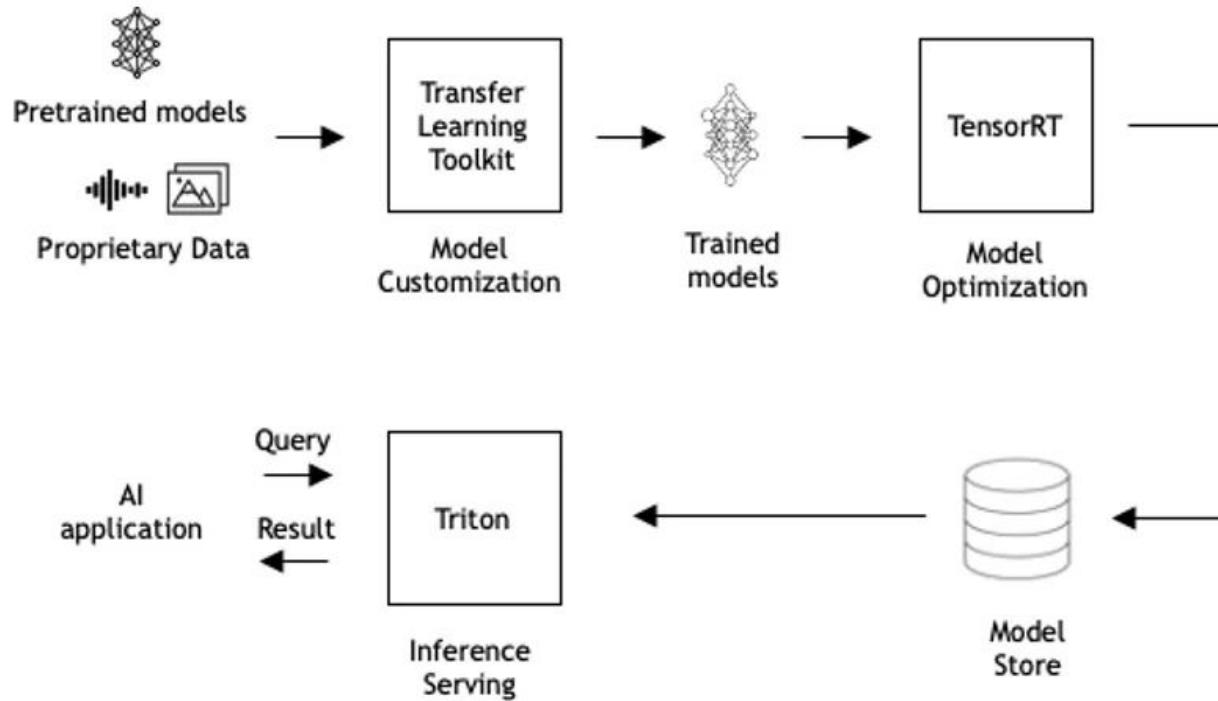
Next serve a minimal configuration with an http server, basic round-robin router, and a single worker.

```
cd examples/llm
dynamo serve graphs.agg:Frontend -f configs/agg.yaml
```

Send a Request

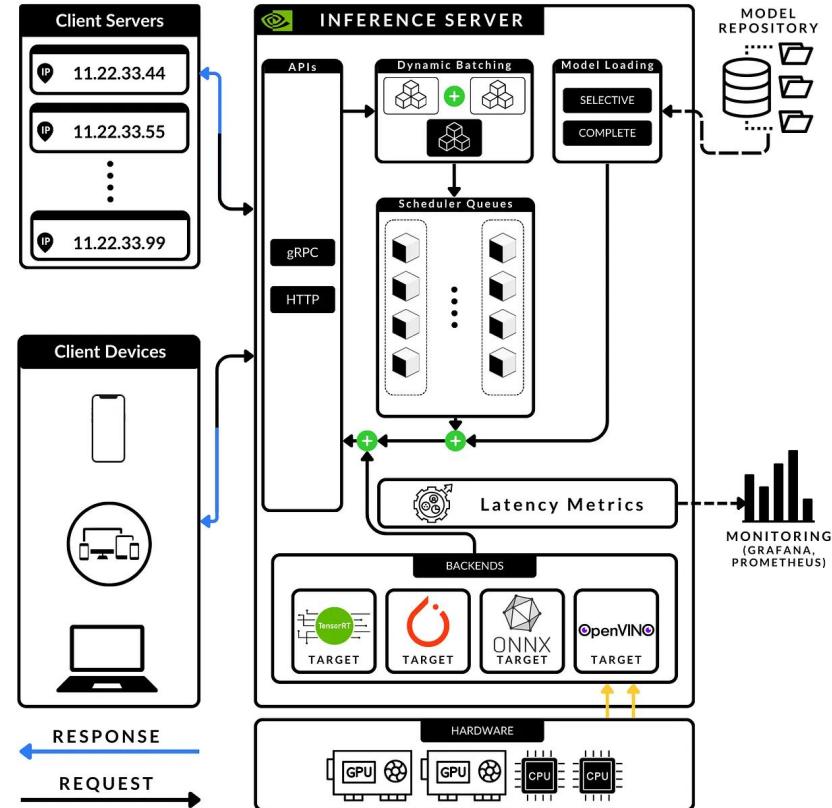
```
curl localhost:8000/v1/chat/completions -H "Content-Type: application/json" -d '{
    "model": "deepseek-ai/DeepSeek-R1-Distill-Llama-8B",
    "messages": [
        {
            "role": "user",
            "content": "Hello, how are you?"
        }
    ],
    "stream":false,
    "max_tokens": 300
}' | jq
```

What is NVIDIA Triton Inference Server (T.I.S) ?



NVIDIA TRITON SERVER

What is NVIDIA Triton Inference Server (T.I.S) ?



Why use Triton servers ?

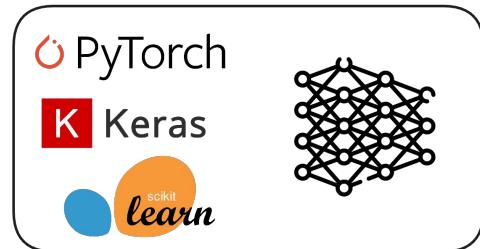
- **Optimisation:** Nvidia Triton is optimized for GPU and CPU performance. This includes support for NVIDIA GPUs for faster computation, which can significantly reduce inference times, especially for deep learning models.
- **Scalability:** It is designed to scale across multiple nodes and GPUs, allowing you to serve high volumes of inference requests efficiently.
- **Cloud and Edge Deployment:** It can be deployed in the cloud, on-premises, or at the edge, providing flexibility depending on your infrastructure needs.
- **Ease of use:** Supports multiple frameworks: TensorRT, TensorFlow SavedModel, ONNX, PyTorch TorchScript
- **Monitoring:** Automatically provides metrics on :8002 port in Prometheus data format which includes GPU utilization, server throughput, latency, and many more.
- **Auto-scaling:** You can control gpu_clusters and model_replicas from within the model configuration files.
- **Online testing:** Deploy multiple version of a model for A/B testing.



ONNX

What is ONNX ?

Open Neural Network Exchange

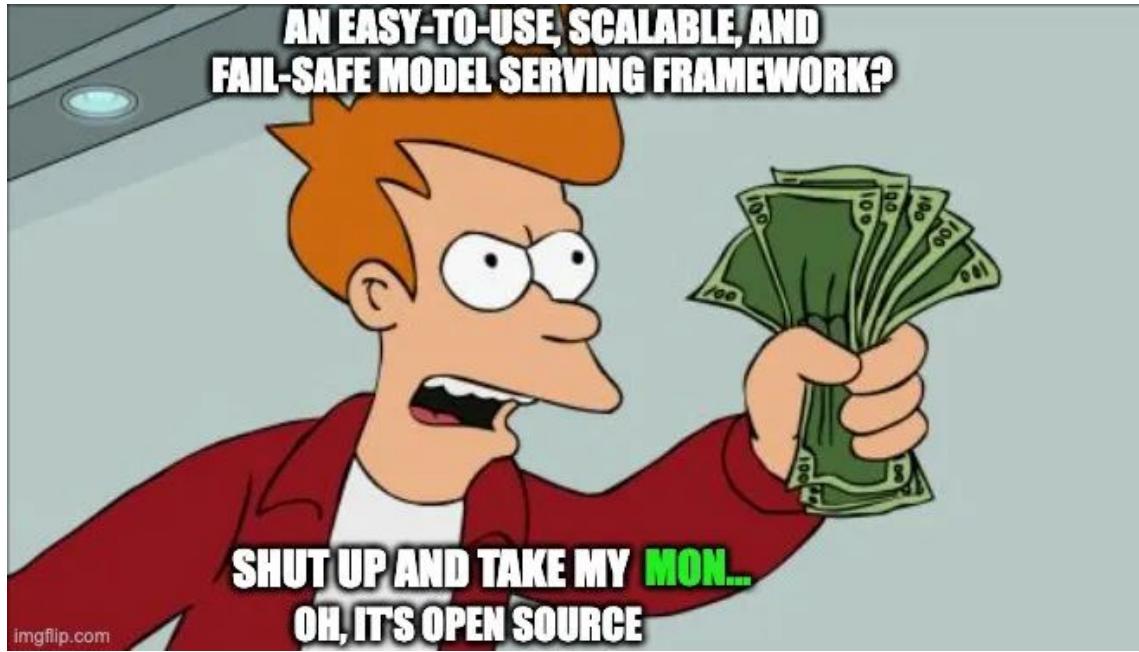


ONNX defines a common **set of operators** - the building blocks of machine learning and deep learning models - and a **common file format** to enable AI developers to use models with a variety of **frameworks, tools, runtimes, and compilers**.

Key benefits:

- **Interoperability:** Develop in your preferred framework without worrying about downstream inferencing implications. ONNX enables you to use your preferred framework with your chosen inference engine.
- **Hardware Access:** ONNX makes it easier to access hardware optimizations. Use ONNX-compatible runtimes and libraries designed to maximize performance across hardware.

What is NVIDIA Triton Inference Server (T.I.S) ?



Zoom on how requests are processed by the Triton server

1. **Client** packs the input data (image, text, audio).
2. **Client** specifies which model to use (by name and version)
3. **Client** sends the inference request to the server via either HTTP or gRPC.
4. **Server** receives a request and places it in a queue as Triton is designed to handle multiple requests simultaneously.
5. **Server** retrieves the specified model from the model repository and performs inference.
6. **Server** sends the response back to the client using the same protocol gRPC or HTTP.
7. **Client** receives the response and extracts the result tensor() → numpy().

Wrap-up

Project milestones

MS 2

Model pipeline is delayed 1 week. It's ok if you show it in the next milestone

MS	Topics	Sprints
1	Present your general <i>use case</i> , the <i>data preparation</i> and the result from your model <i>experimentation</i> .	Sprint 1 & 2
2	Present your architecture for <i>model serving</i> , <i>model deployment</i> and (if possible) <i>model pipeline</i> .	Sprint 3 (& 4)
3	Present your overall project work. You can present a demo of your model/use case and any other topic you think is relevant.	All sprints

Project milestones

Practicals

- The milestone presentation will take 15min (10min presentation + 5min QA)
- MS 2 will be the week of the 31st of March. You have the choice to do it
 - In presence: Monday 31st of march afternoon
 - Online: Google Meets, other afternoons of the same week
- We will send a mail with a link to book a Google Meets slot



MS presentations

Rules of the game

- Share your PR (+ email) *before* your presentation
 - Document in the README what is important for the teaching staff to check!
 - Also put slides!
- Make sure to be ready at the moment it starts
 - All team available
 - Slides ready
 - Mic check
 - Screen sharing check
- Spread the work 
 - Split the speaking time

Project objective for sprint 4

#	Week	Work package	Requirement
4.1	W08	<p>Build a pipeline to automatically run different sequential components such as training your model and deploying your model. For it you can use orchestrated pipeline tools such as Kubeflow Pipelines, AWS Sagemaker or GCP Vertex.</p> <p>Attention: If you run this pipeline in the Cloud it can incur Cloud costs. Make sure to use a platform where you have credits and not burn through them. You can ask for support from the teaching staff in that regard.</p>	Optional

Lecture summary

Topic	Concepts	To know for...	
		Project	Exam
Model serving optimisation	<ul style="list-style-type: none">Optimise different parts of model serving for latency optimisation		Yes
Parallel and Distributed Training	<ul style="list-style-type: none">Data parallelisationModel parallelisationPipeline parallelisation		Yes
Model complexity optimisation	<ul style="list-style-type: none">What is model simplification and why it mattersPruning, quantisation and distillation		Yes

Lecture summary

(split over next week)

Topic	Concepts	To know for...	
		Project	Exam
Triton Inference Server	<ul style="list-style-type: none">• What it is• ONNX• Lab on deploying a MobileNet model on a Triton server		
ML model pipeline	<ul style="list-style-type: none">• What it is• Standard steps of ML pipelines		Yes
ML platforms & orchestrators	<ul style="list-style-type: none">• Sagemaker offerings (workbench, training, registry. Predictions & pipelines)	Possibly (Vertex equivalent)	
Vertex Pipeline	<ul style="list-style-type: none">• Build a vertex pipeline	Possibly	

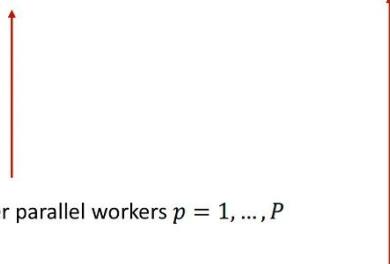
That's it for today!



1. Data parallelism

Given **p worker** devices/machines, and a batch of **K data samples**, the iterative-convergent formula would be:

$$A(t) = F(A(t-1), \sum_{p=1}^P \Delta_{\mathcal{L}}(A(t-1), \mathbf{x}_p))$$



Sum of update functions $\Delta_{\mathcal{L}}$ over parallel workers $p = 1, \dots, P$

\mathbf{x}_p is the subset of data assigned to worker p
Data indices at different workers do not overlap