Serving & training optimisation

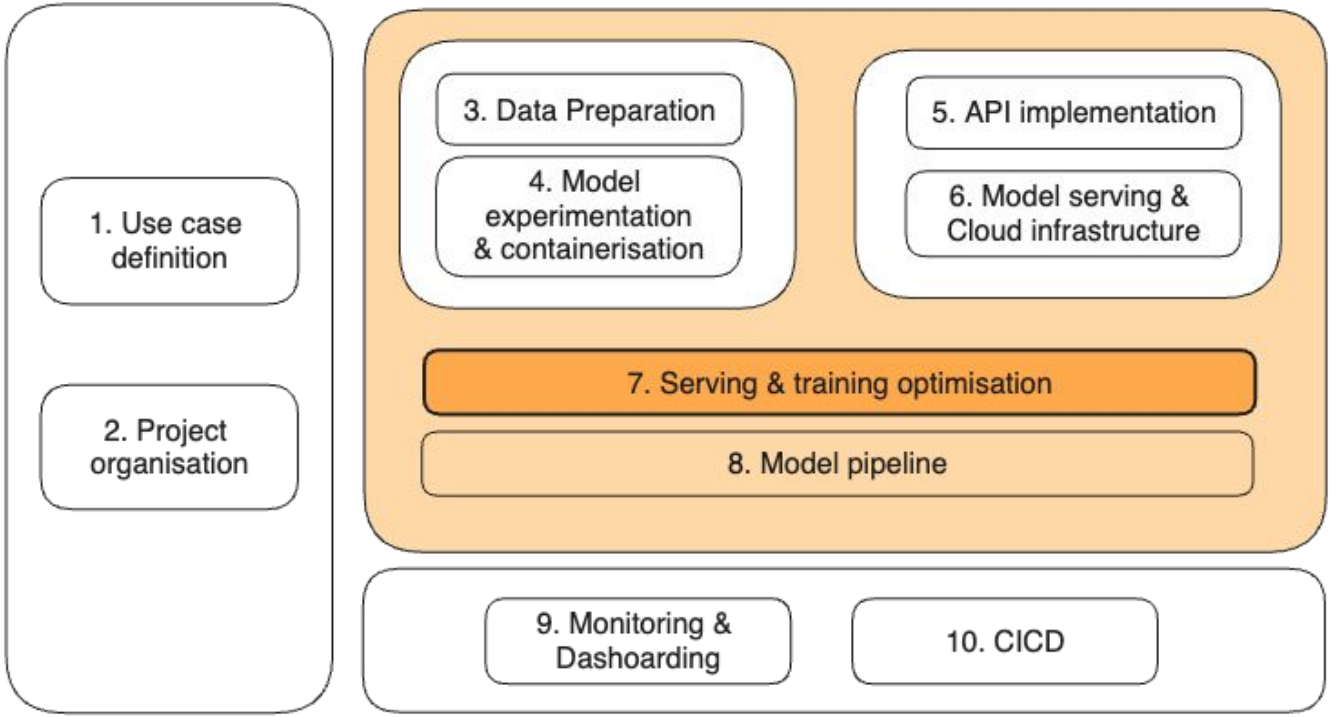
Sprint 4 - Week 7

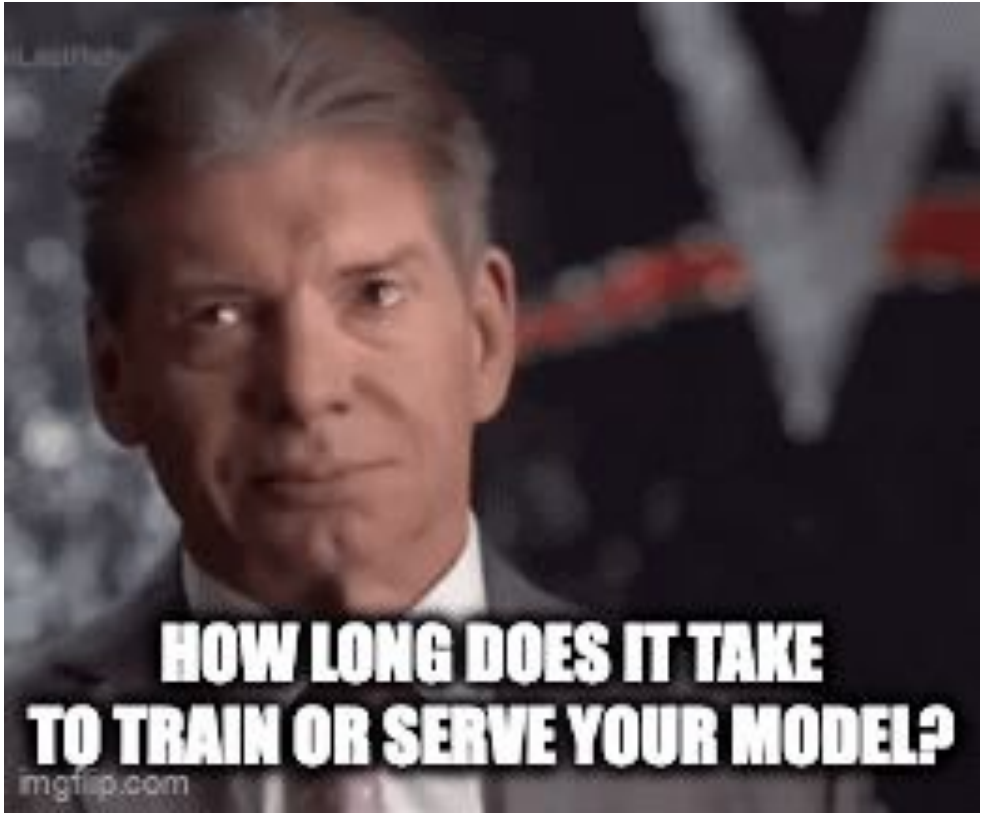
INFO 9023 - Machine Learning Systems Design

2024 H1

Thomas Vrancken (t.vrancken@uliege.be)
Matthias Pirlet (matthias.pirlet@uliege.be)

Status on our overall course roadmap




HOW LONG DOES IT TAKE
TO TRAIN OR SERVE YOUR MODEL?
imgflip.com

Agenda

What will we talk about today

Lab (1 hour)

- Deploy an API in the Cloud

Lecture (45min)

- Model serving optimisation
- Parallel and Distributed Training

Lecture + lab (30min) ⇒ *If enough time*

- Triton Inference Server

Lecture (30min) ⇒ *If enough time*

- Model complexity optimisation

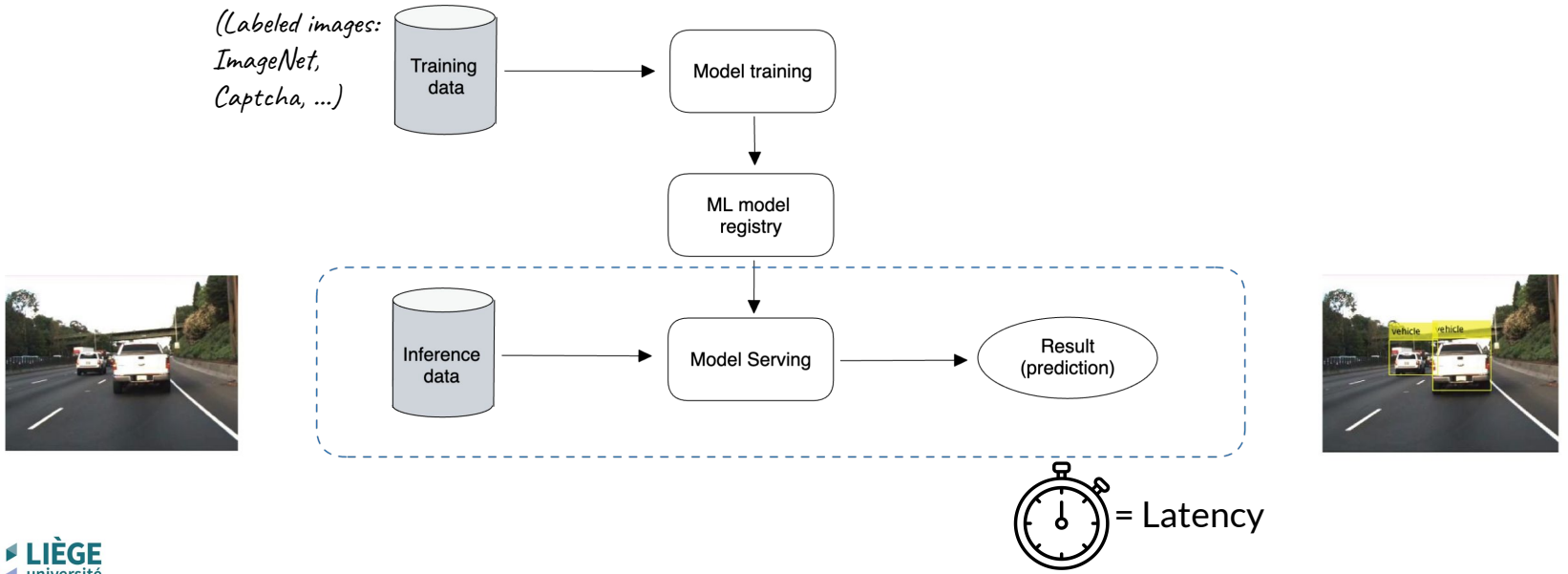
Lab: Deploy an API in the Cloud

Model serving optimisation

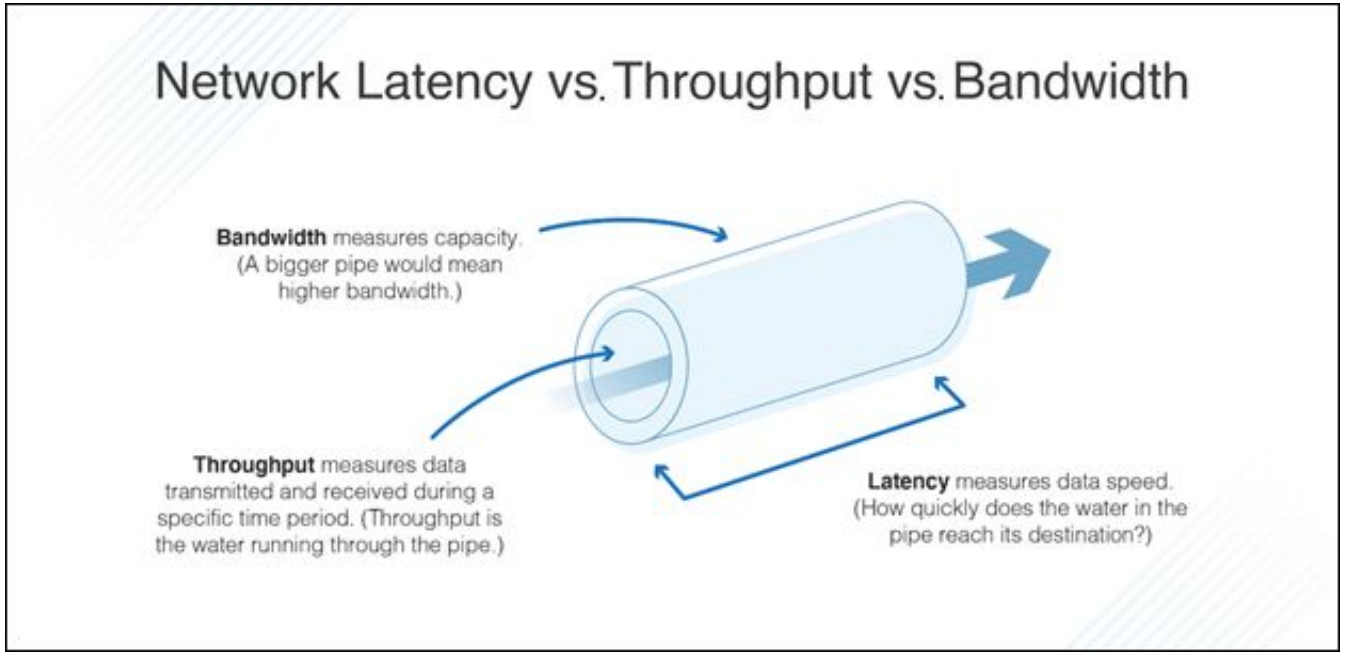
What is model serving and latency?

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



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

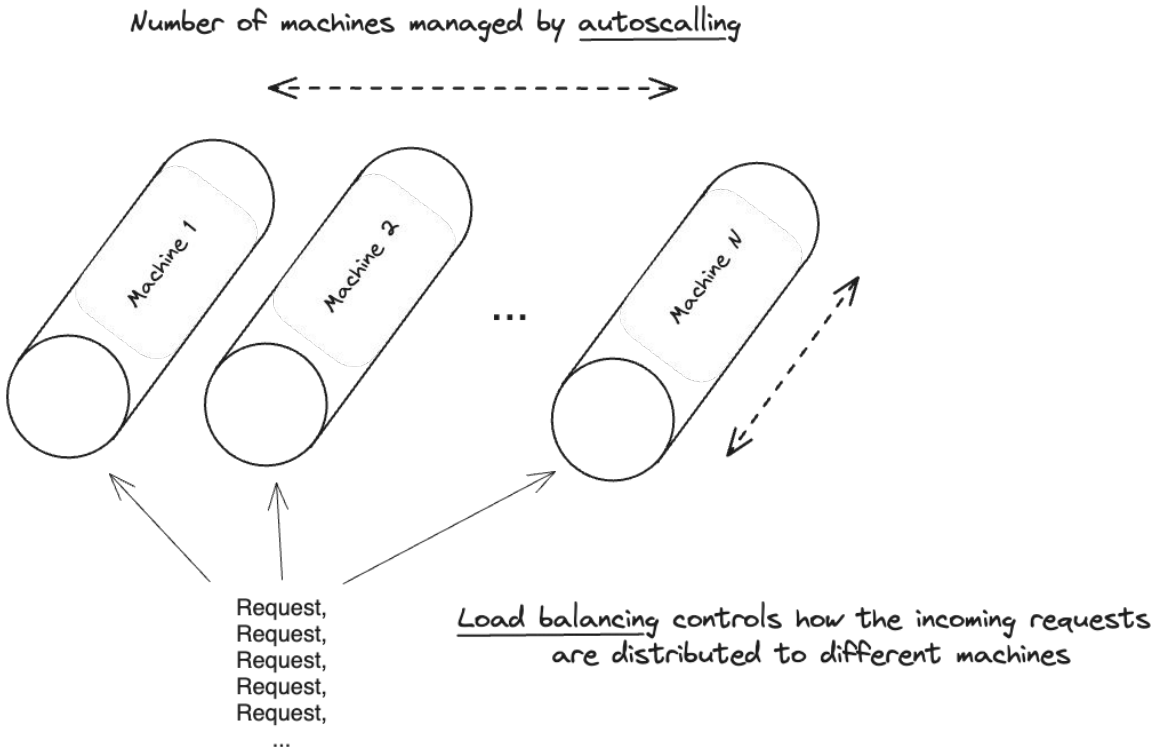
https://shardeum.org/blog/throughput-vs-bandwidth/

Cloud deployment: Autoscalling

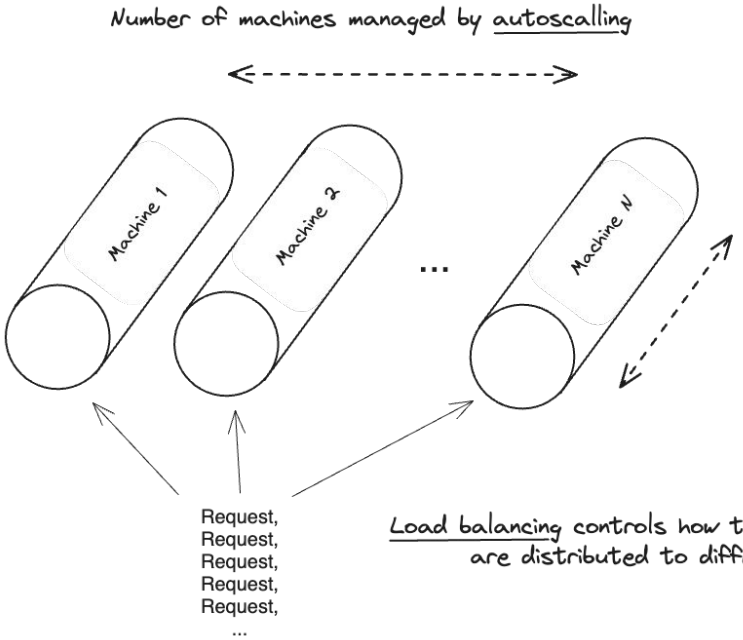
Different Cloud platforms allow for **auto scaling** and **load balancing**.

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

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



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



Helps keeping throughput and bandwidth to a good level

Latency remains the same

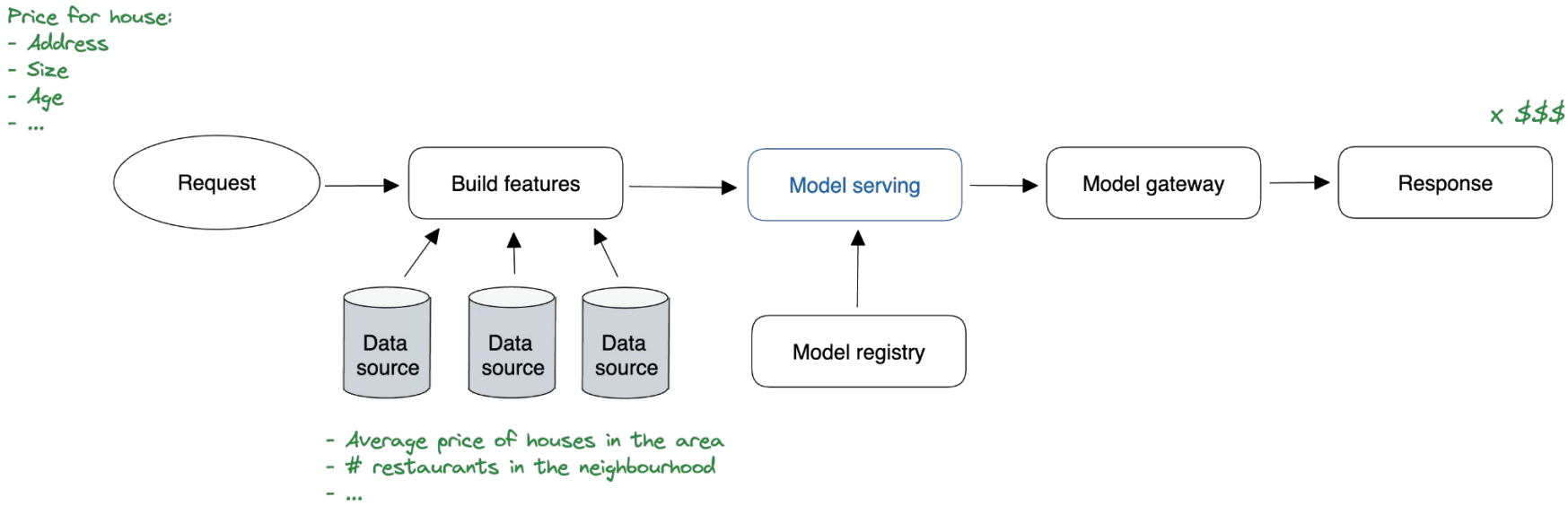
⇒ *Some applications can* ***scale to 0***

Ways of reducing your serving latency without changing your ML model

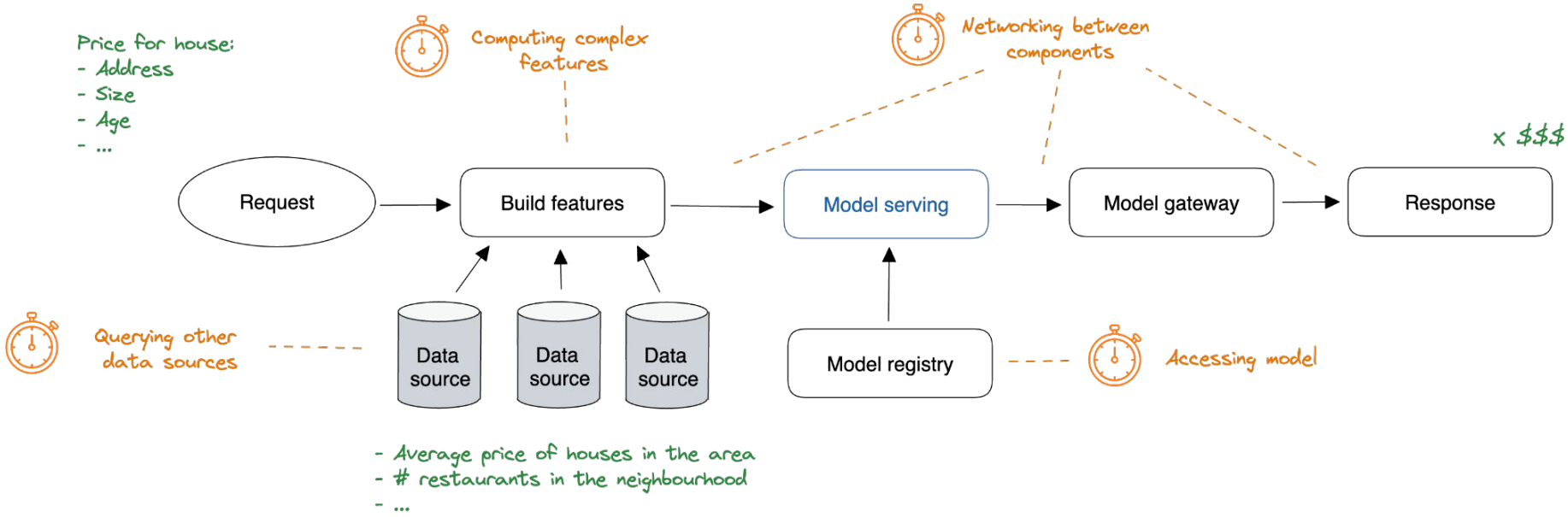
We look at three different methods:

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.

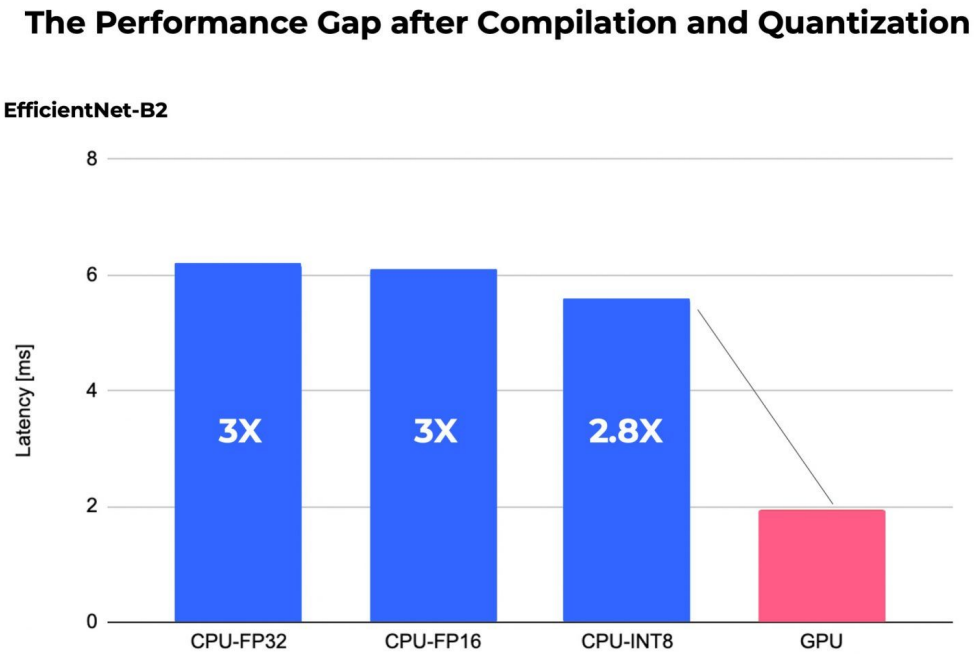


You ML model might not be the bottleneck!



Optimise the hardware used by your ML model.

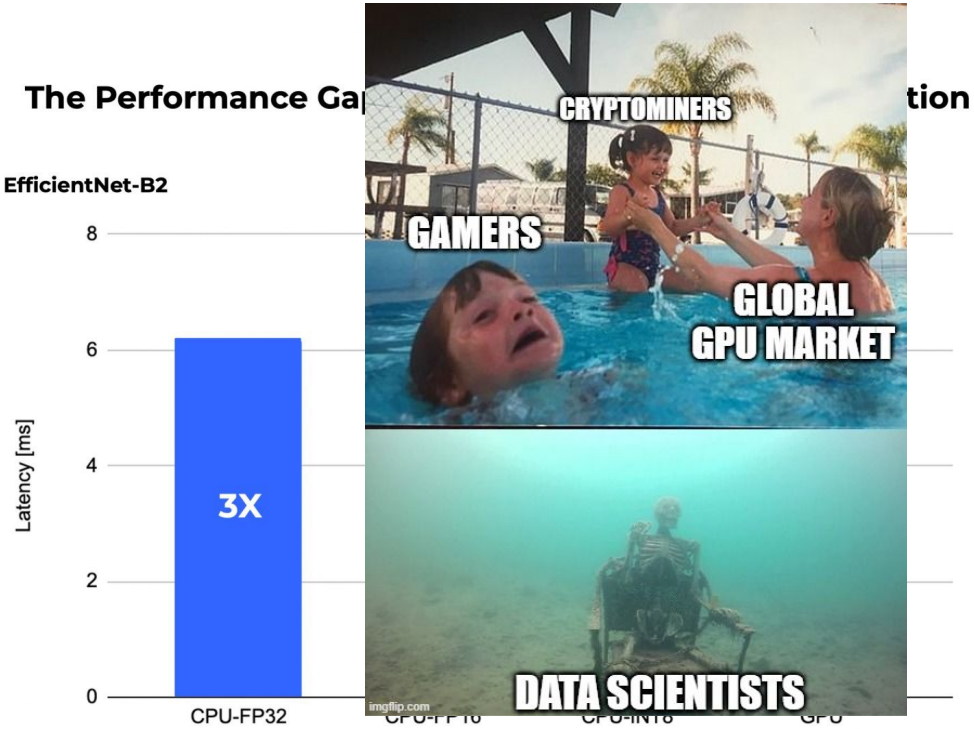
Hardware



Credit: deci.ai

Optimise the hardware used by your ML model.

Hardware



Credit: deci.ai

Tensor Processing Unit (TPU)

Processor	*mm²*	*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

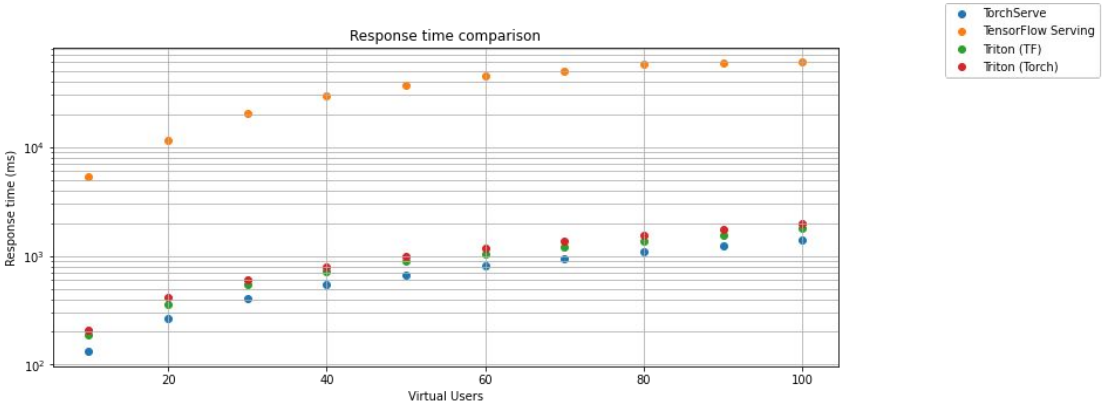
These chips and platforms chosen for comparison because widely deployed in Google data centers

You can wrap your model using different model frameworks.

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

TensorFlow Serving: TensorFlow Serving is a flexible, high-performance serving system for machine learning models designed for production environments created by Google.

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



Credit: biano-ai.github.io

Breakout exercise

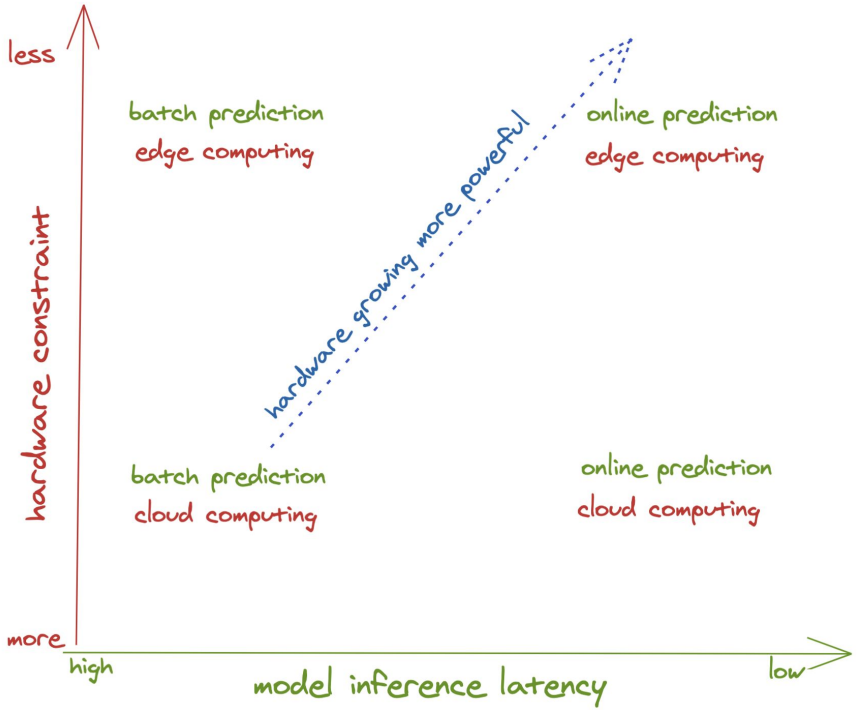
Group of ~4, duration 10 minutes

Identify 3 applications for each quadrant.

How do you determine:

- Batch vs. online prediction
- Edge vs. cloud

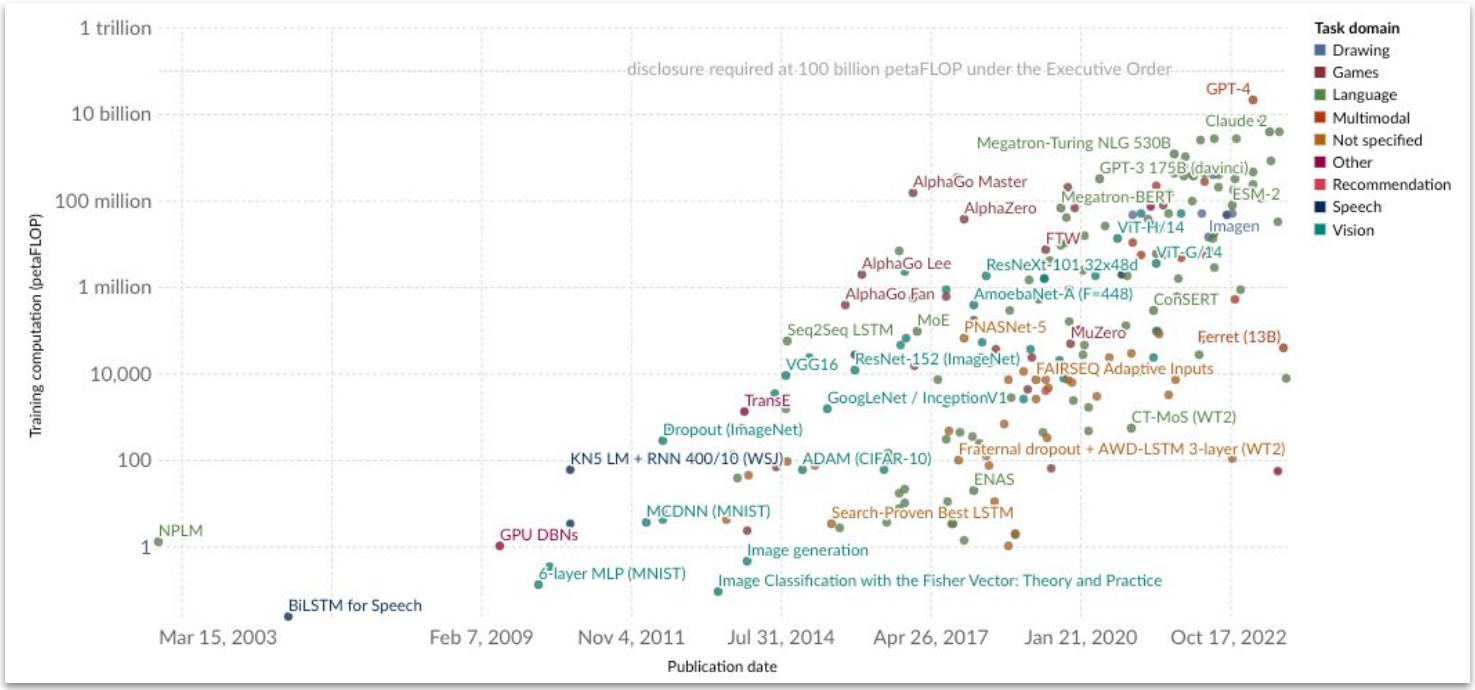
Hints: Look at some of the applications on your phone.



Computational cost of training ML models

Training ML models is often a computationally heavy task

LLMs are pushing the boundaries



https://ourworldindata.org/grapher/artificial-intelligence-training-computation?time=2003-03-15..latest

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 (FLOP / seconds = FLOPS) and the total number of operations for a specific computational job.

PetaFLOP: ~ 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}
quettaFLOPS	QFLOPS	10^{30}

Apple M2 GPU = 3.6 TFLOPS

https://epochai.org/blog/estimating-training-compute

Calculate the size of an ML training job

Method 1: Counting operations in the model

How do we count the number of operations?

$$\text{training\_compute} = (\text{ops\_per\_forward\_pass} + \text{ops\_per\_backward\_pass}) * \text{n\_passes}$$

$$\text{n\_passes} = \text{n\_epochs} * \text{n\_examples}$$

Layer	# parameters	# floating point operations
Fully connected layer from N neurons to M neurons	N*M + M ≈ N*M	2*N*M+M+M ≈ 2*N*M
	WEIGHTS BIASES	NONLINEARITIES

Number of operations in a single forward pass.

https://epochai.org/blog/estimating-training-compute

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…

https://epochai.org/blog/estimating-training-compute

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 is of **125 TeraFLOP**
- **Usage**: Hard to make full usage of the tensor performance, we assume **30%**
- **Time**: Well we then know that it ran for **2500 days** and which each make for **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% x 125e12 FLOPS x 2500 days x 86400 sec/day = **8.1e21 = 8.1e6 PetaFLOP**

https://epochai.org/blog/estimating-training-compute

Parallel and Distributed Training

Why should we look at parallelisation?

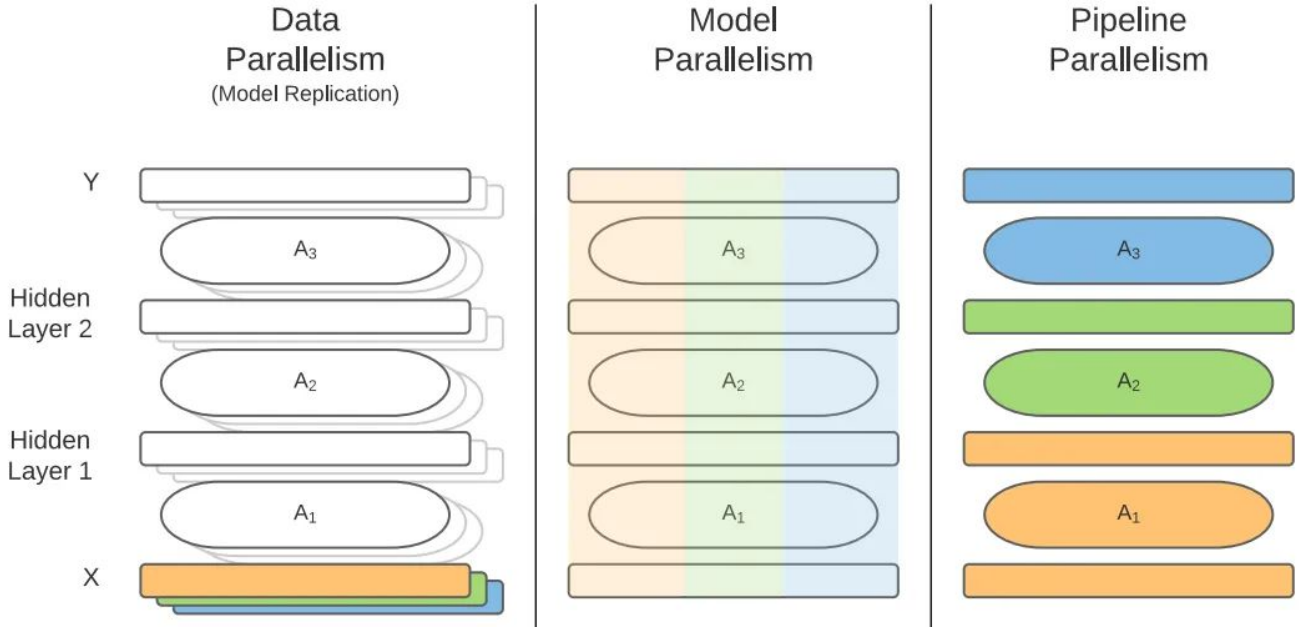
Model training is too slow? You have multiple GPUs? ⇒ Use **parallelisation**

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

How to parallelise this process?

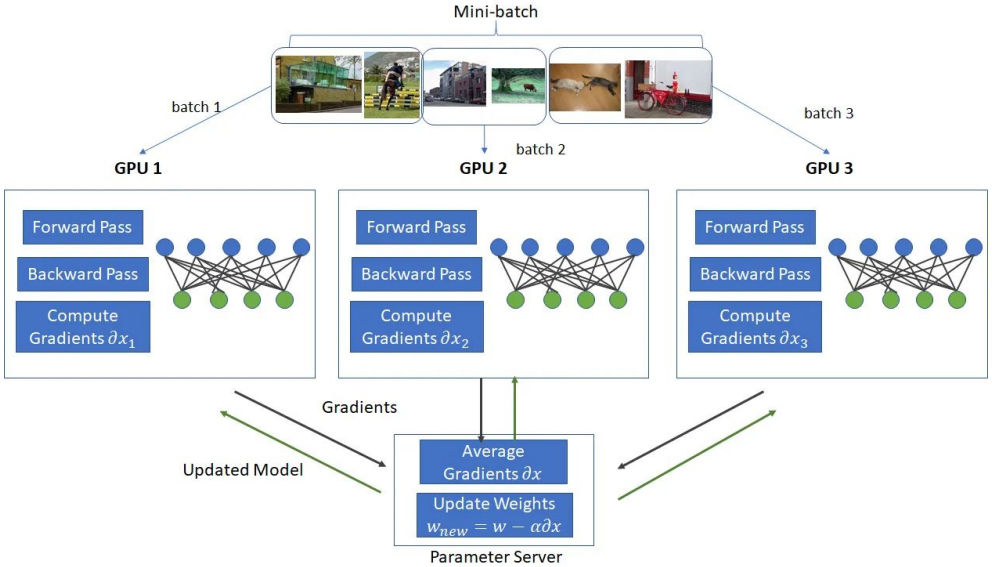
```
# Essential working of ML model training

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
```

Overview of the different methods for model parallel training



https://betterprogramming.pub/parallel-and-distributed-training-in-deep-learning-for-beginners-part-1-introduction-612a4534a117

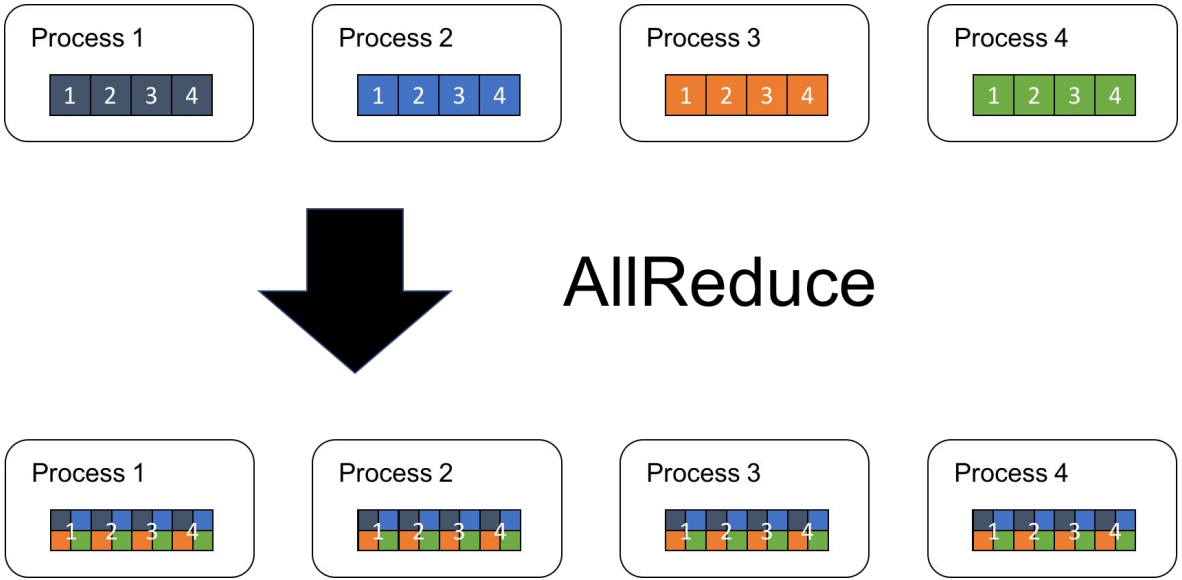
1. Data parallelism



Split a single batch into Split a 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.

https://www.telesens.co/2017/12/25/understanding-data-parallelism-in-machine-learning/

1. Data parallelism



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

https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce/

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 $p = 1, \ldots, P$

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

https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce/

1. Data parallelism

Pros	Cons
• 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).	• 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 • Basically fine for GPUs on the same machine or on a local network

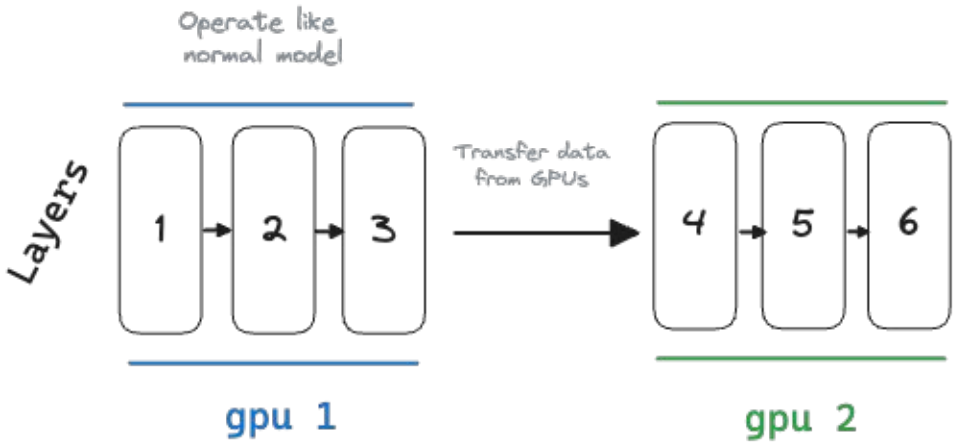
https://betterprogramming.pub/parallel-and-distributed-training-in-deep-learning-for-beginners-part-1-introduction-612a4534a117

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.



https://huggingface.co/docs/transformers/v4.15.0/en/parallelism

2. Naive model parallelism

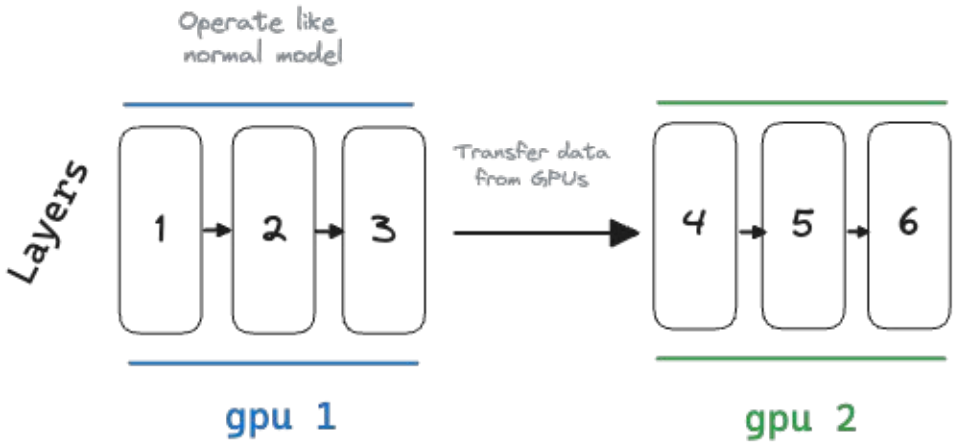
Overhead in copying data:

E.g.

1x 24GB card = 4x 6GB cards using naive MP

↓

Slower due to data transfer



https://huggingface.co/docs/transformers/v4.15.0/en/parallelism

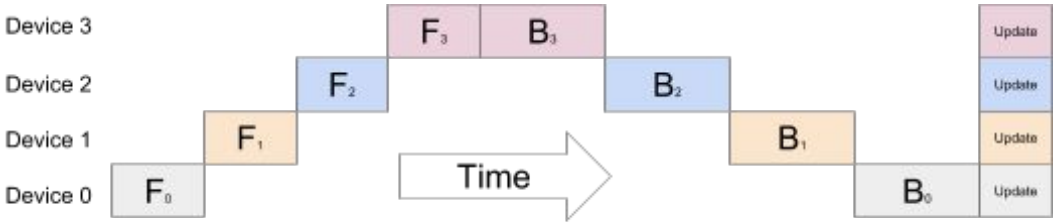
2. Naive model parallelism

Pros	Cons
• Greatly reduced memory usage, proportional to the number of devices used. Super important in the era of Large models • Can use NVIDIA's Megatron-LM library	• 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

https://betterprogramming.pub/parallel-and-distributed-training-in-deep-learning-for-beginners-part-1-introduction-612a4534a117

2. Naive model parallelism

Long idle time of your GPUs.



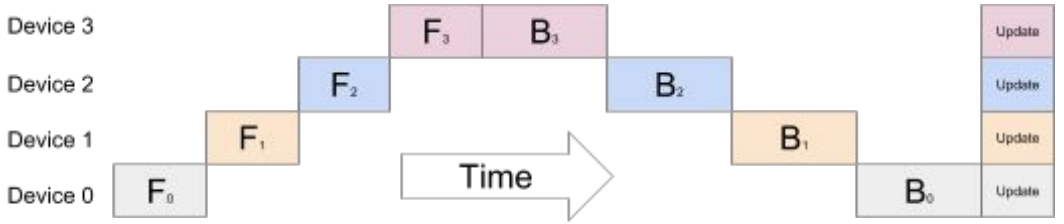
Model parallelism

https://blog.research.google/2019/03/introducing-gpipe-open-source-library.html

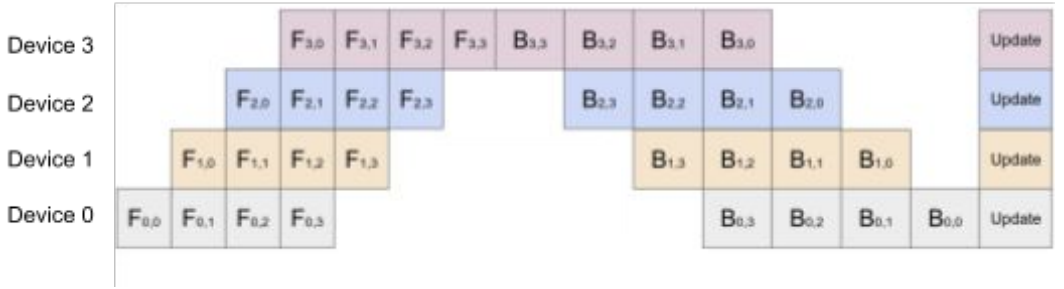
3. Pipeline parallelism

Mix of both previous methods.

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



Model parallelism



Pipeline parallelism

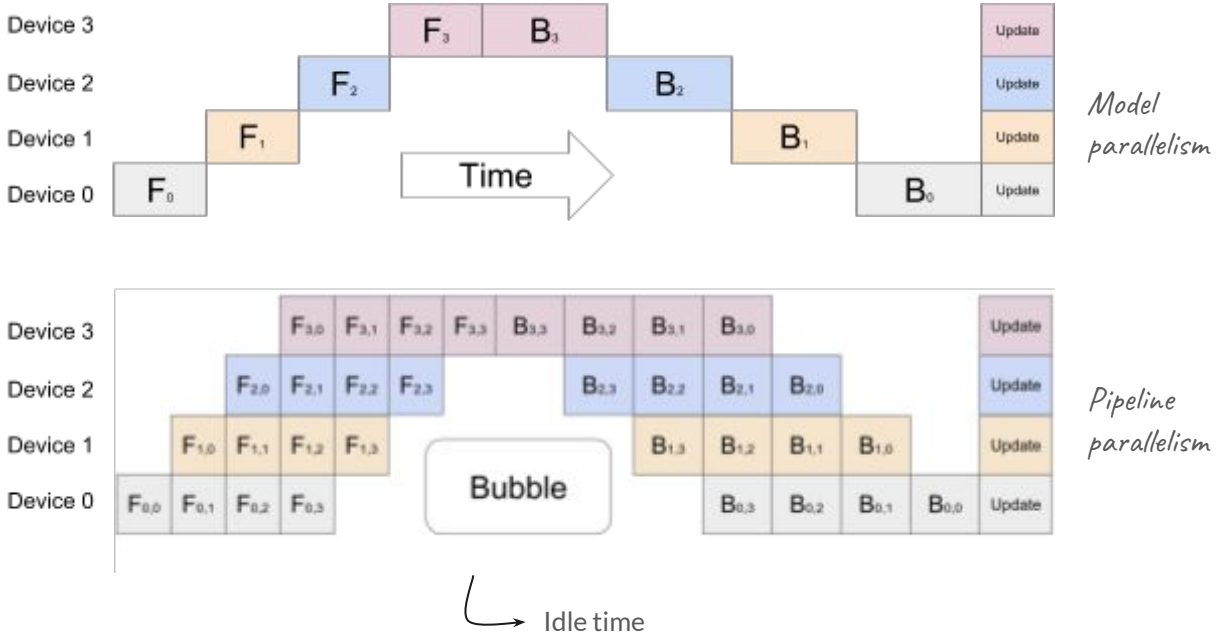
https://blog.research.google/2019/03/introducing-gpipe-open-source-library.html

3. Pipeline parallelism

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

Traditional Pipeline API solutions:

- PyTorch
- FairScale
- DeepSpeed
- Megatron-LM



https://blog.research.google/2019/03/introducing-gpipe-open-source-library.html

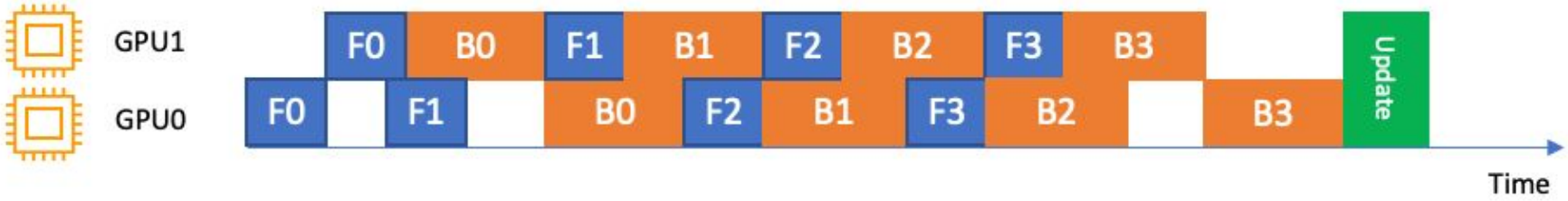
3. Pipeline parallelism

Pros	Cons
• Reduces memory usage **and** GPU idle time	• More complexity (extra MBS 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.

https://huggingface.co/docs/transformers/v4.15.0/en/parallelism

3. Pipeline parallelism

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



Supported by DeepSpeed, Varuna and SageMaker.

https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-core-features.html

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.

https://betterprogramming.pub/parallel-and-distributed-training-in-deep-learning-for-beginners-part-1-introduction-612a4534a117

Ray

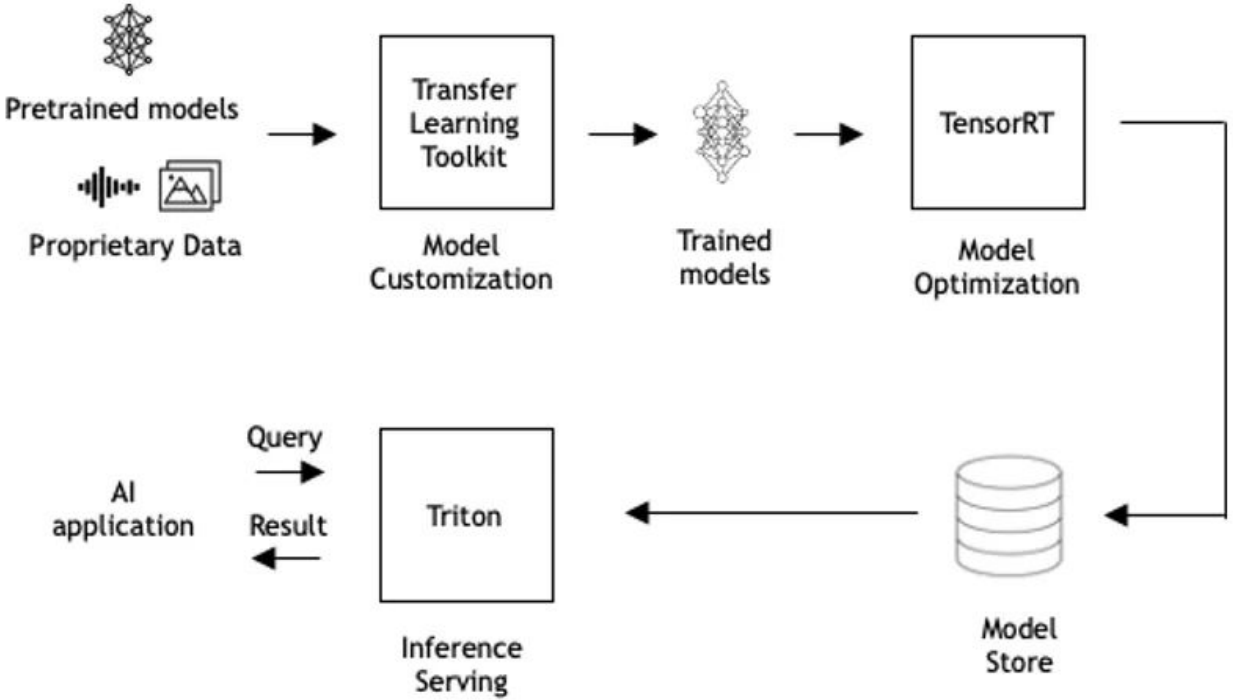
Ray is a framework that provides a simple, universal API to build distributed ML applications. It is designed to scale Python applications from a single computer to a cluster with ease.

- **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.

Lecture: Triton Inference Server

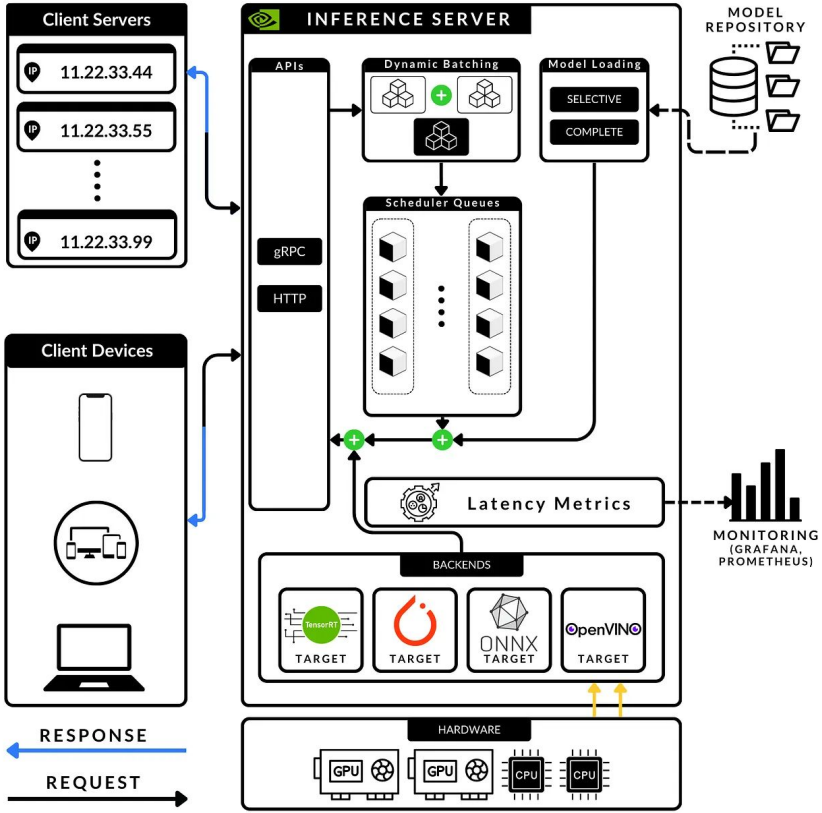
LIÈGE université

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



https://medium.com/@tam.tamanna18/deploying-an-onnx-model-with-triton-inference-server-929484be7d2b

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



https://decodingml.substack.com/p/dml-how-to-deploy-deep-learning-models

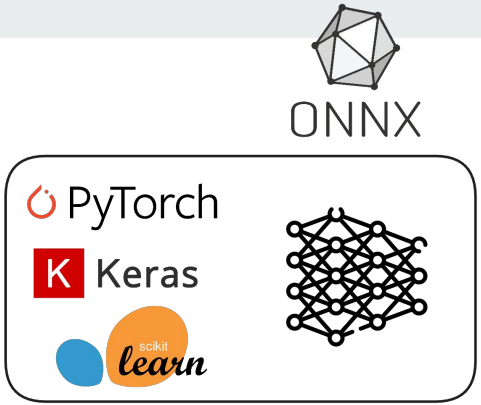
Why use Triton servers ?

- **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.

https://decodingml.substack.com/p/dml-how-to-deploy-deep-learning-models

What is ONNX ?

Open Neural Network Exchange



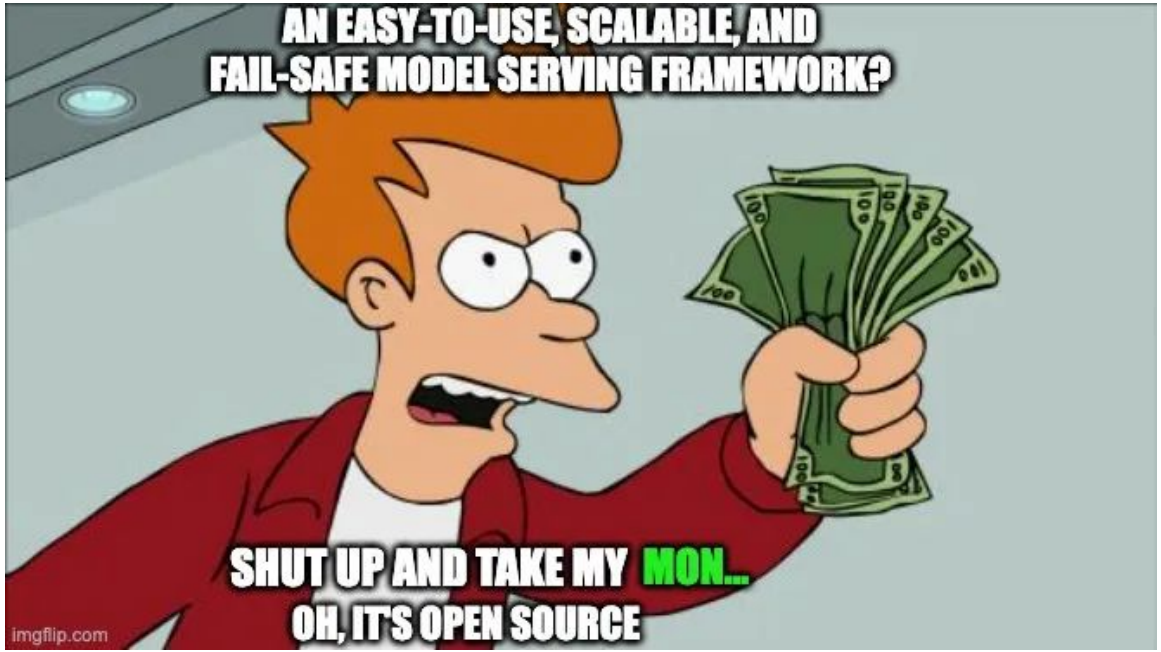
ONNX defines a common **set of operators** - the building blocks of machine learning and deep learning models - and a **common file format** to enable AI developers to use models with a variety of **frameworks**, **tools**, **runtimes**, and **compilers**.

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.

https://onnx.ai/

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



https://decodingml.substack.com/p/dml-how-to-deploy-deep-learning-models

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().

Lab: Triton Inference Server

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.

VS

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.

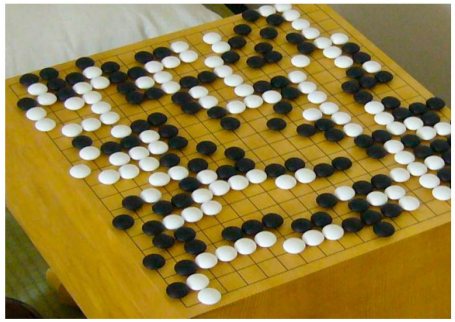
Clear objective: What are your speed requirements?

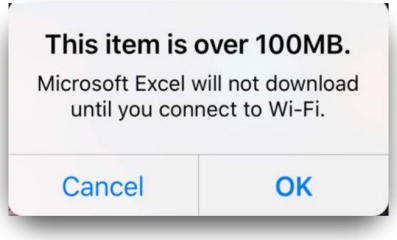
- Satisficing metrics: E.g. prediction latency, cost, … ⇒ Set a threshold
- Optimising metrics: Accuracy, precision, recall, … ⇒ Optimise

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

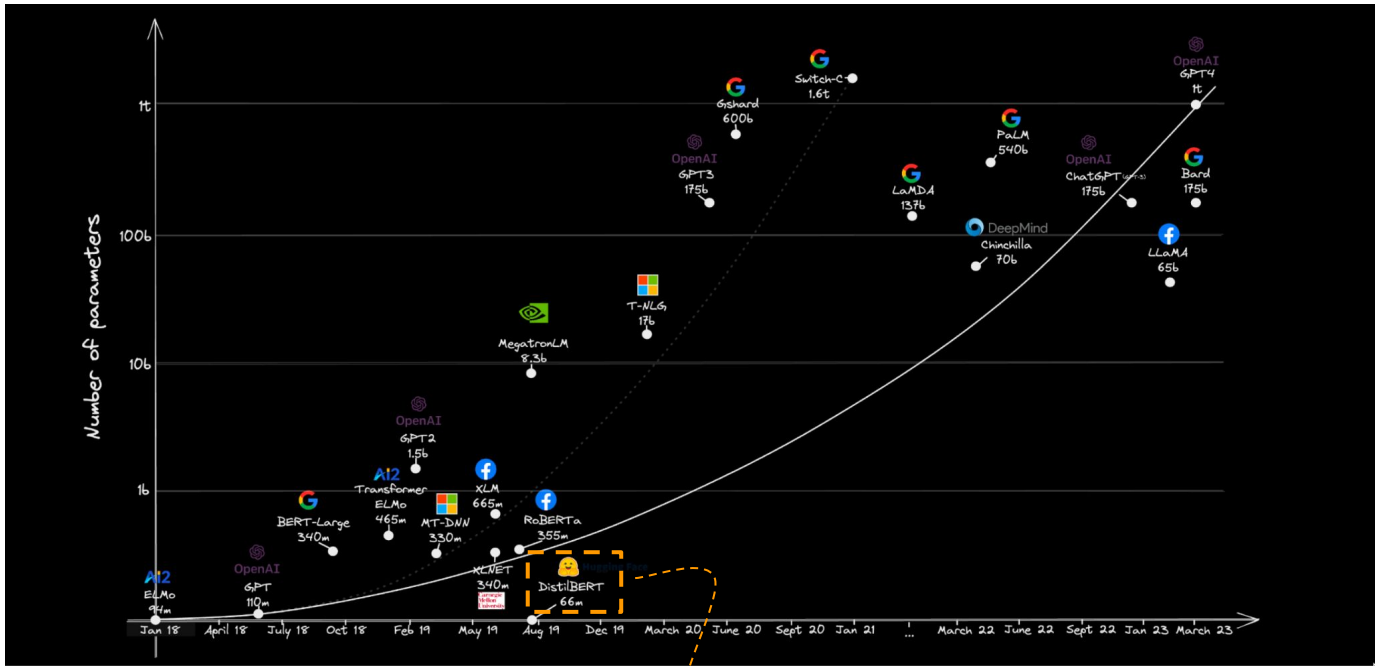
- Model speed
  - *E.g. not detecting pedestrian in time*
- Model size
  - *E.g. uploading/updating model over network*
- Energy efficiency
  - *E.g. AlphaGo: 1920 CPUs and 280 GPUs*

⇒ $3000 electric bill per game





Optimising your model can be done through model optimisation.



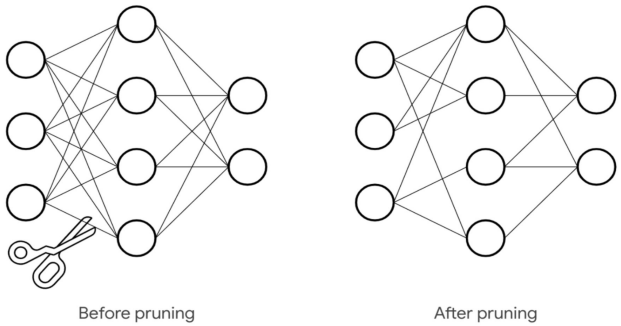
If you still want a large model, there are ways of making it more efficient

1. Pruning
2. Quantisation
3. Distillation

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.

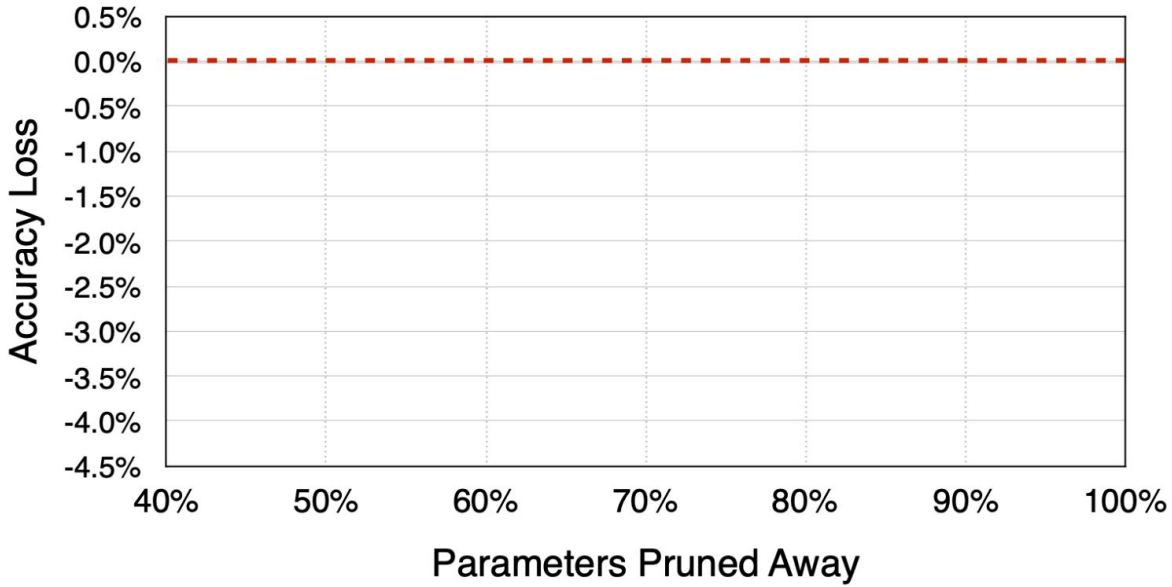


Model	Non-sparse accuracy (float)	Sparse accuracy (float)	Number of non-zero parameters (NNZ) in sparse models
InceptionV3	78.1%, 27.1M parameters	78.0% @ 50% sparsity	13.6M
		76.1% @ 75% sparsity	6.8M
		74.6% @ 87.5% sparsity	3.3M
GNMT EN-DE	26.77 BLEU, 211 M parameters	26.86 BLEU @ 80% sparsity	44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

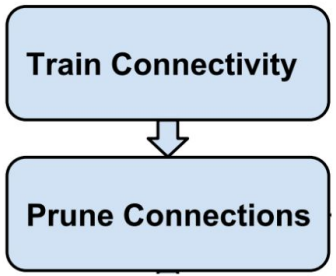
TensorFlow Model Optimization Toolkit

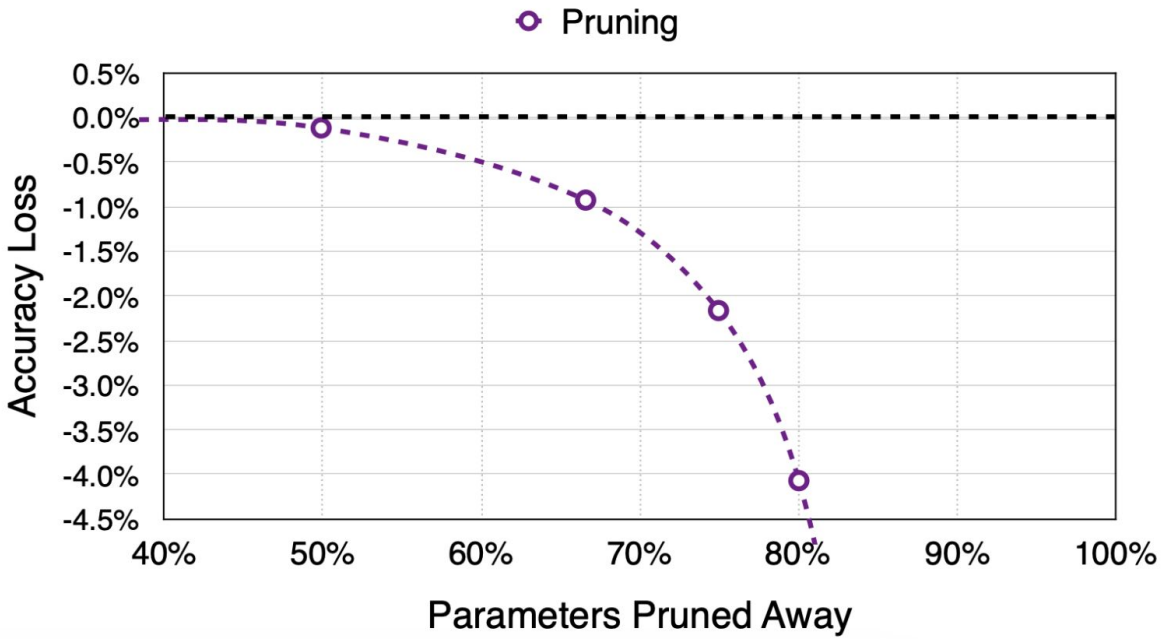
Pruning for model compression.

Train Connectivity

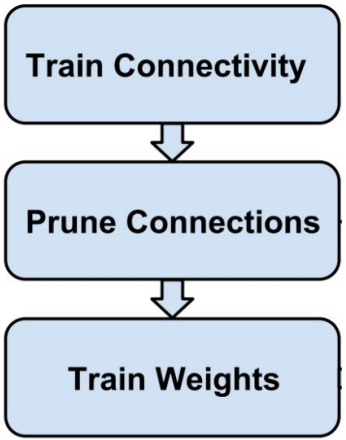


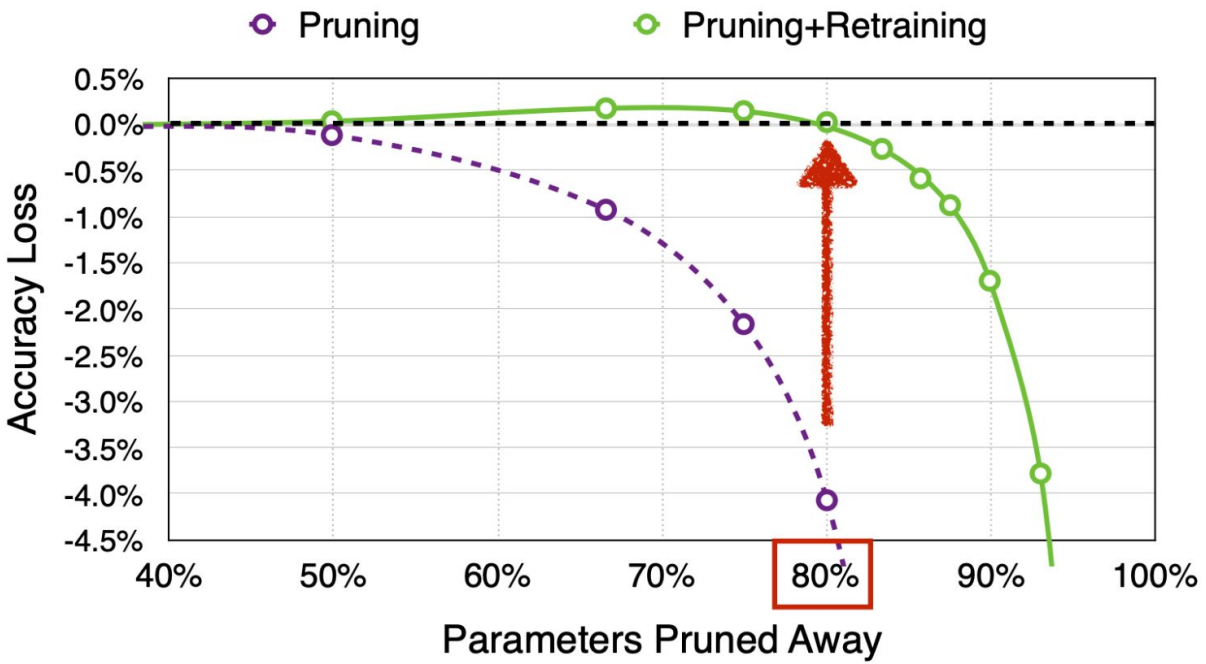
Pruning for model compression.



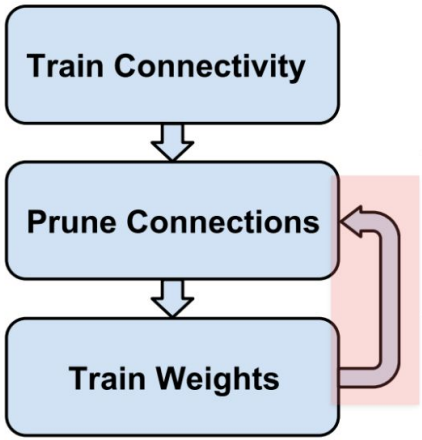


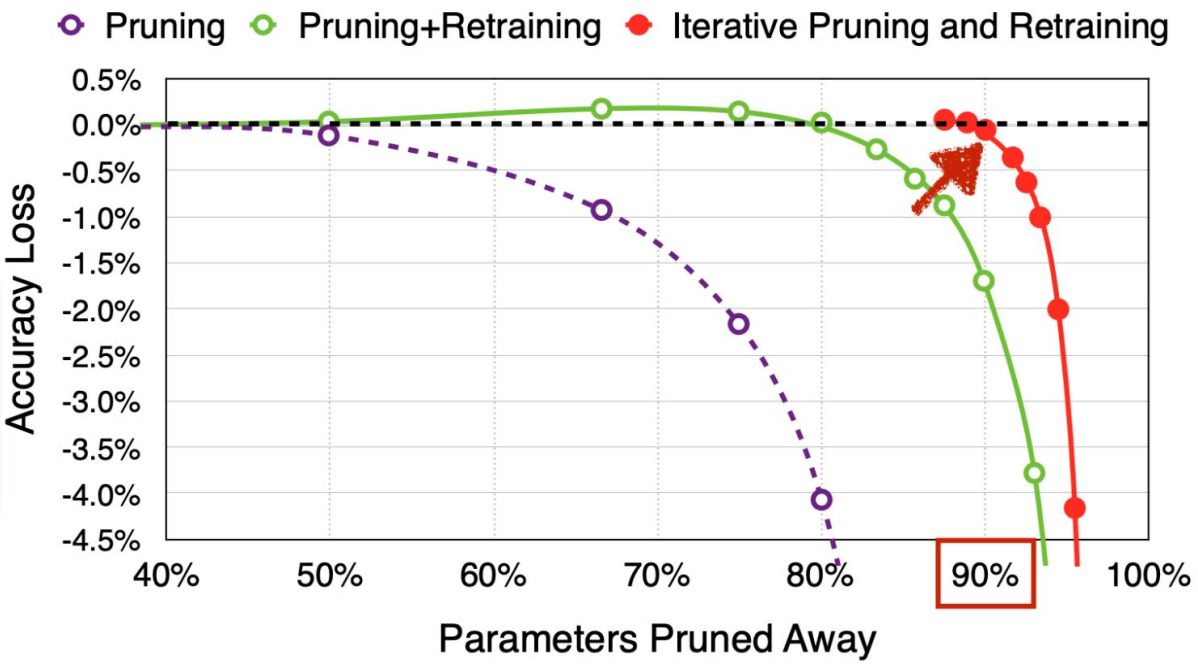
Pruning for model compression.





Pruning for model compression.





Pruning happens in human brain.

50 Trillion Synapses

This image is in the public domain

Newborn

1000 Trillion Synapses

This image is in the public domain

1 year old

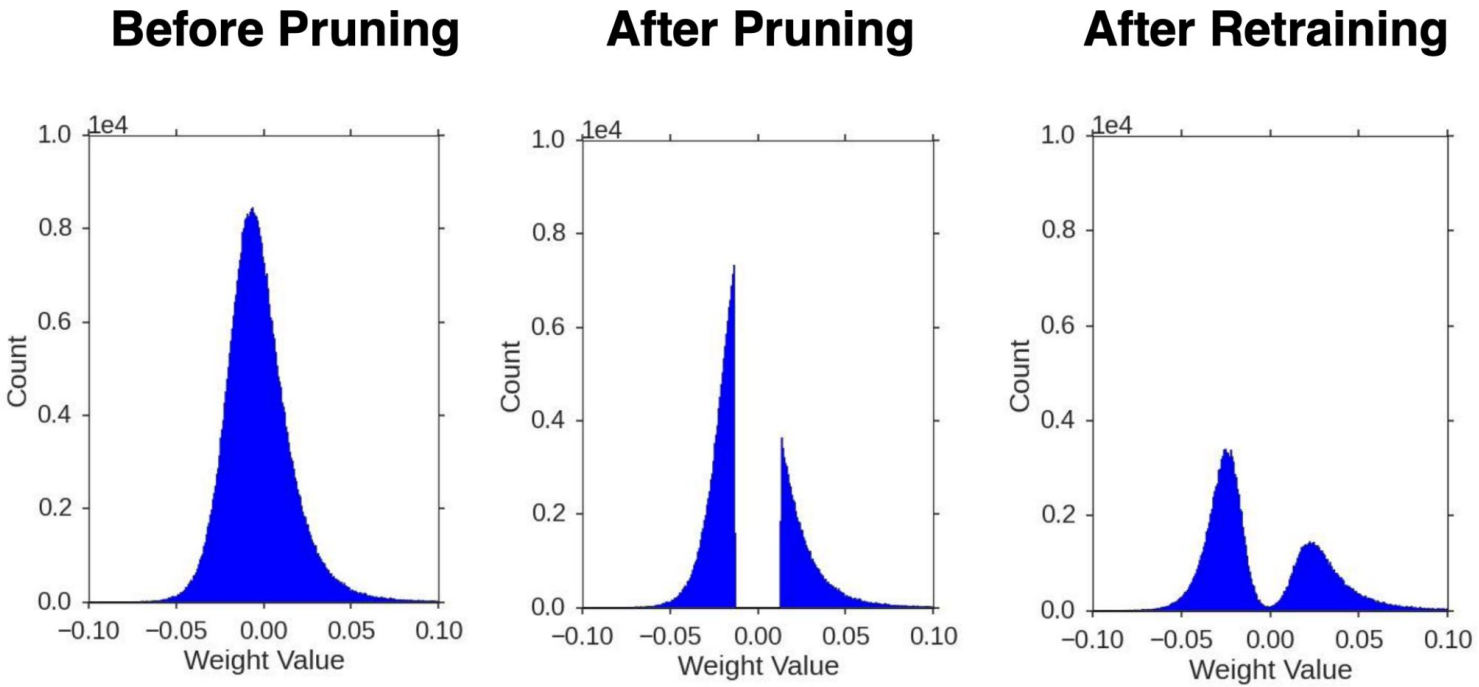
500 Trillion Synapses

This image is in the public domain

Adolescent

Christopher A Walsh. Peter Huttenlocher (1931-2013). Nature, 502(7470):172–172, 2013.

Pruning changes weight distributions.

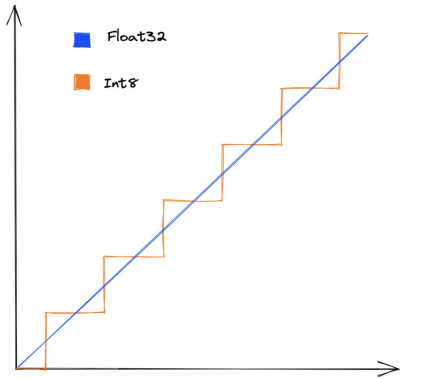


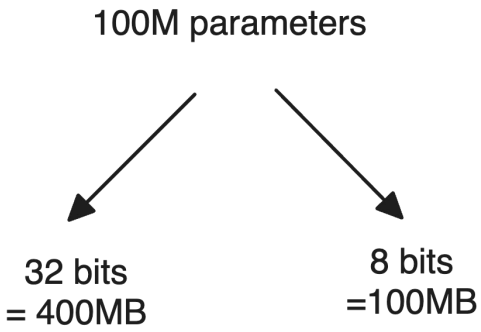
Christopher A Walsh. Peter Huttenlocher (1931-2013). Nature, 502(7470):172–172, 2013.

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

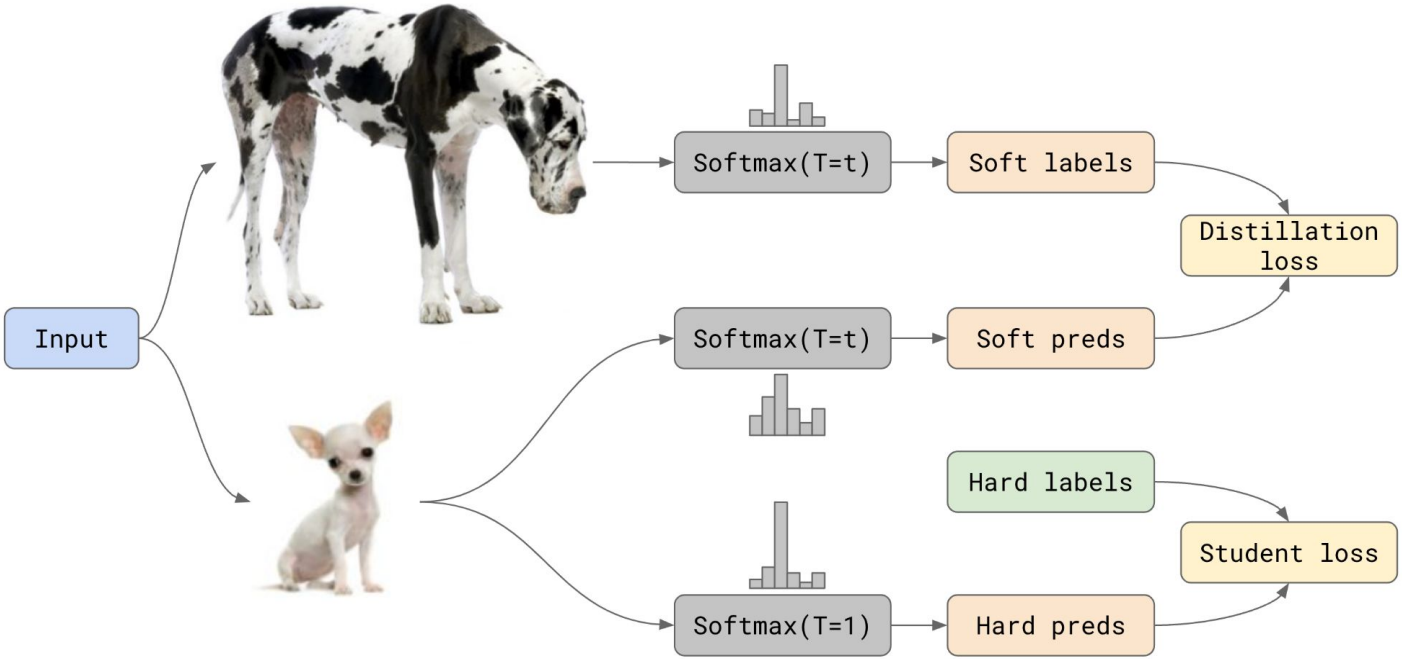
Model	Top-1 Accuracy (Original)	Top-1 Accuracy (Post Training Quantized)	Top-1 Accuracy (Quantization Aware Training)	Latency (Original) (ms)	Latency (Post Training Quantized) (ms)	Latency (Quantization Aware Training) (ms)	Size (Original) (MB)	Size (Optimized) (MB)
Mobilenet-v1-1-224	0.709	0.657	0.70	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
• Reduce memory footprint • Increase computation speed • Bigger batch size • Computation on 16 bits is faster than on 32 bits	• 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.



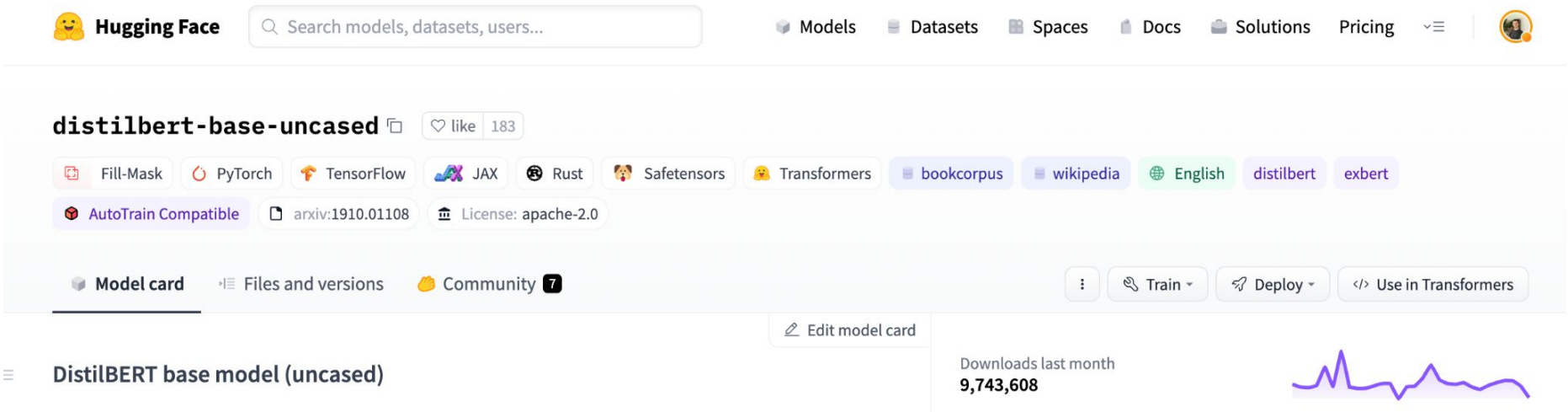
Interesting example: DistilBERT

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

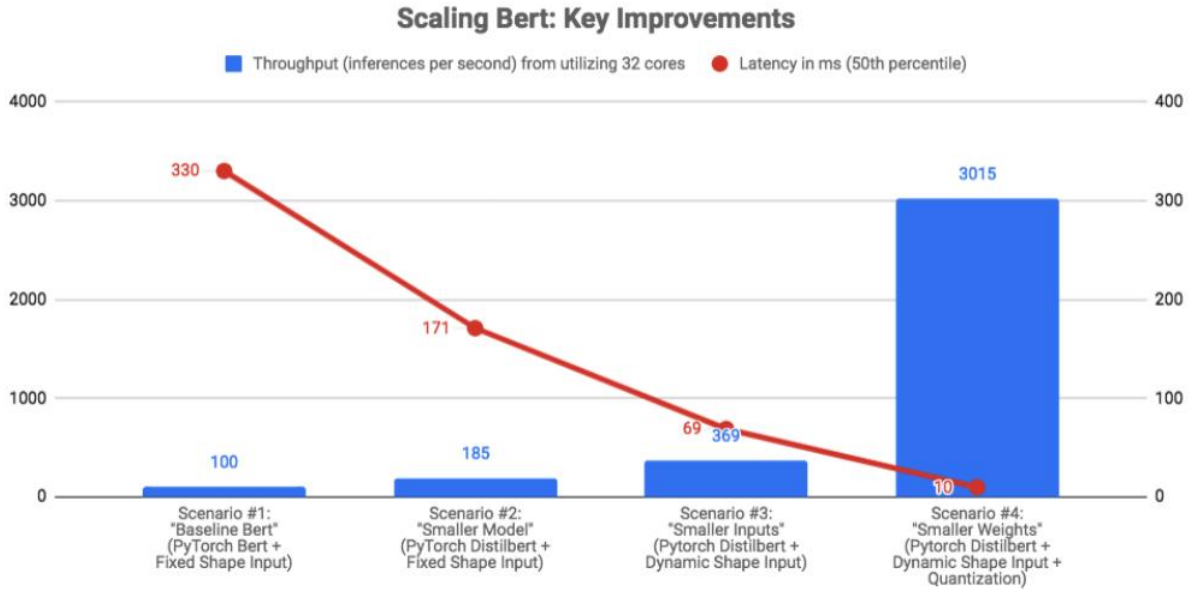
Pros	Cons
● Fast to train student network if teacher is pre-trained. ● Teacher and student can be completely different architectures.	● If teacher is not pre-trained, may need more data & time to first train teacher. ● Sensitive to applications and model architectures.

Interesting example: DistilBERT

DistilBERT is a good example of distilled model.



Improvement in terms of latency and throughput is significant.



Wrap-up

Lecture summary

Topic	Concepts	To know for…	
		Project	Exam
Model serving optimisation	• Optimise different parts of model serving for latency optimisation		Yes
Parallel and Distributed Training	• Data parallelisation • Model parallelisation • Pipeline parallelisation		Yes
Lab: Deploy an API in the Cloud	• Use Cloud Run and GCP to deploy a Flask API in the Cloud	Yes	
Triton Inference Server	• What it is • ONNX • Lab on deploying a MobileNet model on a Triton server		
Model complexity optimisation	• What is model simplification and why it matters • Pruning, quantisation and distillation		

Project objective for sprint 4

Only optional work packages this sprint. Catch up work from last sprint.

⚠️ **Last two steps can incur cloud costs! Make sure to not use anything you will be charged for.** ⚠️

#	Week	Work package	Required
4.1	W07	Package your **model training script** in a **Docker container**. You should be able to run it locally.	Optional
4.2	W07	Run your model training as a job in the Cloud. You can implement this in different ways: ● Containerise your training script and run it on a VM in the Cloud (e.g. on EC2 or on Cloud Run, example,) ● Use a managed service such as Vertex Training or Sagemaker Training	Optional
4.3	W08	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, Docker Compose, AWS Sagemaker, GCP Vertex.	Optional