AI Done Quick

Joe Early, AI Research Engineer, Helsing

AI for Ukraine Recovery Hackathon, 7 September 2024
Tallinn, Estonia

Introduction

Creating and deploying AI models can take a lot of time

- When developing AI models, the cost of training often receives the most attention (on the order of days/weeks/months)

- However, once a model is deployed, time spent on inference quickly outpaces that of training, despite inference being very fast

- There is also the time taken to iterate on different model designs, tune hyperparameters, implement and test the code, etc.

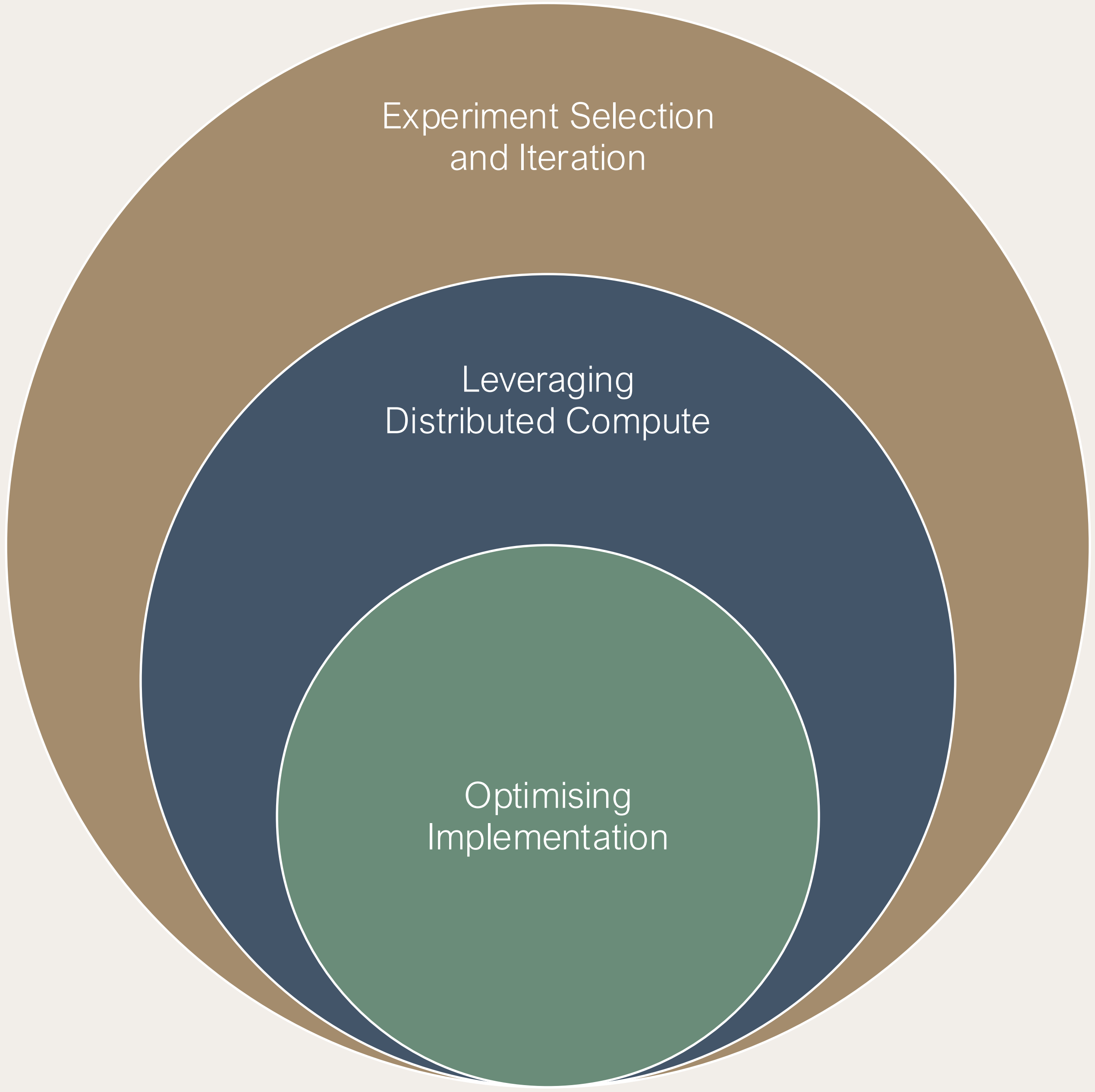
Motivation

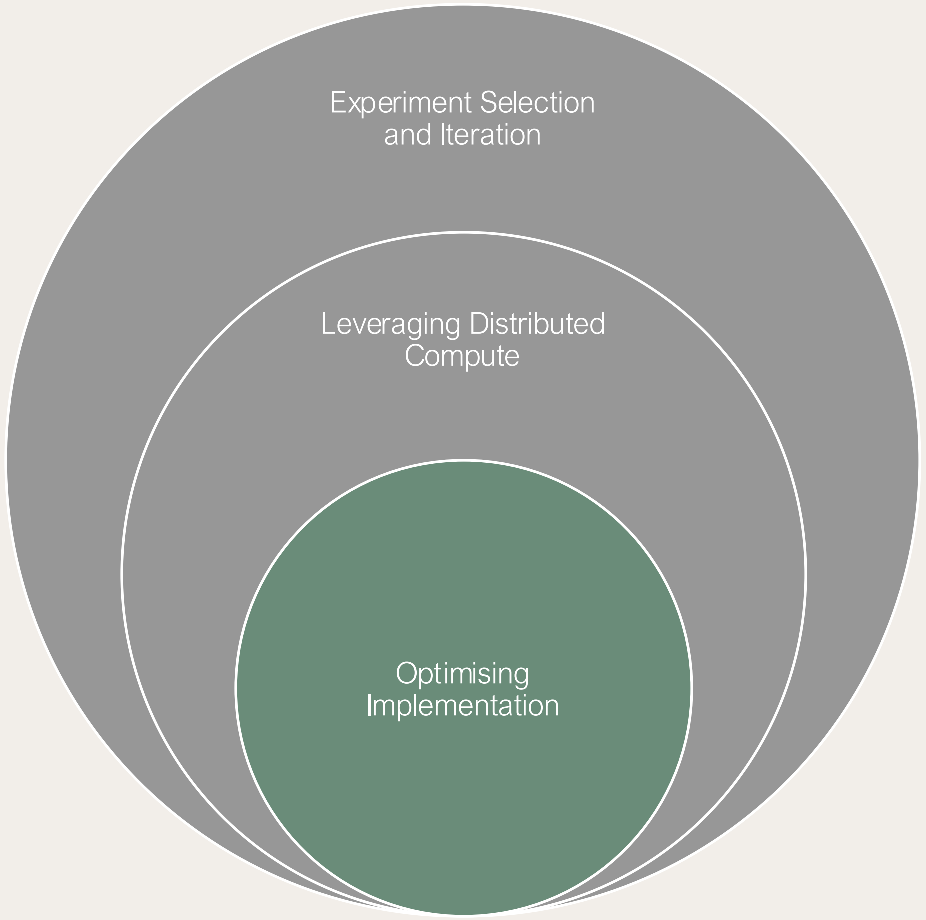
If you can reduce the time it takes to do training, evaluation, and inference…

- You can reduce GPU time, and thus reduce cost
- You can iterate and explore more ideas in the same amount of time
- You don’t have to wait as long for results!

So how can AI be done quick?

Overview


Experiment Selection and Iteration
Leveraging Distributed Compute
Optimising Implementation

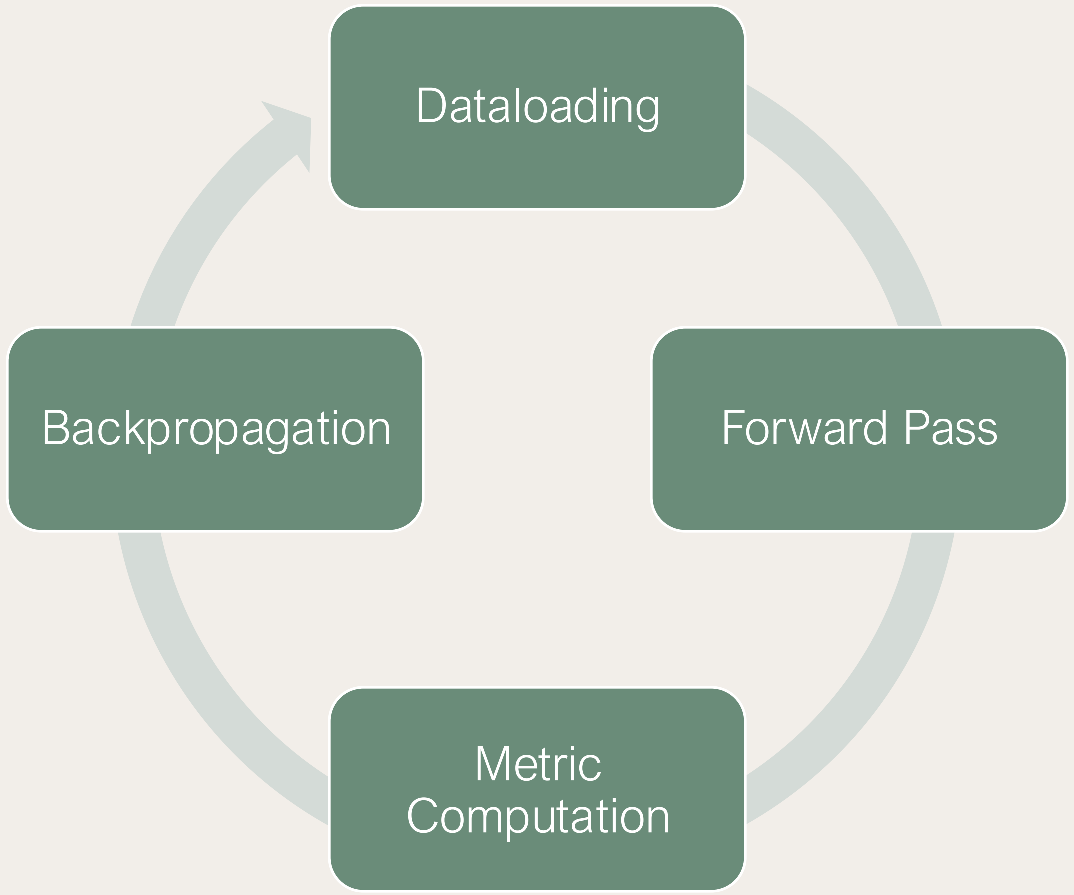

Experiment Selection and Iteration
Leveraging Distributed Compute
Optimising Implementation

01

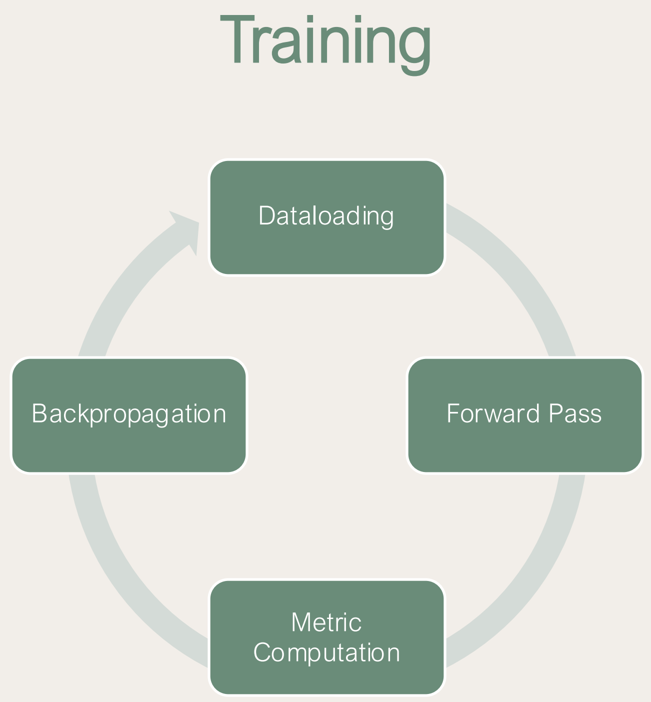
Optimising Implementation

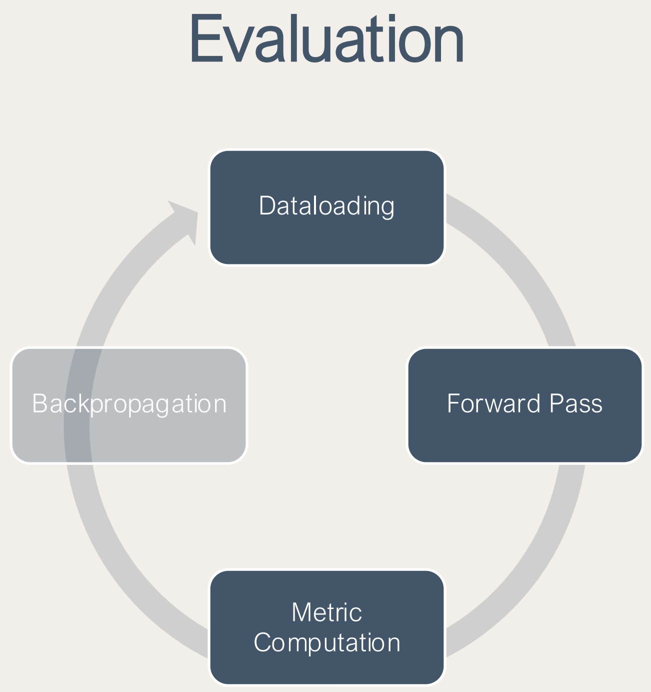
The Core Machine Learning Loop

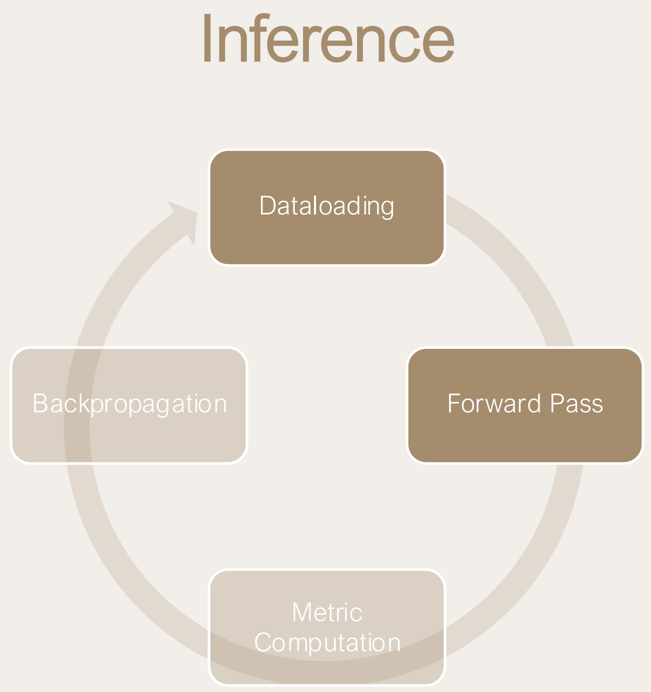
- The same cycle is used for the vast majority of machine learning models
- Each part of the cycle has an associated time cost, the balance of which depends on your model/problem/data
- If we can speed up this cycle, we can speed up all stages* of AI development and deployment



All Stages*


Training
Dataloading
Forward Pass
Metric Computation
Backpropagation


Evaluation
Dataloading
Forward Pass
Metric Computation
Backpropagation


Inference
Dataloading
Forward Pass
Metric Computation
Backpropagation

Attributing Time

As an example…

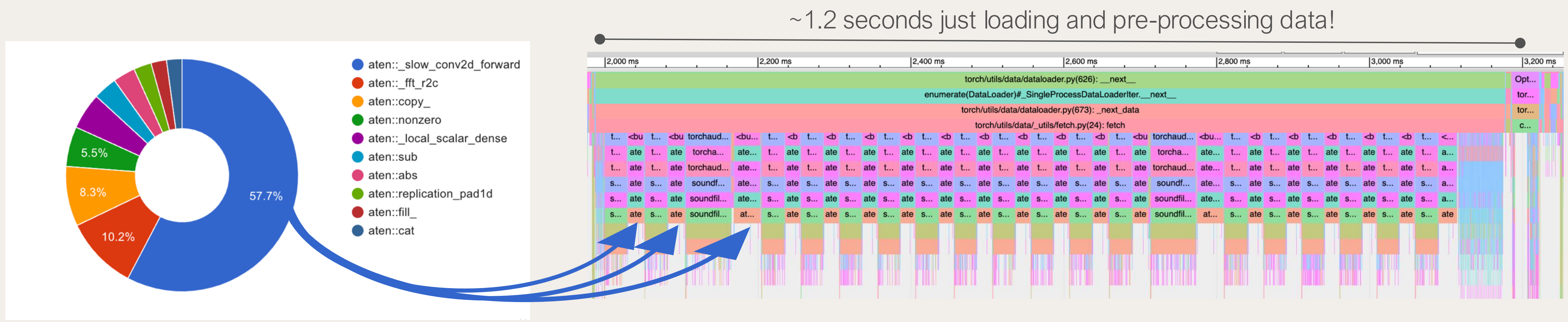
Stage	Pipeline A (ms)	Pipeline B (ms)
Dataloading	850 (92%)	120 (38%)
Forward Pass	70 (8%)	170 (54%)
Metric Computation	5 (<1%)	25 (8%)
Backpropagation	2 (<1%)	2 (<1%)
Total	927	317

For Pipeline A, the biggest bottleneck is loading data, but for Pipeline B it is the forward pass of the model.

Identifying Bottlenecks – An Example

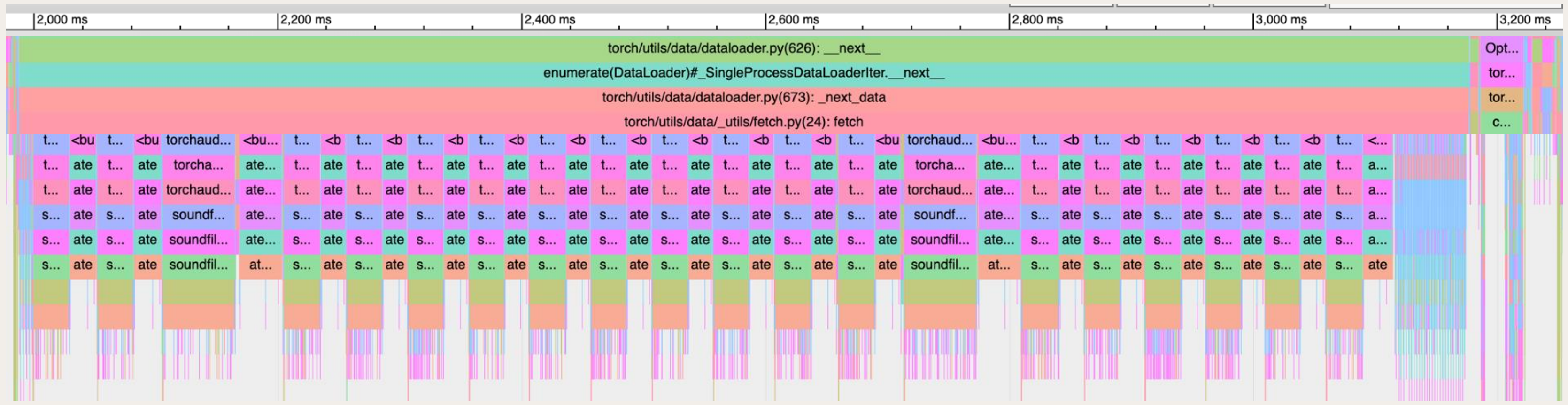
Profiling is an invaluable tool for understanding the expensive parts of your pipelines (e.g. using TensorBoard PyTorch Profiler)

- This allows you to 1) see the contribution of different calls, and 2) identify where these calls occur
- In one of our pipelines, 57.7% of the core loop was spent on calling *slow_conv2d_forward*, and profiling showed this was during the dataloading process

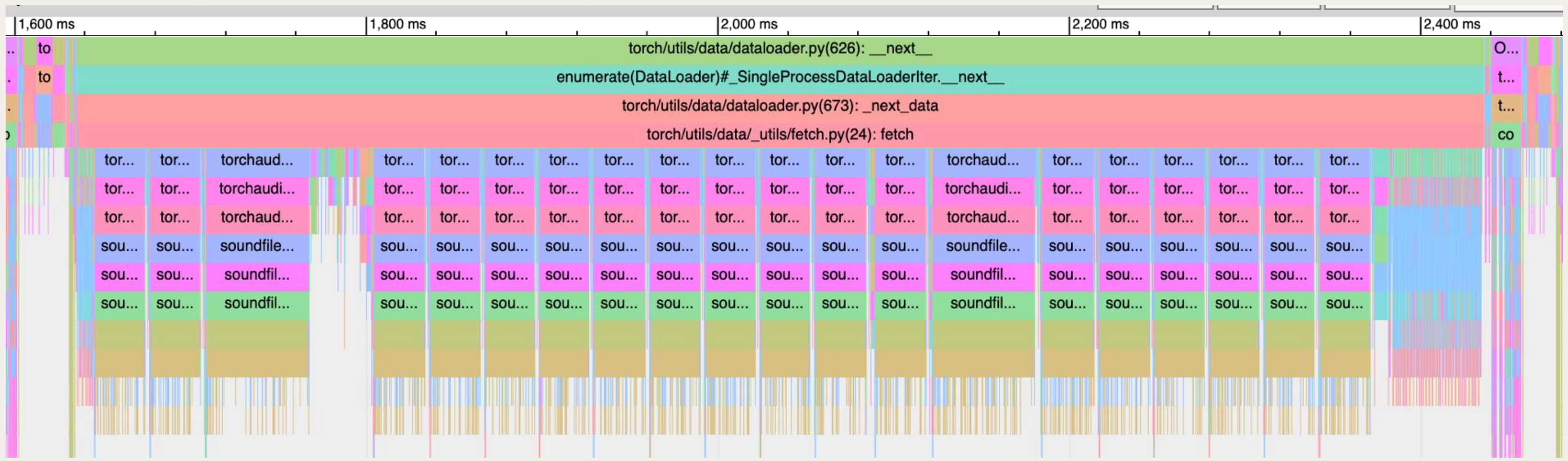


Mitigating Bottlenecks – An Example

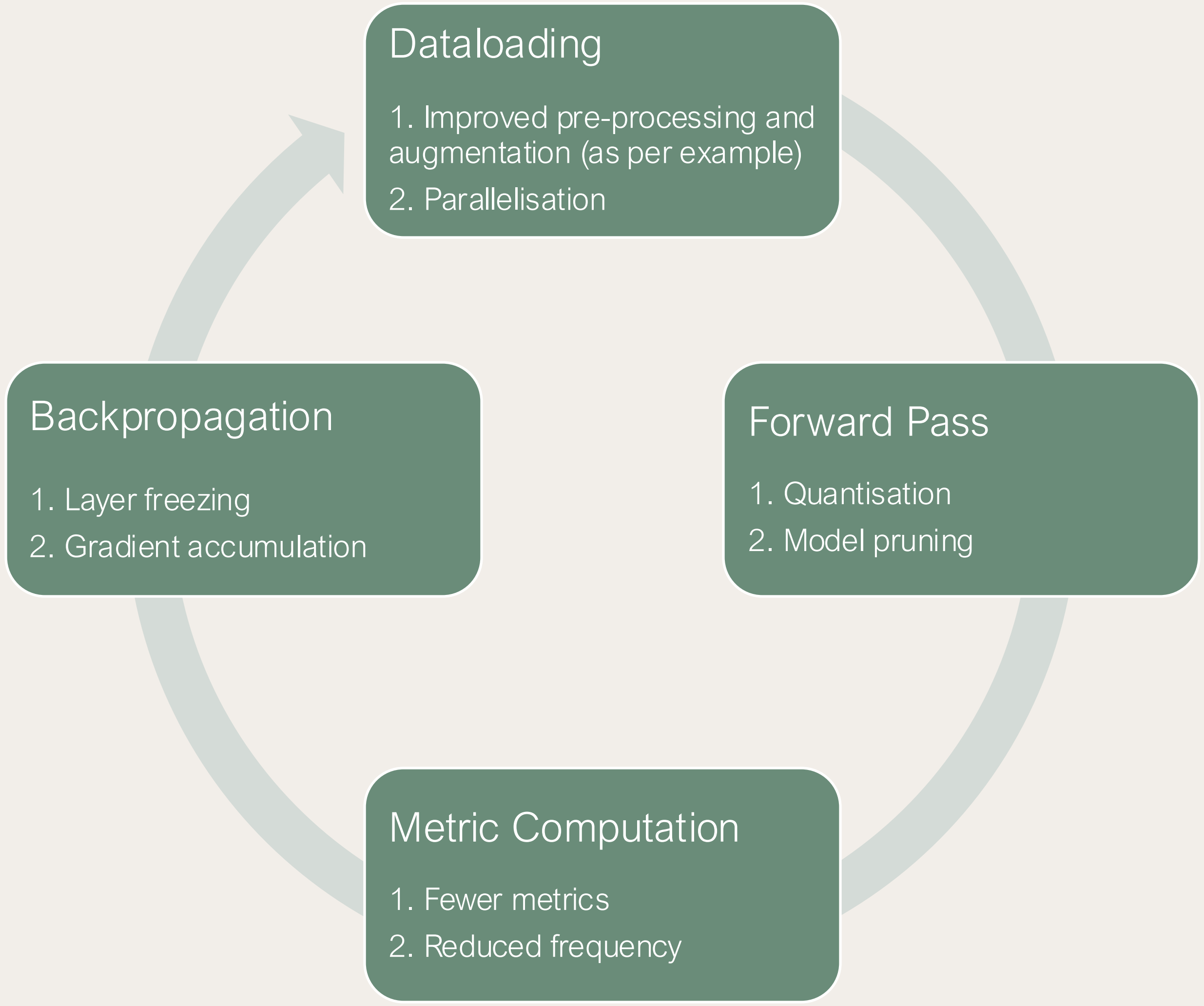
- By replacing the calls to *slow_conv2d_forward* with a faster GPU implementation, we removed this bottleneck


2,000 ms
2,200 ms
2,400 ms
2,600 ms
2,800 ms
3,000 ms
3,200 ms
torch/utils/data/dataloader.py(626): __next__
enumerate(DataLoader)#_SingleProcessDataLoaderIter.__next__
torch/utils/data/dataloader.py(673): _next_data
torch/utils/data/_utils/fetch.py(24): fetch
torchaud...
soundf...
soundfil...

- This reduced the core loop time by ~46%; jumping from 2.64 to 4.91 batches/second


1,600 ms
1,800 ms
2,000 ms
2,200 ms
2,400 ms
torch/utils/data/dataloader.py(626): __next__
enumerate(DataLoader)#_SingleProcessDataLoaderIter.__next__
torch/utils/data/dataloader.py(673): _next_data
torch/utils/data/_utils/fetch.py(24): fetch
torchaud...
torchaudi...
soundfile...
soundfil...

Summary: Optimising Implementation



Dataloading

1. Improved pre-processing and augmentation (as per example)
2. Parallelisation

Forward Pass

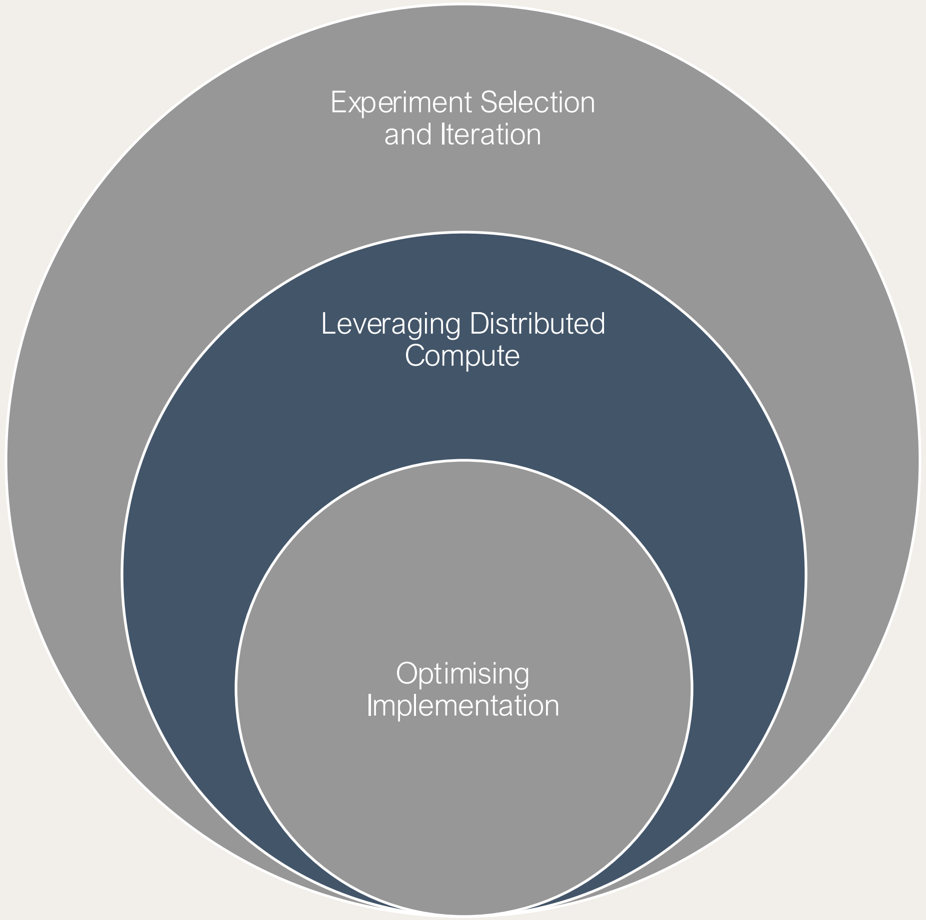
1. Quantisation
2. Model pruning

Metric Computation

1. Fewer metrics
2. Reduced frequency

Backpropagation

1. Layer freezing
2. Gradient accumulation


Experiment Selection and Iteration
Leveraging Distributed Compute
Optimising Implementation

02

Leveraging Distributed Compute

What’s better than one GPU?

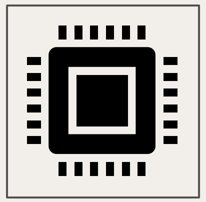
Optimising a model implementation can only go so far…

- If we have identified and mitigated as many bottlenecks as possible, we have reached a limit on how quickly the core machine learning loop can be run

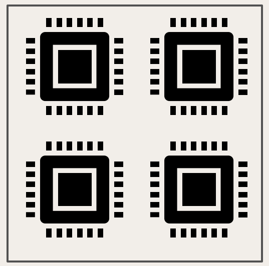
- But we can still further speed up performance by leveraging distributed compute

- Using multiple GPUs can significantly reduce training/evaluation/inference time if we are able to minimise overheads

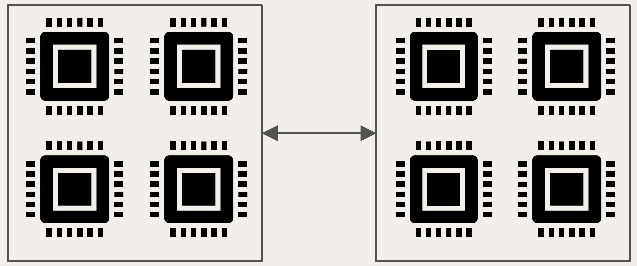
1 x 1: One node with one GPU



1 x 4: One node with four GPUs

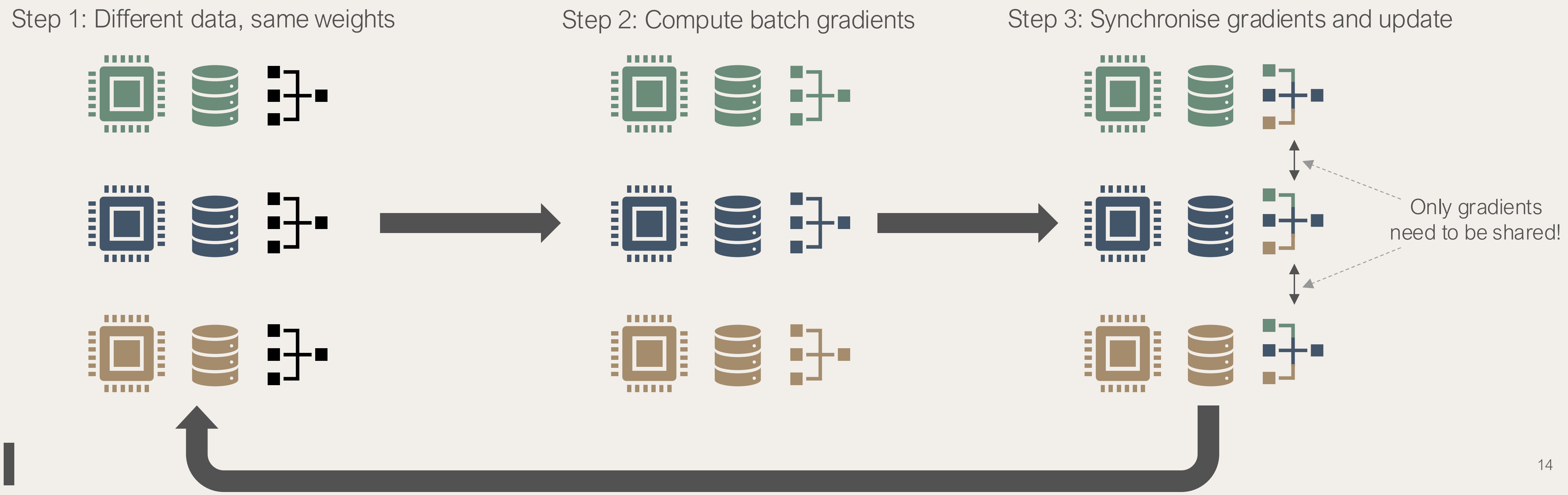


2 x 4: Two nodes with four GPUs each



Distributed Data Parallel

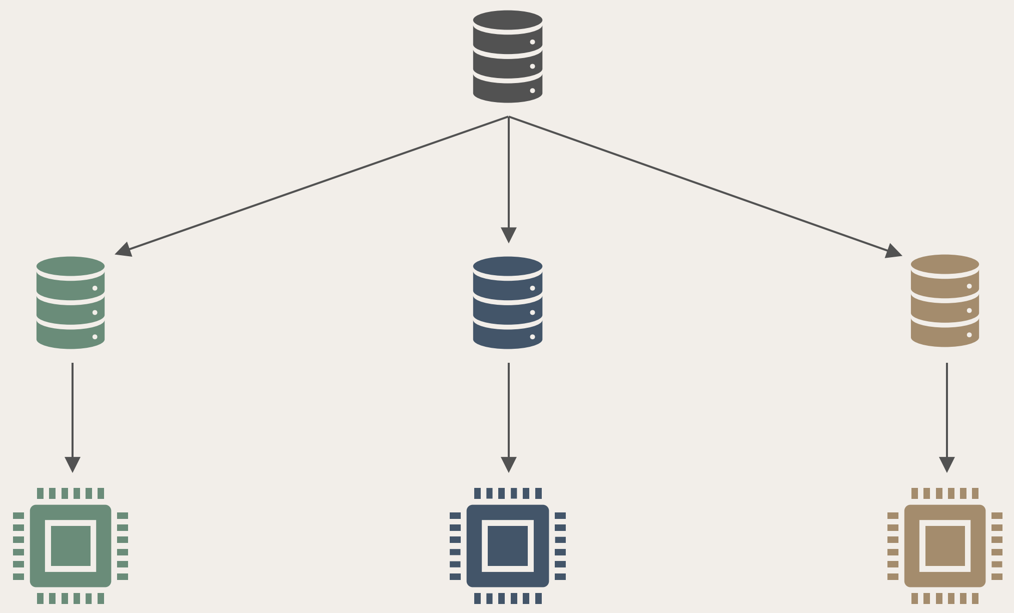
- Given some configuration of GPUs, one approach to parallelisation is to divide the data
- Each GPU has the same model and weights, but operates on a different portion of the data
- During training, each GPU computes gradients independently, then the gradients across all GPUs are summed (all reduce) and synchronised



Step 1: Different data, same weights
Step 2: Compute batch gradients
Step 3: Synchronise gradients and update
Only gradients need to be shared!

Sharing Data Across Machines

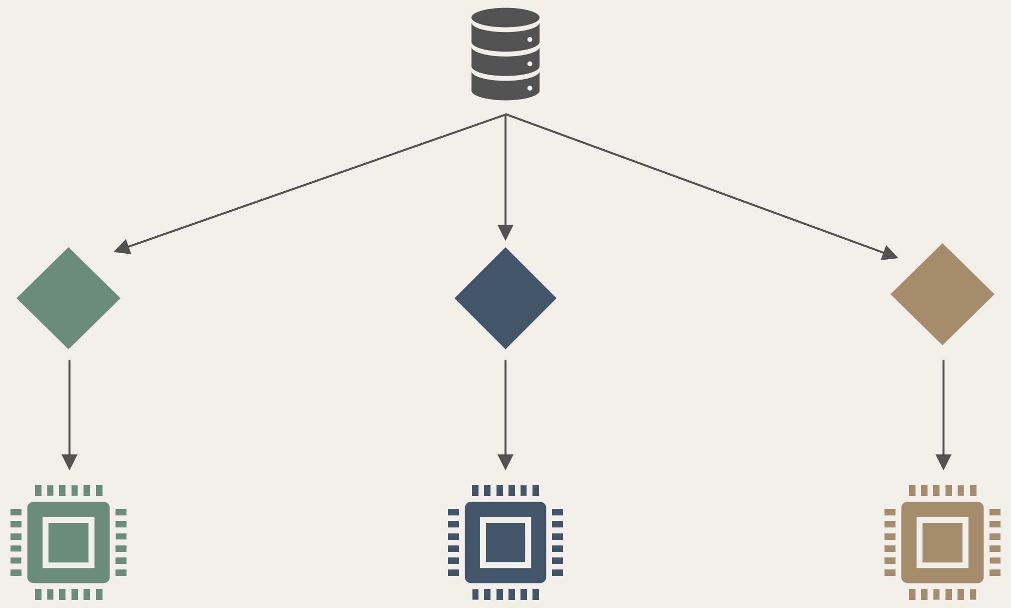
- In the previous example, the dataset was divided into three equal parts; one for each GPU
- If these GPUs were on different machines, ideally each machine would only download the data it requires
- If spinning up different devices each time training is run, there is a large overhead of downloading the data before training can begin



Can we avoid this bottleneck?

Streaming Datasets

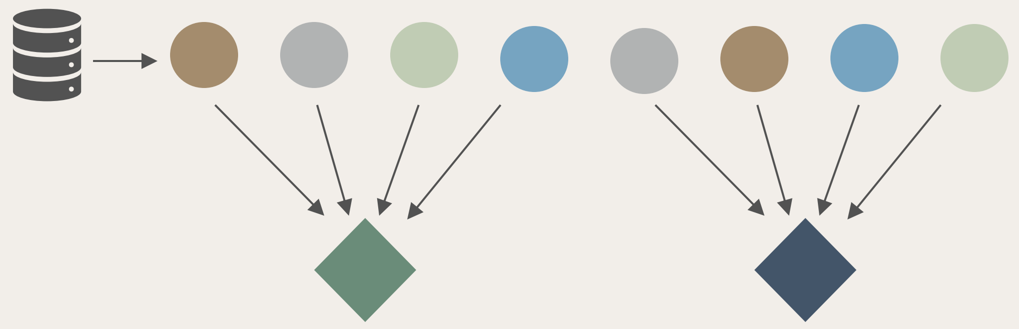
- Streaming datasets (e.g. those provided in MosaicML's Streaming library) download data incrementally
- Through parallelism, data is downloaded in the background while the model is already training
- Benefits:
  - No bottleneck of waiting for the entire dataset to download
  - You can work with very large datasets without requiring lots of disk space (repeatedly streaming data without caching)
  - When deploying you model, your pipelines are already configured to stream and process data in real time
  - Downloading data is more efficient due to sharding (see next slide)



Sharding

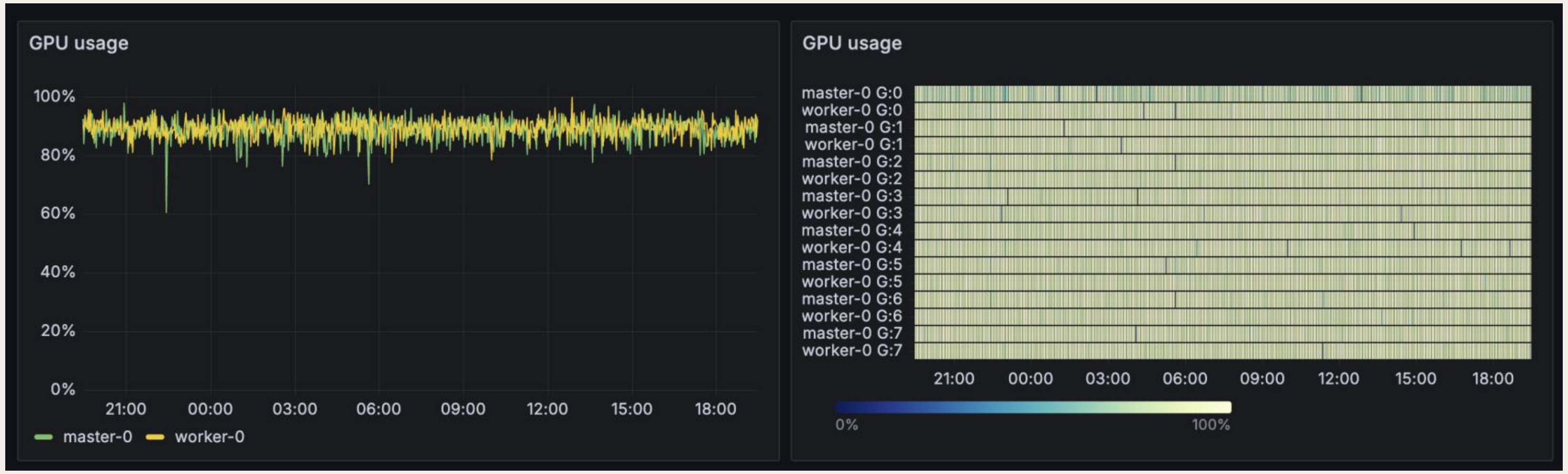
- Streaming typically operates with sharded datasets
- Shards are chunks of data containing several samples which are then compressed
- Sharding a dataset provides a good trade-off between fast random access and fast download speeds

Data Storage Method	Fast Random Access	Fast Download Speed	No Pre-processing Required
Individual Files	✓	✗	✓
Compressed Archive	✗	✓	✗
Sharded	✓	✓	✗



GPU Utilisation

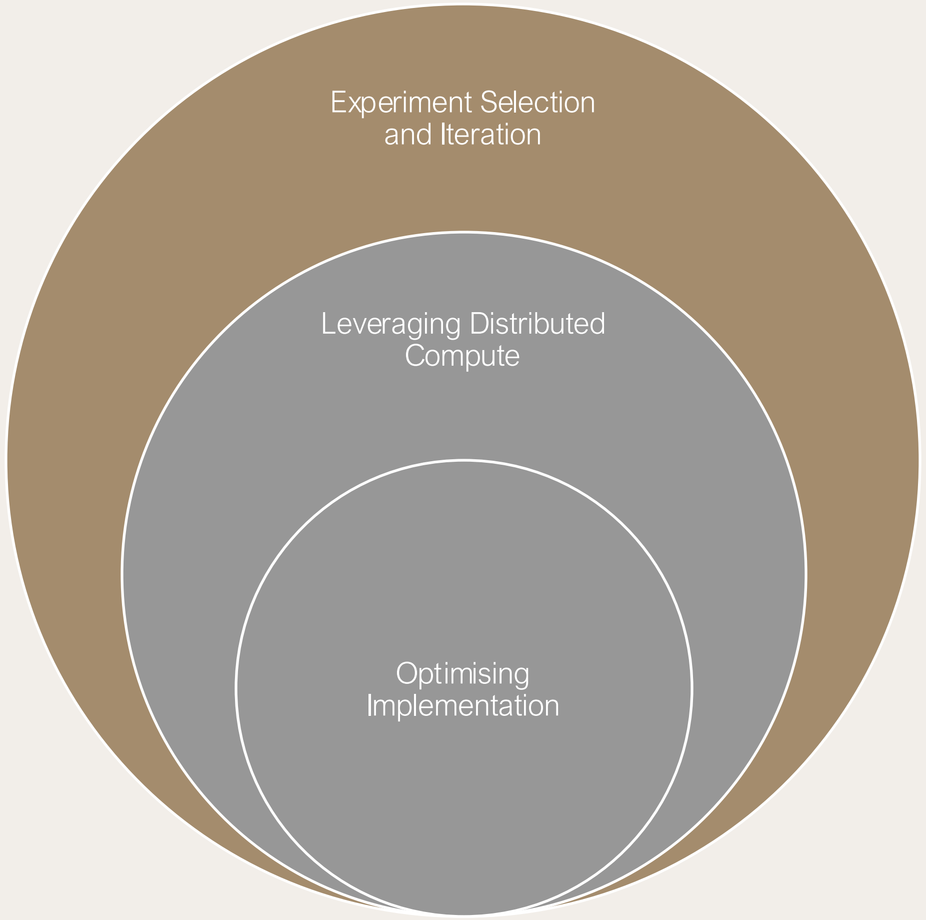
- Regardless of how many GPUs you are using, you want to maximise their usage
- GPUs are expensive but usage is typically independent of cost
  - Having them run at 40% usage costs the same as running them at 100% usage
  - Higher usage = shorter training times = lower costs!
- Optimising the implementation and using streaming datasets improves usage
- Using large batch sizes also improves usage (more efficient use of GPU memory)
- If you can't max out the GPU, it may be too big for your model!



Summary: Leveraging Distributed Compute

More GPUs means faster training/evaluation/inference, but…

- You want to be maximising the usage of each GPU
- Techniques such as streaming dataloaders and distributed data parallel can make using multiple GPUs across multiple devices more efficient
- The type of GPU and configuration (i.e. number of GPUs and nodes) required is dependent on the dataset and model: you don’t need 100 GPUs to train a linear model…


Experiment Selection and Iteration
Leveraging Distributed Compute
Optimising Implementation

03

Experiment Selection and Iteration

Running Lots of Experiments

To get the best performing model in a fixed time window, you want to be running experiments as often as you can; ideally all the time

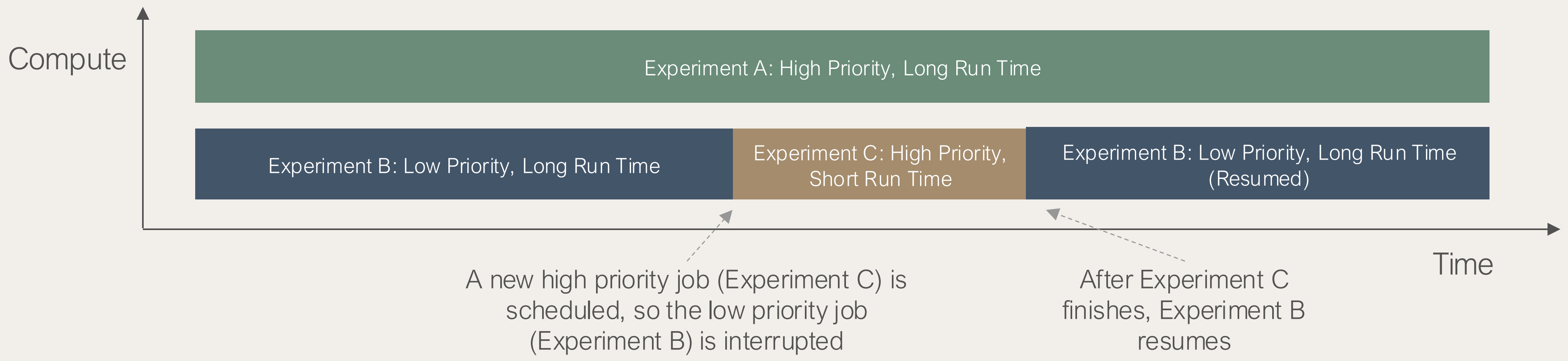
- More experiments = more options = more results = better performance
- Typically, you will have higher priority experiments and varying experiment run times
    - A longer run with a larger model that you can't run very often would have high priority
    - But lots of shorter low priority experiments can still provide valuable information
- Choosing sensible priorities for running experiments is essential for achieving strong performance in a limited amount of time
- You can always do more (and will always want to!), but often don't have the time or resources

Aside: Experiment Resumption

- Clusters can be managed with job priority and scheduling to avoid/reduce manual oversight
  - Users don't start their experiments, instead they request a specific amount of compute and specify a priority (and potentially an estimated run time); tools such as Kubernetes + Volcano achieve this

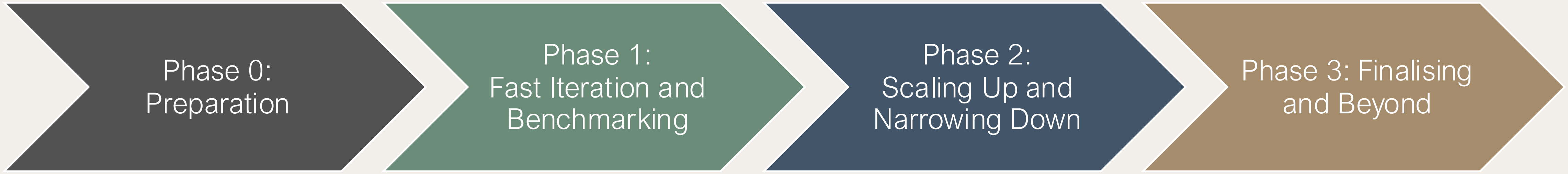
- Long wait times can be somewhat alleviated through experiment resumption
  - Low priority experiments can be interrupted by high priority jobs and later resumed

- Methods such as checkpointing, where the current weights and state are saved, allow resumption part-way through training, ideally with as little progress lost as possible


Compute
Experiment A: High Priority, Long Run Time
Experiment B: Low Priority, Long Run Time
Experiment C: High Priority, Short Run Time
Experiment B: Low Priority, Long Run Time (Resumed)
Time
A new high priority job (Experiment C) is scheduled, so the low priority job (Experiment B) is interrupted
After Experiment C finishes, Experiment B resumes

Creating an Experiment Plan

Having a strong plan for what experiments you will run and what your objectives are will be invaluable in the long run


Phase 0: Preparation
Phase 1: Fast Iteration and Benchmarking
Phase 2: Scaling Up and Narrowing Down
Phase 3: Finalising and Beyond

Phase 0: Preparation

Prior to starting training, you need to:

- Understand your problem very well to discover constraints such as:
  - What are the top priorities?
    E.g. greatest predictive performance, speed of inference, etc.
  - Are there any resource constraints when the model will be deployed?
    E.g. limited compute, limited/no access to the internet, etc.
  - What are the current state-of-the-art approaches in this area?
    E.g. those from the literature, standard benchmarks that are used, etc.

- Identify the datasets you're going to work with
  - **You really need a gold standard test dataset that you have a lot of confidence in**
  - Are there a variety of datasets you could use, and what are the strengths and weakness of each?
    This involves gathering statistics such as the number of samples, number of classes, potential mislabelling, normalisation values, etc.

- Define the metrics you're going to use to evaluate performance: what does success look like?

Phase 1: Fast Iteration and Benchmarking

Exploring the problem quickly and setting a strong foundation is the first step

- You probably don't want to jump into training the largest model possible straight away – you could end up wasting a lot of time on what you think will work best only to find it isn't great

- Instead, it's much better to start with many "cheap" options that you can train quickly and see what works
  - Lightweight training approaches typically leverage pre-trained weights with custom classifiers
    E.g. linear probing, KNN, etc.
  - Techniques such as early stopping are beneficial here to prune unpromising runs

- It's also essential to use existing benchmarks here to understand the relative performance of your approach
  - Especially if you're using a custom dataset with no public/existing results

- If you have a large training dataset (lucky you!), iteration time can be reduced by using a smaller version of the dataset for initial training

Phase 2: Scaling Up and Narrowing Down

- Once you have a solid foundation, you can look to expand to larger models, more complex training, and bigger datasets (scaling up)
  - E.g. rather than just doing linear probing, you could try fine-tuning all the weights of a model

- In the first phase you've done a broad exploration of different approaches, but now you understand what works so can be more focused (narrow down)
  - E.g. as you start to adapt your existing approaches, you don't need to try this with every configuration you used previously, just the ones that worked best

- As each experiment will now take longer, you'll be running fewer experiments but should still be improving performance

Phase 3: Finalising and Beyond

- Hopefully by this point you have settled on a strong approach that yields good performance

- While the big performance gains have likely come in the prior phases, you can probably still can a small amount of performance with a few more changes
  - E.g. hyperparameter tuning (grid search or better), model ensembling, edge case analysis, etc.

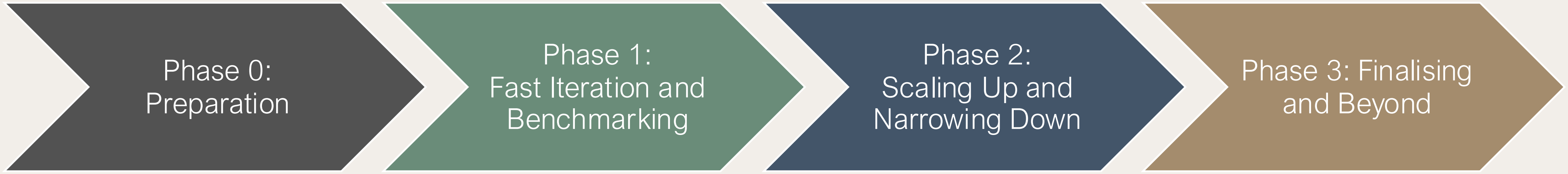
- There are a few final steps prior to deployment (and beyond):
  - Model compression (e.g. pruning/quantisation) and exporting (e.g. ONNX) for specific hardware
  - Documentation and operational guidelines (e.g. assumptions and limitations)
  - Monitoring and potential retraining following deployment

Summary: Experiment Selection and Iteration

- Having a clear plan for the experiments you want to run and what success is will make developing AI models faster and a smoother process

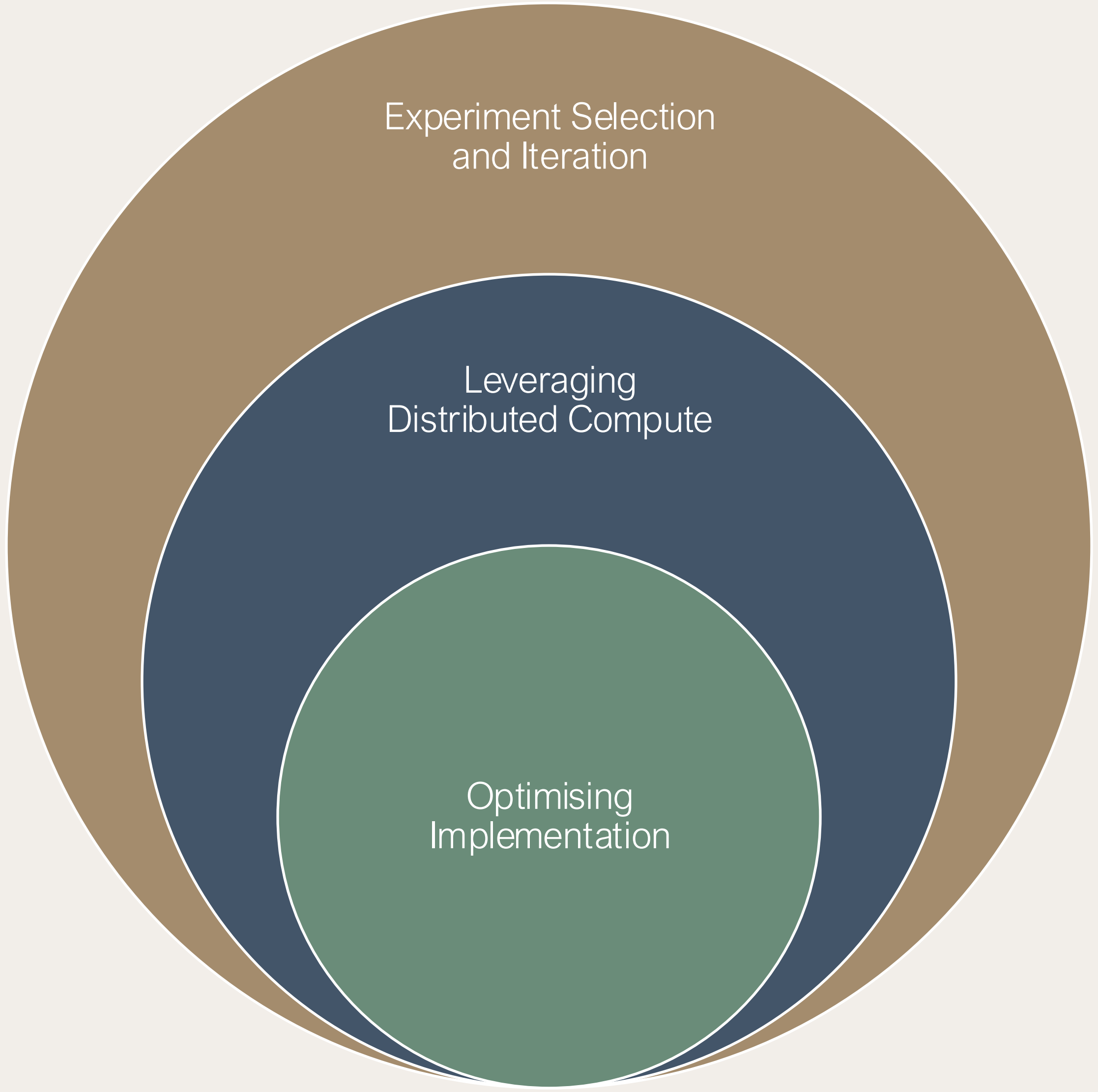
- While these phases can see quite distinct, there is often overlap between them and you'll likely need to adapt
  - If you get more data, you might have to revisit Phase 0
  - If a new benchmark is published, you might have to revisit Phase 1
  - If things aren't working when scaling up, you might have to pivot to alternative models

- Ultimately, things will change along the way as you explore the problem, but having a good plan will help!


Phase 0: Preparation
Phase 1: Fast Iteration and Benchmarking
Phase 2: Scaling Up and Narrowing Down
Phase 3: Finalising and Beyond

Summary: AI Done Quick

- There are many facets to developing AI models quickly
- Not all of them will be relevant to all problems, but you have multiple levers you can pull to move at speed
- The process is often not linear; you'll likely jump around between running experiments, optimising implementations, and improving infrastructure


Experiment Selection and Iteration
Leveraging Distributed Compute
Optimising Implementation

Thank you!

Joe Early, AI Research Engineer, Helsing

AI for Ukraine Recovery Hackathon, 7 September 2024
Tallinn, Estonia