

Machine Learning Systems

Week 1: Introduction

From tensors to clusters

Luo Mai

School of Informatics, University of Edinburgh

Who is teaching this, and why

Luo Mai · Associate Professor, School of Informatics

Research: large-scale ML systems — how to train and serve models efficiently on real hardware. Leads the Large-Scale ML Systems Group.

Systems we built: ServerlessLLM (LLM serving), Tenplex (elastic training), Ekko (large-scale model updates), WaferLLM (wafer-scale inference) — published at OSDI, SOSP and MLSys, open source and in use.

Industry: long-running collaborations with Cerebras, Tencent and Microsoft Research, where these systems run in production.

Community: co-lead of the UKRI Centre for Doctoral Training in Machine Learning Systems; General Chair of EuroSys 2026.

Why this course: every problem in my research is the gap between a model and the hardware it runs on. This course teaches the way of thinking that closes that gap — the same one we use in our papers and with our industry partners.

If this way of working interests you: MSc and UG projects, and PhD places through the **CDT in ML Systems**.

The model runs. Why is the system slow?

Training takes hours.

A chatbot keeps users waiting.

Adding GPUs gives little speedup.

Where does the time go?

In all three cases the model itself is fine: it computes the right answer. What is slow is the execution — how the computation is mapped onto processors, memory and networks. This course is about that mapping.

Our ten-week journey

1

Foundations

ML systems, data pipelines, GPU architecture and CUDA

2

Building efficient systems

GPU optimisation, agentic coding, distributed ML

3

Operating at scale

LLM inference and serving, cluster management, research topics

By the end of today you should be able to:

- describe a tensor by shape, dtype and device
- read a training loop and name what each line does
- explain why speeding up one stage may not speed up the whole step

Next week: Data systems — Yang Cao · Week 3: GPU / CUDA

How the course runs

Assessment

100% coursework, no exam

Group component (70%)

Individual component (30%)

Details to be announced later by Dr. Yang Cao

Practicalities

No labs or tutorials

Questions: Edstem

Link on the Learn course page

Office hours

Times to be announced

What we assume

Basic systems knowledge

How a program runs, what memory and processes are

Basic machine learning knowledge

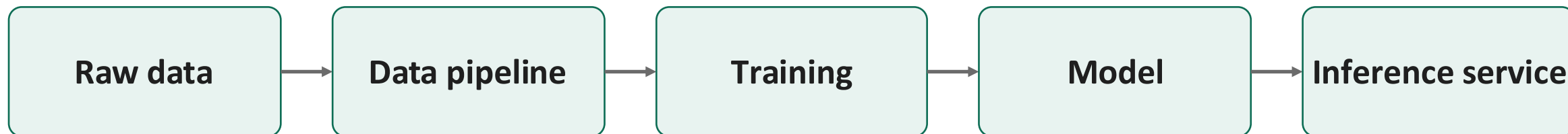
What a model, a loss and a gradient are

Most of the material is self-contained.

If you are missing a concept, ask an AI assistant to explain it, then check it against what we cover here. Having the background helps you move faster; not having it is not a barrier.

AI tools and the coursework: which tools are allowed, and what records you must submit, will be stated when the coursework is released.

A full-stack ML system

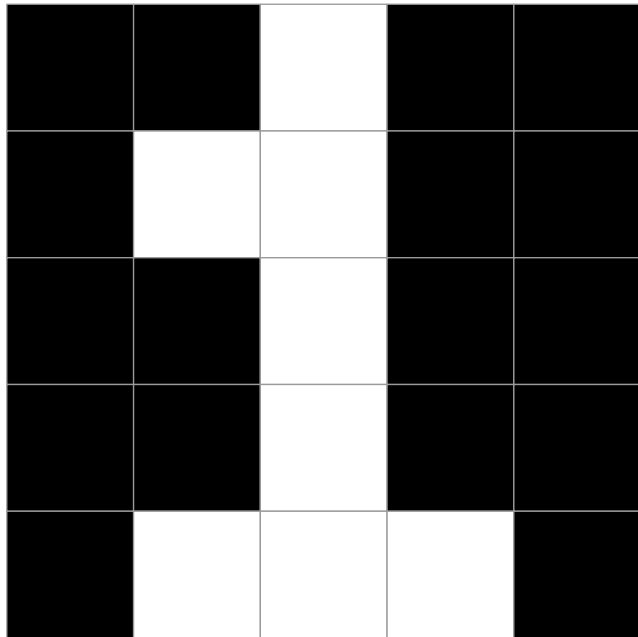


Shared foundations: frameworks, processors, memory and networks

The data pipeline reads and prepares raw data. Training consumes it and produces a model (a set of learned parameters). An inference service loads that model and answers requests.

Every stage is a program, and every arrow is data moving between programs. All stages run on the same kinds of hardware and the same frameworks, so the same performance questions apply to each of them. In production, cluster management runs many such training and serving jobs side by side.

An image is an array of values



What we see



0	0	255	0	0
0	255	255	0	0
0	0	255	0	0
0	0	255	0	0
0	255	255	255	0

What the program stores

One intensity value per pixel

0 = black, 255 = white

A 5×5 image, for illustration.
The classifier later uses 28×28 images.

Image → Numerical array → Tensor

A tensor is the framework's name for such an array of numbers, together with some bookkeeping about it (its shape, the type of each value, and where it is stored). A colour image would have three values per pixel; a sound clip would be a one-dimensional array of samples. Everything the model touches is a tensor.

From one image to a batch

One grayscale image

$[28, 28]$

With an explicit channel axis

$[1, 28, 28]$

A batch of 64 images

$[64, 1, 28, 28]$

Axis order used here: **Batch × Channels × Height × Width**

Channels: how many values each pixel has — 1 for grayscale, 3 for RGB. Keeping the axis even when it is 1 lets the same code handle both.

Batch: several examples processed together in one pass. Batching is mainly a systems decision: one large operation is usually much more efficient than many small ones. Other frameworks may order the axes differently (e.g. channels last); the meaning is the same.

Three tensor properties

Property	Question	Example
Shape	How is the data organised?	<code>[64, 1, 28, 28]</code>
Dtype	How is each value represented?	<code>float32</code>
Device	Where is the tensor stored?	<code>CPU</code> or <code>GPU</code>

Shape is set by the data, the model, and choices such as batch size or sequence length. Dtype is a choice: `float32` uses 4 bytes per value, `float16` uses 2, so halving the precision halves that tensor's memory and often speeds up computation, sometimes at a cost in accuracy. Device decides which processor can compute on the tensor; moving a tensor between devices (`tensor.to(device)`) takes time, so systems try to avoid unnecessary moves.

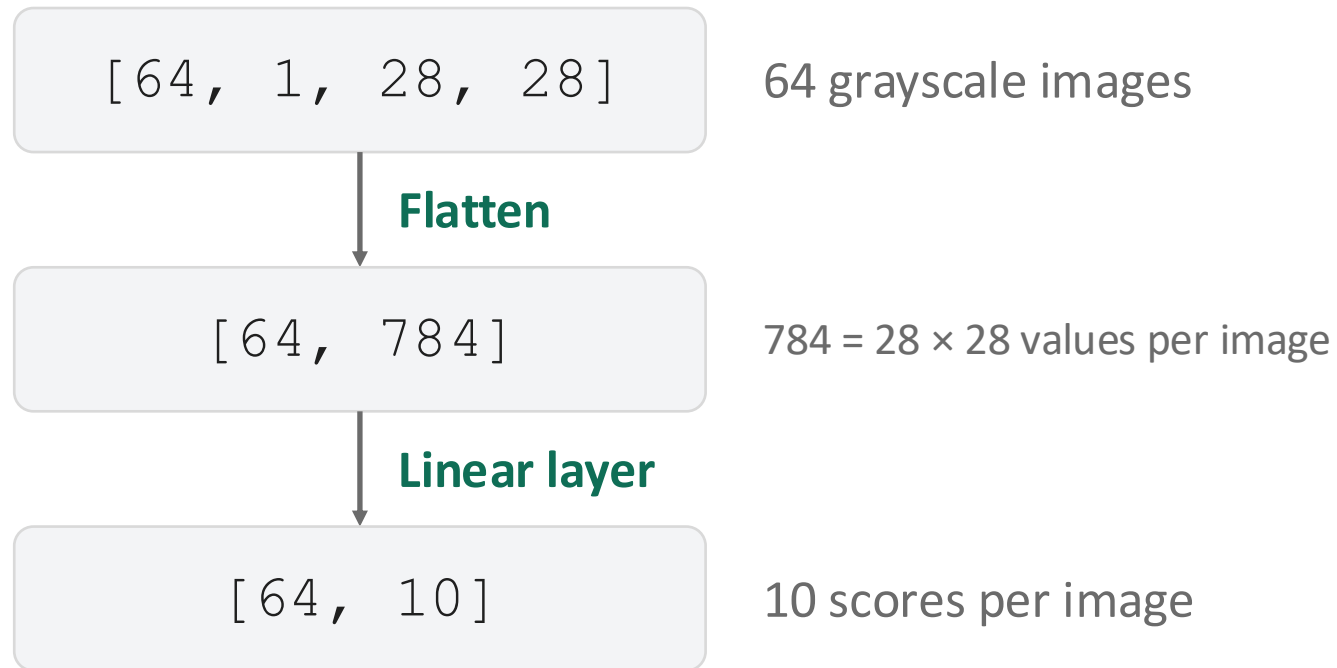
The same shape does not imply the same storage cost or execution location.

Tensors represent more than inputs

Tensor	Role	When it exists
Input batch	Examples being processed	Training and inference
Parameters	Values learned by the model	Training and inference
Activations	Intermediate computation results	Kept for backward in training; freed early in inference
Gradients	Derivatives used for learning	Training only

Peak memory is the largest amount of data alive at one moment, plus framework overhead. Inference also computes activations, but can free each one after its last use. Training must keep the ones backward will need, and optimisers such as Adam add state per parameter on top. This is why a model that fits in memory for inference may not fit for training.

A simple digit classifier



Ten scores per image: one for each digit, 0–9.

Flatten rewrites each 28×28 image as one row of 784 values. No information is lost; only the organisation changes.

The linear layer turns the 784 values into 10 scores. The highest score is the model's prediction.

A real digit classifier would have more layers. This one is deliberately small so that we can follow every step of its execution; everything we learn from it applies to larger models.

The computation behind the model

$$\mathbf{S} = \mathbf{X} \mathbf{W}^T + \mathbf{b}$$

X	[64, 784]
W	[10, 784]
b	[10]
S	[64, 10]

ML view

A linear classifier

Systems view

Tensor storage, matrix multiplication and addition

Each row of X is one image (784 values). Each row of W holds the 784 weights for one digit. S[i, j] says how strongly image i looks like digit j; b adds the same ten offsets to every image.

Check the shapes: $[64, 784] \times [784, 10] \rightarrow [64, 10]$. Almost all of the arithmetic in a neural network is matrix multiplications like this one, which is why hardware and libraries are built around them.

How much memory do these tensors take?

Tensor	Shape	Dtype	Bytes	Size
X	[64, 784]	float32	$64 \times 784 \times 4$	≈ 196 KiB
W	[10, 784]	float32	$10 \times 784 \times 4$	≈ 31 KiB
S	[64, 10]	float32	$64 \times 10 \times 4$	≈ 2.5 KiB

Same dtype, very different sizes.

Here the input batch is larger than the model. For large models the opposite is true.

Memory = number of values $\times$ bytes per value. One float32 is 4 bytes; 1 KiB is 1024 bytes. The same arithmetic gives the size of a 7-billion-parameter model ($7 \times 10^9 \times 2$ bytes ≈ 14 GB in float16) or of an LLM's cache during serving.

Fitting in memory is only the first question; how fast the data can be read and written matters as much. Week 3 returns to this.

Quick check: double the batch size

The batch changes from 64 to 128 images. Which tensors grow?

A. Only X

B. X and S

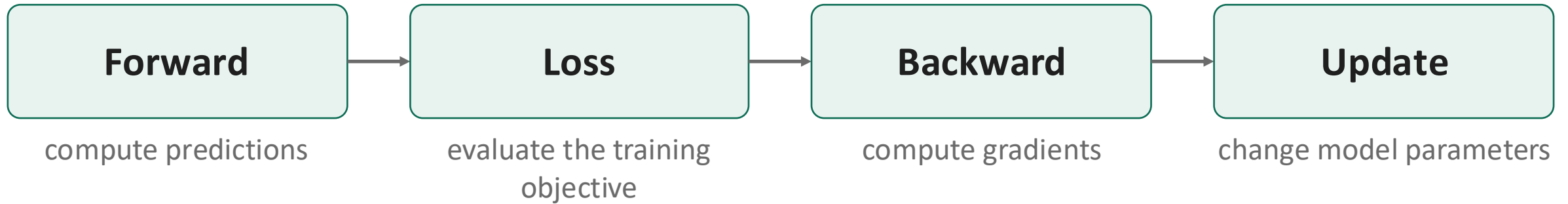
C. X, S and W

D. All of them

X	[64, 784]
W	[10, 784]
b	[10]
S	[64, 10]

Think about it for 15 seconds before looking at the answer (in the speaker notes). Then ask yourself: which of these tensors would grow if the model had more layers instead?

One training iteration



Forward: run the model on a batch and get scores S . **Loss:** compare the scores with the true labels and produce one number that says how wrong the model is — if an image shows a 3 but the highest score is for 8, the loss is high.

Backward: compute, for every parameter, how the loss would change if that parameter changed slightly (the gradient).

Update: move every parameter a small step in the direction that reduces the loss. Repeat for the next batch, many thousands of times.

Computing gradients (backward) and updating parameters (update) are different operations.

Questions

Tensors, batches and training steps

What is unclear so far?

10-minute break

Next: reading the training code and finding performance bottlenecks

Edinburgh University
Venture & Innovation
Society (EUVIS)



From a model to an executing system

We know what to compute. Now ask:

Where does the data come from?

Who performs each operation?

Where does the time go?

The first half described the computation as mathematics. The second half looks at the same computation as a running program: the code, the software layers underneath it, and the time each stage takes.

What makes a system better?

Workload	Example objective
Training	Reach target quality sooner
Interactive inference	Respond within a latency target
Batch inference	Process more inputs per second
Shared cluster	Use resources effectively across jobs

"Faster" only means something relative to an objective. A change that makes each training step faster but needs more steps to reach the same accuracy is not an improvement. High GPU utilisation is a useful signal, not a goal in itself: a system can keep the GPU busy doing unnecessary work.

State the workload, constraints and metric before comparing systems.

Latency and throughput

Latency

Time to complete one request or one step

Throughput

Work completed per unit time

A larger batch may improve throughput while increasing waiting time.

Example: a chatbot server that waits to collect 32 requests before running the model does more work per second (higher throughput), but the first request in the group waits for the other 31 to arrive (higher latency). Which matters more depends on the workload: batch inference cares about throughput, an interactive service has a latency target it must meet.

Dataset and DataLoader

Dataset

Defines how to obtain one example

DataLoader

Organises examples into batches

Can support shuffling and parallel loading

The Dataset answers "what is example number i ?" — typically by reading a file and converting it to a tensor. The DataLoader answers "give me the next batch": it asks the Dataset for many examples, stacks them into one tensor such as $[64, 1, 28, 28]$, and can do this in parallel worker processes while the GPU is busy. Each loop iteration receives one batch of images and their labels.

Next week: how to build an effective data systems.

```
for images, labels in loader:  
    ...
```

Define the model

```
model = nn.Sequential(  
    nn.Flatten(),  
    nn.Linear(784, 10),  
) .to(device)
```

Flatten each image.

Compute ten scores.

Place the model on the target device.

Shapes through the model:

$[64, 1, 28, 28] \rightarrow [64, 784] \rightarrow [64, 10]$

`nn.Sequential` runs its layers in order. `nn.Linear(784, 10)` owns the parameters W $[10, 784]$ and b $[10]$ from the earlier slide, and computes $S = XW^T + b$. `.to(device)` moves those parameters to the GPU (or keeps them on the CPU); the input batch must later be moved to the same device.

Read the training loop

```
model.train()

for images, labels in loader:
    images = images.to(device)
    labels = labels.to(device)

    optimizer.zero_grad()
    scores = model(images)
    loss = criterion(scores, labels)
    loss.backward()
    optimizer.step()
```

The loss function (criterion), optimiser and loader are already configured.

`model.train()` switches the model into training mode; it does not train anything by itself.

Every iteration of the loop is one training step on one batch.

Map the code to the training steps

Code	What it does
<code>images.to(device)</code>	Move the batch to the device that holds the model
<code>optimizer.zero_grad()</code>	Clear the gradients left over from the previous step
<code>model(images)</code>	Forward: compute the scores
<code>criterion(scores, labels)</code>	Loss: compare scores with the true labels
<code>loss.backward()</code>	Backward: compute the gradient of every parameter
<code>optimizer.step()</code>	Update: change the parameters using those gradients

PyTorch adds new gradients to whatever is already stored, which is why `zero_grad()` comes first. Adding on purpose (gradient accumulation) is a way to simulate a larger batch, but then `step()` must be called less often.

Training and inference

Training (today)

Forward → Loss → Backward → Update

Keeps activations and gradients

Many steps; measure time to target quality

Inference (serving a model)

```
model.eval()  
with torch.no_grad():  
    scores = model(images)
```

Forward only; no gradients, no update

Measure latency and throughput

Same questions in both: where is the data, how is it batched, is the output correct, where does the time go. Everything in the first half — shapes, dtypes, devices, memory arithmetic — applies to inference unchanged.

`model.eval()` and `torch.no_grad()` are two different things: `eval()` switches layer behaviour (e.g. dropout, batch norm) to inference mode; `no_grad()` stops the framework recording the graph, which saves memory and time.

Serving a model is today's training loop with two steps removed.

Quick check: remove zero_grad()

Delete this line from the loop:

```
optimizer.zero_grad()
```

What happens?

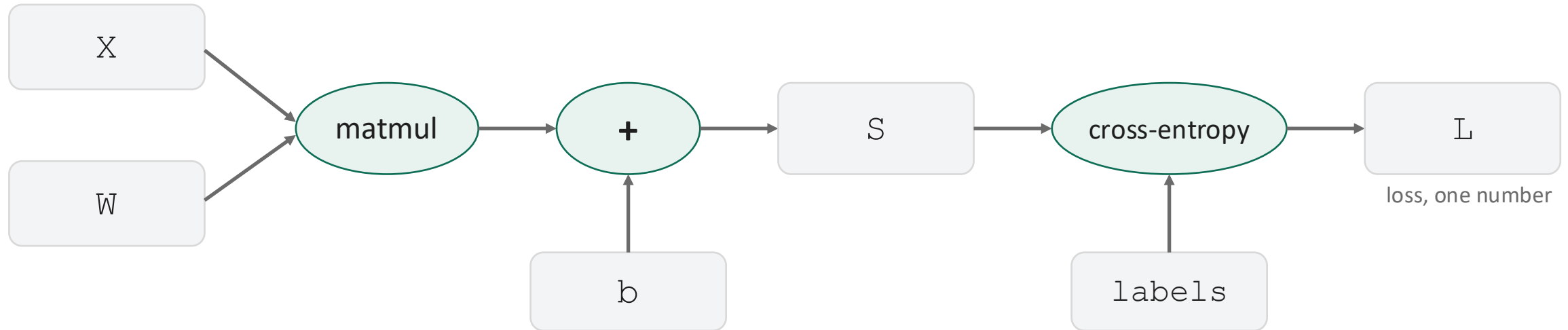
A. The program raises an error

B. Gradients accumulate across batches; each update uses old gradients too

C. Nothing changes

Answer in the speaker notes. Hint: look at the previous slide's note on how PyTorch stores gradients.

The computational graph



← Backward: gradients flow from L back to W and b, through the same graph

During forward, the framework records every operation and its inputs. The result is this graph: circles are operations, boxes are tensors. Backward walks the graph in reverse, applying the chain rule at each operation to turn dL/dS into dL/dW and dL/db . This is automatic differentiation: you write forward, the framework derives backward.

The graph is also what compilers optimise, what decides which activations must be kept in memory, and where distributed training inserts communication. It will come back in later weeks.

Backward is also matrix multiplication

dL/dS $[64, 10]$ how the loss changes with each score
 $dL/dW = (dL/dS)^T \times [10, 64] \times [64, 784] = [10, 784]$ same shape as W
 $dL/db = \text{sum of the rows of } dL/dS$ $[10]$ same shape as b

Why: $S[i, j] = \sum_k X[i, k] \cdot W[j, k] + b[j]$, so the chain rule gives $dL/dW[j, k] = \sum_i dL/dS[i, j] \cdot X[i, k]$ — which, written as a matrix, is exactly $(dL/dS)^T X$.

Backward needs X

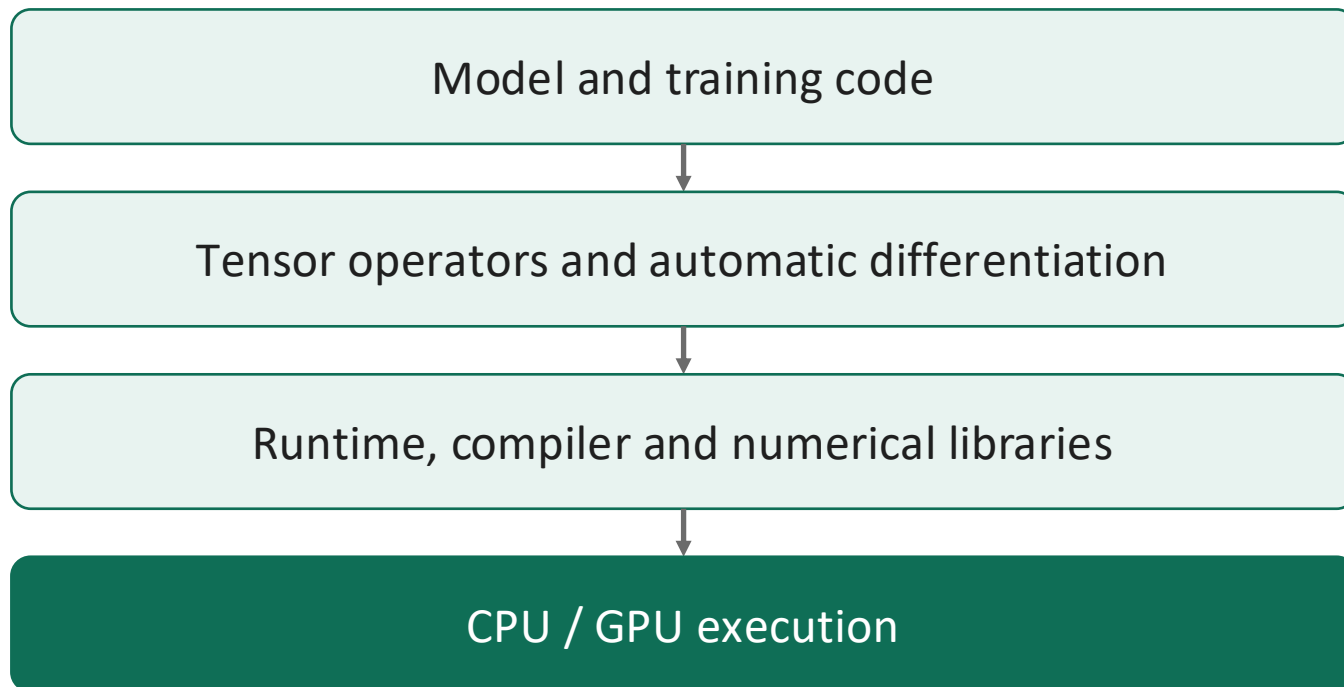
The inputs and intermediate values of forward must stay in memory until backward has used them. This is the activation memory from the first half.

Backward costs about as much as forward

It is another matrix multiplication of the same size. At the same batch size, a training step does roughly 2–3× the arithmetic of an inference step (arithmetic, not time).

You will not be asked to derive any of this. What to remember is in the two boxes. (dL/dS comes from the loss function; the framework computes it.)

From Python to hardware



A high-level operation is not necessarily one GPU kernel.

A kernel is a program that runs on the GPU. One line of Python may launch several kernels; a compiler may fuse several operations into one kernel.

Your code calls operators such as `matmul`. The framework records them (for backward), then hands them to a runtime that picks an implementation — often from a vendor library such as `cuBLAS` — and launches it on the device. Each layer can add overhead or remove it; when a model is slow, the cause may sit in any of them, which is why we will look at all four during the course.

What the framework does while your loop runs

Eager execution

Each line runs as soon as Python reaches it. Easy to read, easy to debug.

Cost: every operation makes one trip from Python to C++ to a kernel launch. For small operations that overhead can exceed the computation.

Asynchronous GPU execution

Python only enqueues kernels; the GPU runs them later, in order. CPU and GPU overlap without you doing anything.

Consequence: `time.time()` around a GPU call measures the enqueue, not the work — synchronise first. Reading a value back (`loss.item()`, `print`) forces a wait.

Caching memory allocator

Asking the driver for GPU memory is slow, so PyTorch keeps freed blocks and hands them out again.

Consequence: `nvidia-smi` shows what PyTorch has reserved, not what your tensors use. Fragmentation can cause out-of-memory with memory to spare.

Three design choices, three things you will see in every profile this term. Paper 1 of this week's reading, Section 5.

Eager or compiled?

```
model = torch.compile(model)
```

Capture the graph from Python bytecode (TorchDynamo)



Unsupported code? Break the graph and fall back to eager



Compile the graph into fused GPU kernels (TorchInductor → Triton)

What you gain

Fewer, larger kernels: several operations fused into one, so fewer launches and less memory traffic between them.

The same Python code: one line added, nothing rewritten.

What you pay

Compile time on the first run, and again when shapes change.

Harder to debug: the code that runs is no longer the code you wrote.

Graph breaks quietly give the gain back.

Compilation attacks overhead and GPU time. It does nothing for data preparation — next slide. Paper 2 of this week's reading.

Where does the step time go?

An illustrative workload — not the digit classifier from earlier. Sequential execution, no overlap.

Stage	Time
Prepare data	60 ms
Transfer data	10 ms
GPU computation	30 ms
Total	100 ms

GPU computation becomes 2× faster.

Does the whole step become 2× faster?

Prepare data: read files from disk, decode and transform them on the CPU. Transfer: copy the batch into GPU memory. GPU computation: forward, backward and update.

The numbers are made up to make the arithmetic easy; real values depend on the hardware, the data format and the model. You will measure real ones in the coursework.

Faster computation, smaller end-to-end gain

Before: $60 + 10 + 30 = 100$ ms

After: $60 + 10 + 15 = 85$ ms

Speedup = $100 / 85 \approx 1.18\times$

Even with zero GPU time, this sequential step still takes 70 ms.

This is Amdahl's law: if a fraction p of the time is spent in the part you speed up by a factor k , the overall speedup is $1 / ((1 - p) + p / k)$. Here $p = 0.3$ and $k = 2$, giving $1.18\times$. The parts you did not touch put a ceiling on the gain — no matter how fast the GPU becomes, the step cannot go below 70 ms while the stages run one after another.

So the first question is never "how do I make this component faster?" but "which component is actually taking the time?"

Optimise a component in the context of the whole workload.

What if stages overlap?

Sequential



Overlapped

prepare batch k+1 while
computing batch k

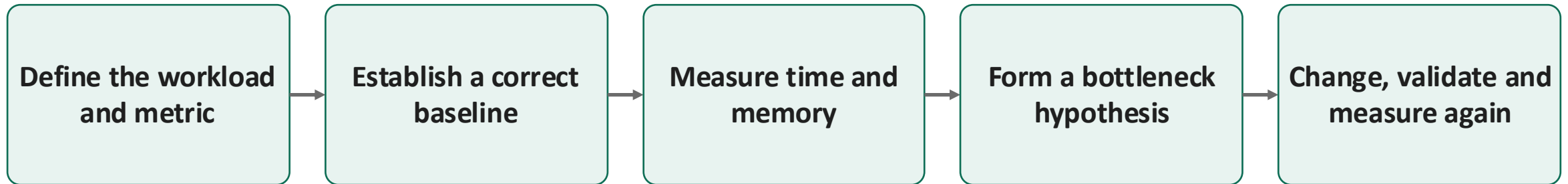


Nothing got faster. In steady state a batch completes every 60 ms instead of every 100 ms.

The CPU prepares the next batch while the GPU computes on the current one, so the two stages run at the same time instead of one after the other. This needs a buffer between them and enough CPU workers to keep up. Now the data preparation (60 ms) is the bottleneck, and every millisecond saved there shows up directly — for as long as it remains the bottleneck. Each batch still spends 100 ms in the pipeline: what improved is throughput, not latency (idealised: no start-up, drain or overheads).

Next week: how the data pipeline makes this overlap possible.

A reliable optimisation process



Measure before you change anything: intuition about where time goes is often wrong. Validate after every change: an optimisation whose output falls outside the stated numerical tolerance, or lowers model quality, is a bug, not a speedup. Floating-point results are not bit-identical across implementations, so the tolerance must be stated upfront. Report the conditions — hardware, input sizes, precision, and how you timed it — since a result without them cannot be reproduced. Two things make timing GPU code tricky: the first iterations are slower (warm-up), and the GPU runs asynchronously, so the CPU may reach the end of the loop before the GPU has finished. Week 3 covers how to time correctly.

Locate the work. Identify the constraint. Verify the improvement.

The same questions, at every scale

Scale	A system question
Data pipeline	Is computation waiting for data?
GPU kernel	Is performance limited by computation or memory access?
Coding agents	Does the generated kernel pass independent correctness and performance tests?
Distributed training	How much time is spent communicating?
LLM service	How do memory and request queues affect latency?
Shared cluster	How should competing jobs share resources?

Each later week takes one row of this table and answers it properly. The method is always the one from the previous slide.

Locate the work. Identify the constraint. Verify the improvement.

This week's reading

Two papers on the systems underneath PyTorch. They go deeper than the lecture; read them alongside the course, about two hours or three a week. Each week will have its own reading.

PyTorch: An Imperative Style, High-Performance Deep Learning Library

Paszke et al., NeurIPS 2019

Why PyTorch is designed the way it is, and what that costs: eager execution, autograd recorded at run time, asynchronous GPU execution, the caching memory allocator, multi-process data loading.

Read: Sections 4–6, about two hours.

Connects to: Slides 22, 28 and 30–31 today.

arxiv.org/abs/1912.01703

PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation

Ansel et al., ASPLOS 2024

What the framework did about that cost: `torch.compile` captures the graph from Python bytecode, compiles it into fused GPU kernels, and falls back to eager execution when it cannot.

Read: Sections 1–2 and the evaluation now; the middle sections when we reach compilers.

Connects to: Slides 28, 30 and 32 today.

https://www.youtube.com/watch?v=3LW4_fwD2Z8

Read 2019 first, then 2024. From each paper, reproduce one number — a memory saving, a speedup — with your own arithmetic or your own measurement.

Questions and next steps

Today

Tensors, training execution and performance reasoning

Next week

Data pipelines — Yang Cao

Week 3

GPU architecture and CUDA programming

After the lecture

- Run the training loop from today on your own machine (CPU is fine) and time one iteration. Which stage do you think dominates?
- Read the PyTorch tutorials on Tensors and on Autograd; every term in them appeared today.
- Repeat both quick checks for a two-layer model: `nn.Flatten()`, `nn.Linear(784, 256)`, `nn.ReLU()`, `nn.Linear(256, 10)`. Which tensors change?

Questions?

Ask on Edstem after the lecture — link on the Learn course page.

Appendix: answers to the quick checks

Double the batch size (64 → 128): which tensors grow?

B — X and S.

X becomes [128, 784] and S becomes [128, 10]. W [10, 784], b [10] and the gradient of W keep their shapes. Inputs and activations scale with the batch; parameters and their gradients do not — which is why the input can fit on the GPU while training still runs out of memory.

Remove optimizer.zero_grad(): what happens?

B — gradients accumulate across batches.

The program runs without error, but PyTorch adds each new gradient to the stored one, so every update uses a mixture of the current and all previous gradients. The loop still runs; it no longer does the learning process we intended. Accumulating on purpose is a real technique, but then step() is called less often.

This page is for the exported slides; it is not shown in the lecture.