Tensor Workbook: The Hidden Champions of the AI Revolution

Dr. Yves J. Hilpisch[1]

December 11, 2025 (work in progress)

[1]Get in touch: `https://linktr.ee/dyjh`. Web page `https://hilpisch.com`. Research, structuring, drafting, and visualizations were assisted by GPT 5.1 as a co-writing tool under human direction.

Contents

1 How to Use This Workbook **2**

2 Concept Map and Learning Goals **3**

3 Tensor Foundations: From Scalars to Multilinear Maps **5**
- 3.1 Scalars, Vectors, Matrices, Tensors . . . 5
- 3.2 Core Tensor Operations . . . 6
- 3.3 Multilinear Maps and Einstein Summation . . . 6

4 Tensors in NumPy **7**
- 4.1 NumPy Arrays as Tensors . . . 7
- 4.2 Broadcasting, Views, and Memory Layout . . . 8

5 Tensors in PyTorch **9**
- 5.1 Torch Tensors, Devices, and Autograd . . . 9
- 5.2 A Two-Layer Network as a Tensor Program . . . 10

6 Tensor Programs Behind Deep Learning **11**
- 6.1 Everything Is a Tensor Operation . . . 11
- 6.2 Convolutions, Matrix Multiplication, and Batching . . . 12

7 Self-Attention as a Tensor Pipeline **13**
- 7.1 Mathematical Definition . . . 13
- 7.2 Tensor Shapes in a Transformer Block . . . 14
- 7.3 Implementing Self-Attention in PyTorch . . . 15

8 Tensors and the AI Hardware–Software Stack **16**
- 8.1 GPTs and LLMs as Giant Tensor Graphs . . . 17
- 8.2 GPUs, TPUs, and Tensor Cores . . . 17
- 8.3 Deep Learning Frameworks as Tensor Compilers . . . 18

9 Practice Projects and Further Explorations **18**

1 How to Use This Workbook

This workbook is a self-contained, hands-on companion to *A Short History of Computer Science: From Commodore C64 to Cloud AI*. It zooms in on tensors as the hidden data structures and operations powering modern deep learning: from small NumPy arrays on your laptop to massive multi-head self-attention layers running on GPU and TPU clusters. The tone is intentionally conversational, but every story is backed by equations and runnable code.

The primary audience is a curious reader with basic linear-algebra and Python literacy who wants to connect three layers of understanding: the mathematics of tensors, the concrete APIs in NumPy and PyTorch, and the way large language models (LLMs, large language models) such as GPTs are built almost entirely from tensor operations. Mentors and instructors can also use the workbook as a lab-style module inside a broader course on machine learning or the history of artificial intelligence.

You will get the most value if you treat the workbook as a *studio* rather than a *lecture*. Read a short section, then immediately switch to your favourite Python environment and reproduce the examples, extending them by one or two small experiments of your own. Whenever you see a shape such as (B, L, d_{model}) in the text, ask: "What are the concrete numbers in this example? How big is B? What happens if I double L?"

Mentors and instructors can either run the workbook as a compact two- or three-session mini-course, or sprinkle individual activities into a longer class. The early sections (up to and including *Tensors in NumPy*) work well as a gentle introduction for mixed audiences; the later sections on self-attention and hardware are best suited to readers who already saw a basic deep-learning model at least once.

The compiled PDF of this workbook is available at `https://hilpisch.com/tensors.pdf`.

Suggested Learning Paths

Different readers can take different routes through the material:

- If you mainly care about *using* deep-learning libraries, focus on the NumPy and PyTorch sections first, then skim the mathematical subsections for intuition.
- If you are mathematically curious and want to know what tensors "really" are, read the foundations carefully and work through the index-based derivations before opening a notebook.
- If you are a mentor or instructor, pick one worked example (for instance, the self-attention implementation) and build a live-coding session around it, using the other sections as background reading.

Whichever path you choose, keep a notebook—digital or paper—where you write down shapes, index ranges, and small sketches of the diagrams. This practice pays off quickly once tensors grow beyond two dimensions.

Prerequisites and Setup

You should be comfortable with:

- basic linear algebra (vectors, matrices, matrix–vector products),
- elementary Python syntax (variables, functions, loops, list comprehensions),
- and running code either in a Jupyter notebook, IPython shell, or simple Python script.

For the hands-on parts you need recent versions of `numpy` and `torch`. A typical environment check in an IPython session looks like this:

Environment check in IPython

```
In [1]: import numpy as np
In [2]: import torch

In [3]: np.__version__
Out[3]: '1.26.0'  # your version can differ

In [4]: torch.__version__
Out[4]: '2.1.0'  # any reasonably recent 2.x is fine
```

What You Will Build

By the end of this workbook you will have:

- a precise mental model of tensors as multidimensional arrays and multilinear maps,
- working NumPy and PyTorch code that manipulates tensors with clear control over shapes and broadcasting,
- a from-scratch implementation of a small self-attention block written purely in tensor operations,
- and a "big picture" view of how GPT-style LLMs, GPUs, and TPUs are optimised engines for tensor programs.

The rest of the workbook follows a simple rhythm: each new theoretical idea is immediately grounded in a short code experiment, and each code experiment is interpreted back in terms of tensors, indices, and shapes. You are encouraged to pause at the end of every section and ask yourself: "Can I explain this in my own words? Could I sketch this diagram from memory? Could I reimplement this function without copy–paste?"

2 Concept Map and Learning Goals

This section lays out the conceptual map of the workbook so you always see how abstract definitions, code, and hardware connect. The idea is simple: *tensors* sit in the middle; four perspectives—mathematics, implementation, models, and hardware—form the surrounding ring. The concept map in Figure 1 is not meant to be memorised; it is a visual reminder of the conversations that will keep happening between these perspectives.

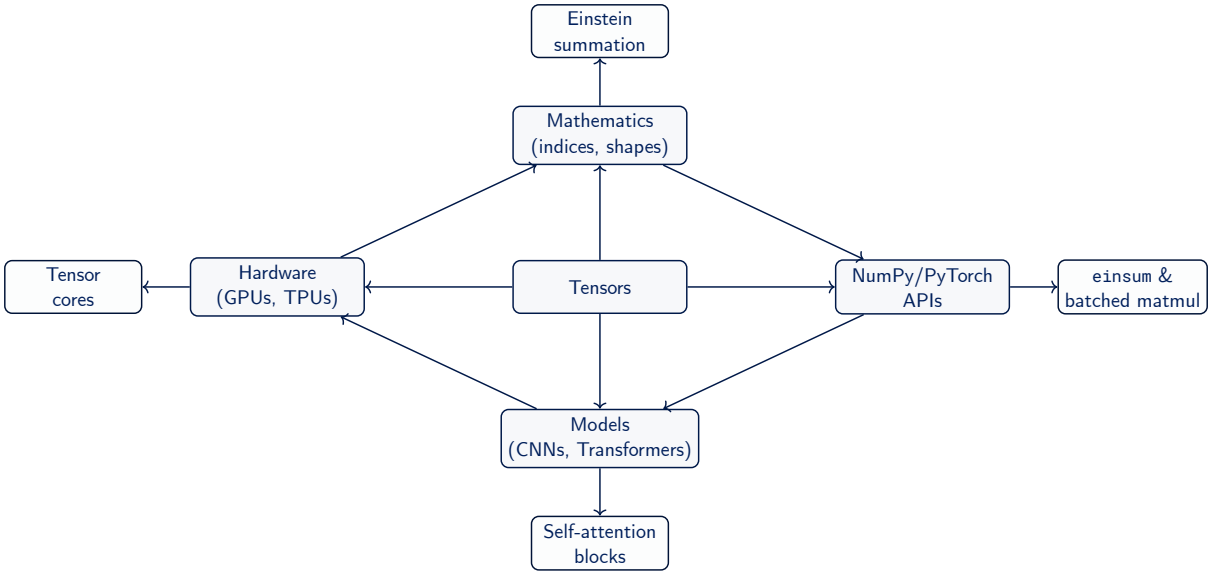

Figure 1: Concept map for the tensor workbook: tensors link precise mathematics, array libraries (NumPy/PyTorch), model architectures (especially Transformers), and modern hardware such as GPUs and TPUs.

Starting in the north of the diagram, the *Mathematics* node represents the clean, index-based view of tensors: arrays with shapes, indices that range over those shapes, and operations such as contractions and outer products that turn some indices into others. When we talk about Einstein summation or multilinear maps, we are standing in this part of the map. The small "Einstein summation" box above reminds you that there is a compact symbolic way to write most of the operations that drive deep learning.

Moving clockwise, the *NumPy/PyTorch APIs* node is where these mathematical ideas become concrete functions: `np.einsum`, `torch.matmul`, broadcasting rules, reshapes, transposes. Every arrow between "Mathematics" and "NumPy/PyTorch" is bidirectional: a tensor identity from the mathematical side often translates into a one-line performance improvement in code, and a puzzling runtime error about incompatible shapes is usually resolved by revisiting the underlying index story.

The *Models* node covers familiar architectures: convolutional neural networks (CNNs), recurrent networks, and, most prominently for this workbook, Transformer blocks with self-attention. Here you will see how to read a model diagram as a tensor program: each edge carries a tensor of known shape, and each box applies a well-defined tensor operation to those inputs.

Finally, the *Hardware* node reminds us that everything ultimately runs on physical devices. GPUs and TPUs are fast not because they understand "intelligence" but because they can push vast numbers of multiply–accumulate operations on large tensors through specialised circuits called tensor cores. When we pay attention to batch sizes, memory layouts, and precision (for example using `float16` instead of `float32`), we are operating along the edge between "Models" and "Hardware."

Learning Goals at a Glance

Across the whole workbook you will:

- learn to translate between index notation, diagrams, and concrete tensor shapes,
- practice implementing common tensor patterns in both NumPy and PyTorch,
- recognise self-attention and related mechanisms as short sequences of tensor operations,
- and develop an intuition for how hardware constraints such as memory and parallelism shape modern deep-learning practice.

Keep Figure 1 in mind as a map of these goals: each new section will light up one or more of the boxes and arrows.

With the map in place we can now slow down and build a careful foundation for what tensors are, both as mathematical objects and as concrete arrays in memory.

3 Tensor Foundations: From Scalars to Multilinear Maps

Before touching any code we want a clean, notation-light picture of what tensors are. The emphasis is on shapes, ranks, and the way indices move—all things you can see and debug in actual arrays. The goal of this section is not to be maximally general, but to build a sturdy mental model that stays useful when you later read more abstract texts on differential geometry or representation theory.

3.1 Scalars, Vectors, Matrices, Tensors

At the lowest level a *scalar* is just a single number, a *vector* is a one-dimensional array of numbers, and a *matrix* is a two-dimensional array. A tensor is the natural generalisation: an array with an arbitrary number of indices. In practice most of the tensors you meet in deep learning have between one and four indices: batches of sequences of embedding vectors, grids of image patches, stacks of feature maps.

We use the following basic conventions throughout the workbook:

- Latin indices $i, j, k, \ldots$ range over positions in one dimension, for example $i = 1, \ldots, n$.
- Bold lowercase letters such as $\boldsymbol{x}$ denote vectors, bold uppercase letters such as $\boldsymbol{A}$ denote matrices.
- Higher-order tensors are written as $\mathcal{T}$ or with explicit indices, for example $T_{ijk\ell}$.

In index notation we might write a matrix–vector product as

$$y_i = \sum_{j=1}^{n} A_{ij} x_j,$$

and a simple tensor contraction as

$$z_{ik} = \sum_{j=1}^{n} T_{ijk} x_j.$$

Here the index i or the pair (i, k) is *free* (it survives the computation), while j is *summed* or *contracted* away. This way of thinking lines up perfectly with the way shapes change in code: a contraction over index j removes the corresponding dimension from the result.

Two Complementary Views of a Tensor

It is helpful to keep two mental pictures of a tensor in play:

- a tensor as a *box of numbers* arranged in a grid with a certain shape, which is how libraries such as NumPy and PyTorch implement them in memory,
- and a tensor as a *multilinear function* that eats several vectors and spits out another vector or scalar, which is how many mathematical texts introduce them.

In finite dimensions these views are equivalent once you choose a basis, but they highlight different aspects. In this workbook we mostly stay with the "box of numbers" view while borrowing just enough of the multilinear perspective to understand why certain index patterns keep returning.

3.2 Core Tensor Operations

Most of deep learning is built from a surprisingly small vocabulary of tensor operations. The core ones are:

- elementwise operations such as addition and nonlinearities, e.g. $\boldsymbol{y} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$,
- matrix and batched matrix multiplications, e.g. $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}^\top$,
- tensor contractions and outer products, implemented via Einstein summation,
- reshaping, transposing, and broadcasting to align shapes for computation.

In Einstein notation a batched matrix multiplication can be written compactly as

$$Y_{bik} = \sum_j X_{bij} W_{bjk},$$

where b indexes the batch. On the surface this looks more complicated than writing $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}$, but the index notation pays for itself when we start to mix several batches, heads, and sequence positions. Each repeated index corresponds to a sum, and each non-repeated index corresponds to an axis in the output tensor.

From the implementation side, the same computation appears in code as a single call to a highly optimised kernel. For example, in NumPy one might write `Y = X @ W` when `X` and `W` have compatible shapes. In PyTorch the identical code runs on CPU or GPU depending on where the tensors live. The whole art of efficient deep learning consists of arranging your data so that as many useful operations as possible can be expressed in this compact batched form.

3.3 Multilinear Maps and Einstein Summation

Behind the compact formulas above sits a slightly more abstract but very powerful idea: a tensor as a multilinear map. Concretely, a rank-k tensor over $\mathbb{R}^n$ can be thought of as a function

$$T : \underbrace{\mathbb{R}^n \times \cdots \times \mathbb{R}^n}_{k \text{ copies}} \to \mathbb{R}$$

that is linear in each argument when the others are held fixed. In coordinates this map is described by an array $T_{i_1 \ldots i_k}$, and feeding the tensor a tuple of vectors $(\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(k)})$ amounts to computing

$$T(\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(k)}) = \sum_{i_1, \ldots, i_k} T_{i_1 \ldots i_k} v^{(1)}_{i_1} \cdots v^{(k)}_{i_k}.$$

Einstein summation notation simply hides the explicit summation symbols by adopting the convention "sum over any index that appears twice." In that language the expression above becomes

$$T(\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(k)}) = T_{i_1 \ldots i_k} v^{(1)}_{i_1} \cdots v^{(k)}_{i_k},$$

which is shorter but encodes the same computation. Later, when we meet the `einsum` functions in NumPy and PyTorch, you will see this notation show up almost verbatim in code.
Armed with this foundation we are ready to turn to concrete incarnations of tensors in code, starting with NumPy arrays as our default mental model for "tensors on a CPU." The next section makes that bridge explicit.

4 Tensors in NumPy

NumPy arrays are a concrete incarnation of tensors on the CPU: contiguous (or strided) blocks of memory tagged with a shape and a data type. This section makes that picture fully explicit and links each piece of NumPy vocabulary to the tensor concepts from the previous section.

The three questions to keep in mind are:

- What is the *shape* of this array, and which index letter $(i, j, k, \ldots)$ corresponds to which axis?
- Does this operation change the underlying data, or only how we *view* the same data (via reshapes and transposes)?
- When shapes differ, is NumPy creating a *broadcasted* view, or is it making an explicit copy?

By the end of this section you should be able to answer these questions almost automatically whenever you see a new array expression.

4.1 NumPy Arrays as Tensors

We start with a few simple experiments in IPython to connect shapes, ranks, and axes to things you can print and inspect. The goal is to train your eye to read `.shape` and `.ndim` as quickly as you read a short sentence.

Exploring tensor ranks with NumPy

```
In [1]: import numpy as np

In [2]: scalar = np.array(3.14)
In [3]: vector = np.arange(5)
In [4]: matrix = np.arange(6).reshape(2, 3)
In [5]: tensor = np.arange(24).reshape(2, 3, 4)

In [6]: scalar.shape, scalar.ndim
Out[6]: ((), 0)

In [7]: vector.shape, vector.ndim
Out[7]: ((5,), 1)

In [8]: matrix.shape, matrix.ndim
Out[8]: ((2, 3), 2)

In [9]: tensor.shape, tensor.ndim
Out[9]: ((2, 3, 4), 3)
```

Here `ndim` corresponds to the tensor rank (number of indices), and `shape` records the range of each index. Throughout the workbook we keep these visible as a cheap debugging tool.

Naming Axes Explicitly

Whenever possible, give each axis a short verbal label in your head:

- for a vector of shape $(d,)$ you might say "feature index $i = 1, \ldots, d$,"
- for a matrix of shape (n, d) you might say "row index $i = 1, \ldots, n$, feature index $j = 1, \ldots, d$,"
- for a tensor of shape (B, L, d_{model}) you might say "batch index b, position index ℓ, feature index k."

This tiny habit makes it much easier to translate between index notation such as $X_{b\ell k}$ and concrete arrays such as `X[b, l, k]`.

One of the most common sources of bugs in scientific Python code is silently mixing up axis orders. In this workbook we will always spell out the intended shape when introducing a new tensor and will choose variable names that make the axis roles as clear as possible.

4.2 Broadcasting, Views, and Memory Layout

Broadcasting lets NumPy reuse data without copying by treating dimensions of size 1 as if they were "stretched" to match another tensor. Views and strides determine how logical indices map to memory addresses. At first this can feel like magic; here we slow down and watch the shapes carefully.

A short script-style example illustrates how reshaping and broadcasting work together.

Broadcasting and reshaping in NumPy

```
import numpy as np

batch_size = 2  # number of independent sequences (B)
seq_len = 3  # length of each sequence (L)
d_model = 4  # features per position (d_model)

x = np.arange(batch_size * seq_len * d_model).reshape(batch_size, seq_len, d_model)  # toy
↪  tensor of shape (B,L,d_model)

  # Add a bias vector of shape (d_model,) to every position
bias = np.linspace(0.0, 1.0, d_model)  # one bias value per feature dimension
y = x + bias  # broadcasting: bias is virtually reshaped to (1,1,d_model) and added to every
↪  (b,l,:) slice

print("x shape:", x.shape)  # (2, 3, 4)
print("bias shape:", bias.shape)  # (4,)
print("y shape:", y.shape)  # still (2, 3, 4); broadcasting does not change the output shape
```

In this example the bias vector has shape $(d_{\text{model}},)$. When NumPy sees the expression `x + bias`, it first conceptually reshapes `bias` to shape $(1, 1, d_{\text{model}})$ (by adding two singleton axes) and then stretches these singleton axes to match the $(\texttt{batch\_size}, \texttt{seq\_len}, d_{\text{model}})$ axes of `x`. No data is duplicated; NumPy simply adjusts the way it computes memory addresses.

Copies vs Views

To deepen your broadcasting intuition, it helps to know when arrays share memory:

- Simple reshapes (using `reshape` or `ravel`) often return *views* that share data with the original array.
- Slices such as `x[:, 0]` or `x[..., :2]` are also usually views; modifying them writes back into `x`.
- Operations that change the number of elements or require reordering them in memory (for example `np.concatenate` or some transposes) necessarily allocate new arrays.

You can check this by inspecting the `.base` attribute of an array or using `np.shares_memory`. For most deep-learning workloads it is safe to rely on the high-level broadcasting rules, but when performance or memory usage becomes critical, understanding views versus copies is invaluable.

When you write your own tensor programs it is often enough to reason in terms of shapes and broadcasting rules; NumPy takes care of efficient memory access under the hood. The PyTorch section will mirror this picture almost one-to-one, adding devices and gradients on top.

5 Tensors in PyTorch

PyTorch tensors mirror NumPy arrays but add automatic differentiation and explicit device placement. Conceptually they are the same objects: shaped arrays of numbers with operations applied in bulk. The main new ingredients are:

- a *device* (CPU, GPU, or other accelerator) on which the tensor lives,
- a flag `requires_grad` indicating whether operations on this tensor should be recorded for backpropagation,
- and a *computation graph* that links intermediate tensors so gradients can flow backwards.

Understanding these pieces pays off quickly when you debug training scripts or port models between machines.

5.1 Torch Tensors, Devices, and Autograd

We begin by constructing simple tensors on CPU and, if available, on GPU. The focus is on being explicit about `dtype`, `device`, and `requires_grad`. In the example below we stay on CPU so that the code runs everywhere, but the pattern is identical for GPUs.

Creating and moving PyTorch tensors

```
import torch

x = torch.linspace(0.0, 1.0, steps=5, dtype=torch.float32, requires_grad=True)  # 1D tensor on
↪  CPU with gradient tracking
W = torch.randn(5, 3, dtype=torch.float32)  # weight matrix mapping R^5 -> R^3
b = torch.zeros(3, dtype=torch.float32)  # bias vector in R^3

y = x @ W + b  # affine map: first matrix-vector product, then broadcasted bias addition (shape:
↪  (3,))

print("x:", x)  # shows data, dtype, and device of the input tensor
```

```
print("W shape:", W.shape)  # (5, 3)
print("y shape:", y.shape)  # (3,)

y.sum().backward()  # backpropagate through the computation graph to obtain dy/dx
print("x.grad:", x.grad)  # gradient of the scalar y.sum() w.r.t. each element of x
```

Here `x` is a one-dimensional tensor with five equally spaced values between 0 and 1. The matrix `W` and bias `b` define a simple affine map, and the expression `y = x @ W + b` is the same batched matrix multiplication we saw in the NumPy section, now with gradients attached. Calling `y.sum().backward()` asks PyTorch to differentiate this scalar with respect to every element of `x`; the resulting gradient appears in `x.grad`.

Devices and dtypes in practice

A few small habits make PyTorch tensor code more robust:

- Inspect `x.device` and `x.dtype` when you construct new tensors; mismatches are a common source of runtime errors.
- When moving models to GPU, use patterns such as `device = torch.device("cuda" if torch.cuda.is_available() else "cpu")` and then call `to(device)` on both model and data tensors.
- For large models, consider reduced-precision dtypes such as `torch.float16` or `torch.bfloat16`; they halve memory use and enable specialised hardware paths on modern GPUs and TPUs.

Developing a habit of checking shape, device, and dtype together—for example by printing them in a small utility function—can save hours of debugging time.

Once you are comfortable with gradients on small tensors it becomes easier to trust that the same machinery works at the scale of millions or billions of parameters in an LLM. The underlying engine is the same: record tensor operations in a computation graph, then traverse that graph in reverse to accumulate gradients.

5.2 A Two-Layer Network as a Tensor Program

Instead of diving into full training loops we treat a tiny feed-forward network as a pure tensor function: input in, output out, all expressed as compositions of matrix multiplications and elementwise nonlinearities.

Forward pass of a two-layer MLP

```
import torch
import torch.nn.functional as F

batch_size = 4  # number of examples processed in parallel
d_in = 8  # input feature dimension
d_hidden = 16  # width of the hidden layer
d_out = 4  # output feature dimension (for example, number of classes)

x = torch.randn(batch_size, d_in)  # input tensor X in R^{B x d_in}
W1 = torch.randn(d_in, d_hidden)  # first-layer weight matrix
b1 = torch.zeros(d_hidden)  # first-layer bias, broadcast across the batch
W2 = torch.randn(d_hidden, d_out)  # second-layer weight matrix
b2 = torch.zeros(d_out)  # second-layer bias
```

```
h = F.relu(x @ W1 + b1)  # hidden activations H = ReLU(X W1 + b1), shape (B, d_hidden)
logits = h @ W2 + b2  # output tensor Y = H W2 + b2, shape (B, d_out)
```

Every line here is a tensor operation. The tensors `W1`, `b1`, `W2`, and `b2` would become trainable parameters in a real model; the input `x` and intermediate activation `h` are activations that exist only for the duration of the forward pass (and its corresponding backward pass). A full Transformer block looks more complex, but structurally it is just a carefully wired stack of similar pieces—linear maps, nonlinearities, and normalisation layers—all acting on high-rank tensors instead of simple matrices.

From NumPy to PyTorch and Back

When you already have NumPy code, porting it to PyTorch is often a matter of replacing:

- `np.array`, `np.zeros`, `np.ones` with their `torch` counterparts,
- `@`, `+`, and elementwise functions with the same operators on `torch.Tensor`,
- and optionally adding `requires_grad=True` where you want to learn parameters.

You can move data from NumPy to PyTorch via `torch.from_numpy` (sharing memory) and back via `tensor.numpy()`. Just remember that gradients are only tracked on PyTorch tensors, not on bare NumPy arrays.

6 Tensor Programs Behind Deep Learning

With both NumPy and PyTorch in hand we can step back and see deep learning models as *tensor programs*: directed acyclic graphs whose nodes are tensor operations and whose edges carry tensors of well-defined shape.

6.1 Everything Is a Tensor Operation

From this perspective common layers turn into short formulas:

- embedding layers are simple lookups in a learned table followed by gathering rows,
- fully connected layers are matrix multiplications plus bias vectors,
- convolutions can be expressed as specialised tensor contractions,
- self-attention layers combine a handful of batched matrix multiplications and softmax operations.

In other words, what looks like a zoo of different architectures can be reduced to a surprisingly small toolkit of tensor primitives. Modern GPU and TPU libraries provide highly optimised kernels for each of these operations; models scale by reusing the same primitives at larger shapes.

The Computation Graph View

It is useful to imagine a neural network as a computation graph:

- Nodes correspond to tensor operations such as matrix multiplications, elementwise nonlinearities, or reshapes.
- Edges carry tensors from one operation to the next.
- Training amounts to running a forward pass through this graph, computing a loss tensor, and then running a backward pass that accumulates gradients along the edges in reverse.

Frameworks such as PyTorch build and traverse this graph for you, but keeping the picture in mind clarifies why certain operations (like in-place modifications of tensors that require gradients) are discouraged: they can break the assumptions of the underlying graph.

6.2 Convolutions, Matrix Multiplication, and Batching

Historically, linear algebra libraries such as BLAS and LAPACK focused on dense matrix operations. Deep learning frameworks extend this mindset by making batched matrix multiplications and convolutions first-class citizens.

Batched matrix multiplication in PyTorch

```
import torch

batch = 32  # batch size (for example, number of sequences or images)
n = 64  # rows of each A^{(b)} matrix
m = 128  # shared inner dimension summed over
k = 256  # columns of each B^{(b)} matrix

A = torch.randn(batch, n, m)  # stack of A^{(b)} matrices in R^{n x m}
B = torch.randn(batch, m, k)  # stack of B^{(b)} matrices in R^{m x k}
C = A @ B  # batched matrix multiplication: for each b, compute A[b] @ B[b], shape (batch, n, k)

print("C shape:", C.shape)  # verifies the expected (batch, n, k) shape
```

Under the hood a single call like `A @ B` triggers a highly tuned kernel that saturates the GPU or TPU with parallel tensor operations. A two-dimensional convolution can be expressed in a very similar way: by rearranging patches of the input into a batch of vectors and multiplying them by a matrix of filter weights. In practice you rarely do this rearrangement by hand; libraries provide `conv` functions that encapsulate the pattern. Conceptually, however, both convolutions and attention blocks live in the same world of "take many small inner products in parallel."

Reading Model Summaries as Tensor Programs

When you see a printed model summary—for example from `torchinfo` or a similar tool—try to read each line in tensor language:

- note the input and output shapes of each layer,
- identify which primitive tensor operation is being used (dense, convolution, normalisation, attention),
- and mentally connect it back to the formulas in earlier sections.

Over time this exercise makes model architectures feel far less mysterious; they become slightly larger and more structured versions of the small tensor programs you have already written by hand.

With this tensor-program mindset in place we are ready to examine one of the central building blocks of modern LLMs in more detail: the self-attention mechanism itself.

7 Self-Attention as a Tensor Pipeline

Self-attention is the canonical example where the tensor view becomes indispensable. The same pattern appears in Transformers for language, images, audio, and multimodal models. Conceptually, each token in a sequence looks at all other tokens, decides how much to "pay attention" to each of them, and then replaces its own representation by a weighted average of the others. Every one of these steps is expressed as a small tensor operation.

7.1 Mathematical Definition

In its simplest form, scaled dot-product attention takes query, key, and value tensors $Q, K, V \in \mathbb{R}^{L\times d}$ for a sequence of length L and computes

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V.$$

Here Q contains one query vector per token, K contains one key vector per token, and V contains one value vector per token, all in the same space $\mathbb{R}^d$. The product $QK^\top \in \mathbb{R}^{L\times L}$ collects all pairwise dot products between queries and keys; after scaling by $\sqrt{d}$ and applying a row-wise softmax, each row becomes a probability distribution over positions. Multiplying by V on the right then forms, for each token, a weighted average of all value vectors in the sequence.

In practice everything is batched and split into multiple heads, so the actual tensors have shape (B, H, L, d_h), where B is the batch size, H the number of heads, and $d_h = d_{\text{model}}/H$ the per-head dimension. Each head has its own learned projections and therefore focuses on different kinds of relationships (for example, local syntax versus long-range topic structure).

Causality and Attention Masks

Language models that generate text from left to right must respect causality: token t may only look at positions $1, \dots, t$, not at future tokens. This is enforced by an *attention mask*, a tensor $M \in \mathbb{R}^{L \times L}$ whose entries are 0 for allowed attention pairs and $-\infty$ (or a large negative number) for forbidden ones. Before applying softmax we add the mask to the raw scores:

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right).$$

In code this mask is just another tensor with a carefully chosen shape, broadcast across batch and head dimensions.

7.2 Tensor Shapes in a Transformer Block

Keeping track of shapes is half the battle. A typical path through a single self-attention block looks like this:

- input token embeddings of shape (B, L, d_{model}),
- linear projections to queries, keys, values of shape (B, H, L, d_h),
- attention weights as a tensor of shape (B, H, L, L),
- attended values of shape (B, H, L, d_h),
- output reassembled to shape (B, L, d_{model}).

Each arrow corresponds to a matrix multiplication, a reshape, or an elementwise nonlinearity—no exotic operations required.

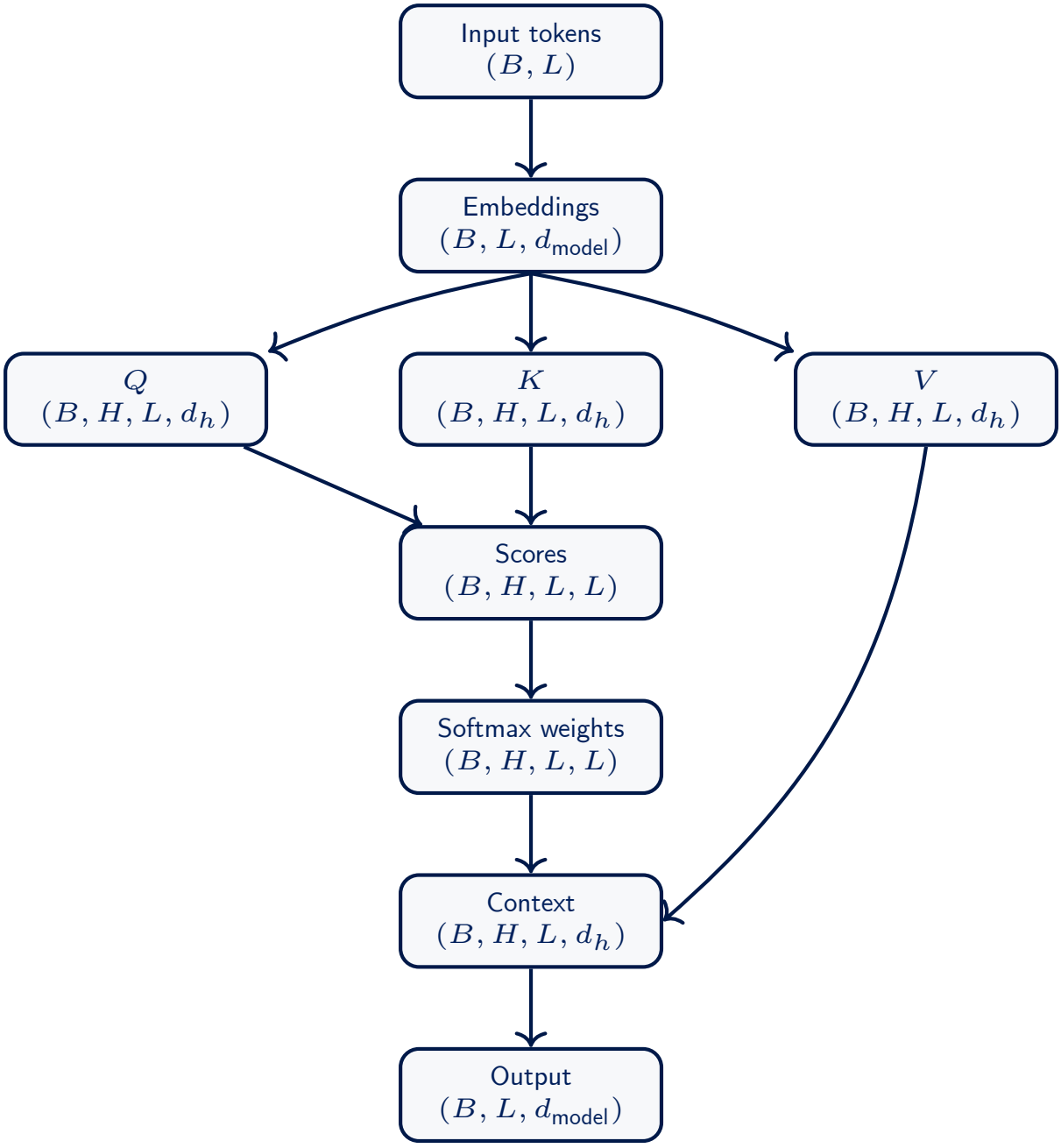

Figure 2: Shape-centric view of a self-attention block: each box is a tensor with explicit shape, and each arrow is a tensor operation such as a linear projection, batched matrix multiplication, or softmax.

The diagram in Figure 2 is deliberately schematic: real Transformer blocks add layer normalisation, skip connections, and position encodings around this core. Nevertheless, if you can track how shapes change along these arrows, you already understand the beating heart of GPT-style models.

7.3 Implementing Self-Attention in PyTorch

We now translate the mathematical definition into a compact PyTorch implementation that exposes all relevant shapes.

Minimal multi-head self-attention in PyTorch

```python
import torch
import torch.nn.functional as F

B, L, d_model = 2, 4, 8  # batch size, sequence length, and model width
H = 2  # number of attention heads
d_h = d_model // H  # dimension per head

x = torch.randn(B, L, d_model)  # input token representations X in R^{B x L x d_model}

W_q = torch.randn(d_model, d_model)  # projection matrix for queries
W_k = torch.randn(d_model, d_model)  # projection matrix for keys
W_v = torch.randn(d_model, d_model)  # projection matrix for values
W_o = torch.randn(d_model, d_model)  # output projection after combining heads

Q = x @ W_q  # project inputs to queries Q in R^{B x L x d_model}
K = x @ W_k  # project inputs to keys K in R^{B x L x d_model}
V = x @ W_v  # project inputs to values V in R^{B x L x d_model}

def reshape_heads(t):
    return t.view(B, L, H, d_h).transpose(1, 2)  # split last dim into heads and move H before L
    ↪ -> (B,H,L,d_h)

Qh = reshape_heads(Q)  # queries per head
Kh = reshape_heads(K)  # keys per head
Vh = reshape_heads(V)  # values per head

scores = (Qh @ Kh.transpose(-1, -2)) / (d_h ** 0.5)  # scaled dot products between queries and
↪ keys, shape (B,H,L,L)
weights = F.softmax(scores, dim=-1)  # attention weights: per (B,H,L) row, probabilities over key
↪ positions
context = weights @ Vh  # weighted sums of values per query position -> context tensor in R^{B x
↪ H x L x d_h}

context = context.transpose(1, 2).contiguous().view(B, L, d_model)  # reassemble heads back into
↪ a single d_model vector
out = context @ W_o  # final linear projection; output has same shape as input (B, L, d_model)
```

The code includes no "magic" beyond reshaping and batched matrix multiplication. Yet scaled up and repeated, this is the core inner loop of GPT-style LLMs. In production systems several practical refinements are added:

- numerically stable softmax implementations that subtract the maximum score along each row before exponentiation,
- fused kernels that combine multiple tensor operations (for example, projection and reshaping) into a single GPU kernel launch,
- and attention variants such as multi-query and grouped-query attention that reduce memory use by sharing keys and values across heads.

All of these remain, at their core, tensor programs operating on the same shapes you have just explored.

8 Tensors and the AI Hardware–Software Stack

At this point we can zoom out again and see tensors as the shared language between abstract models, software frameworks, and specialised hardware. Thinking in tensors does not just clarify the mathematics; it also clarifies why certain hardware designs and software abstractions have become so dominant in the current AI wave.

8.1 GPTs and LLMs as Giant Tensor Graphs

Large language models such as GPT-4 and beyond can be viewed as enormous directed acyclic graphs of tensor operations. Parameters live in large tensors (weight matrices and bias vectors); activations are temporary tensors flowing through layers; training consists of repeatedly applying forward and backward passes to these graphs using gradient-based optimisation.

From this perspective a full GPT model looks like a repetition of just a few structural motifs:

- embedding lookups that map token indices to vectors,
- alternating blocks of self-attention and position-wise feed-forward networks,
- layer normalisation and residual (skip) connections that keep gradients well behaved,
- and a final projection from hidden states back into vocabulary logits.

Each motif is a compact tensor program similar to the self-attention snippet you implemented earlier; scaling up means increasing the dimensions of these tensors (more layers, wider hidden states, longer sequences) and distributing them across many devices.

Scale as Shape

When people talk about the "size" of an LLM they often quote the number of parameters (for example, billions or trillions). In tensor language this is simply the total number of entries across all parameter tensors. Doubling model size usually means:

- increasing some dimension of a core tensor (for example d_{model} or the number of layers),
- which in turn changes the shapes of associated weight matrices and activations,
- and therefore increases both compute and memory in a predictable, tensor-based way.

Thinking in shapes helps you reason about these trade-offs more quantitatively: a twofold increase in d_{model} squares the cost of some matrix multiplications but only linearly increases the cost of others.

8.2 GPUs, TPUs, and Tensor Cores

Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) are optimised for throughput on dense tensor operations. Rather than executing individual scalar instructions, they organise work in blocks and warps that apply the same operation to many elements in parallel. Vendor libraries such as cuBLAS, cuDNN, and XLA expose this power through high-level tensor primitives.

At the hardware level, so-called tensor cores or matrix units implement small fixed-size matrix multiplications (for example 16×16 tiles) extremely efficiently. Larger tensor operations, such as the batched matrix multiplications in attention, are decomposed into many of these tiles. The closer your tensor program is to "giant collections of independent matrix multiplications," the more easily it can saturate these specialised units.

Memory–Bandwidth–Precision Trade-offs

Performance on modern accelerators is often limited less by raw compute and more by:

- memory capacity (how many parameter and activation tensors fit on a device),
- memory bandwidth (how quickly those tensors can be moved between memory and compute units),
- and numerical precision (for example `float32` vs `bfloat16` vs `int8`).

Techniques such as activation checkpointing, gradient accumulation, mixed-precision training, and quantisation are all tensor-level tricks for navigating these constraints: they change which tensors are stored, when they are recomputed, and in which datatype.

8.3 Deep Learning Frameworks as Tensor Compilers

Frameworks like PyTorch, TensorFlow, and JAX can be seen as tensor compilers: they let you write high-level tensor programs which are then lowered to highly optimised kernels on CPUs, GPUs, or TPUs. Once you recognise attention, convolutions, and feed-forward networks as combinations of a small set of tensor operations, it becomes easier to reason about performance, memory, and numerical stability.

Eager-execution frameworks (such as vanilla PyTorch) build the computation graph dynamically as you run Python code; tracing and compilation tools (such as TorchDynamo, XLA, and JAX's `jit`) capture larger chunks of that graph and compile them into fused kernels. Both styles share the same underlying unit of work: a graph whose nodes are tensor operations with specific shapes.

Reading Profiler Output

If you profile a deep-learning workload you will typically see a small set of kernels dominating runtime:

- general matrix multiplications (often labelled as `gemm` or `matmul`),
- convolution kernels,
- and a handful of elementwise operations (nonlinearities, normalisation steps).

Interpreting these lines as tensor operations helps you decide where to focus optimisation efforts: speeding up a single large matrix multiplication can matter far more than micro-optimising dozens of tiny elementwise operations.

9 Practice Projects and Further Explorations

The closing section collects project ideas and extensions that let you deepen your intuition by tinkering. Each project ties directly back to the themes of the workbook.

Project Ideas

Possible explorations include:

- reimplementing the minimal self-attention block from this workbook using NumPy only, then comparing performance,
- instrumenting a small Transformer model in PyTorch to log tensor shapes and peak memory usage during a forward pass,
- experimenting with reduced-precision tensors (`float16`, `bfloat16`) and observing how they affect speed and numerical accuracy,
- drawing your own extension of the concept map in Figure 1 to include datasets, optimisation algorithms, or deployment scenarios.

Hands-on sessions like these turn the abstract slogan "everything is a tensor operation" into lived experience—an essential perspective for understanding why tensors truly are the hidden champions of the current AI revolution.

Suggested Project Scaffolds

To make these projects concrete, here are three possible paths:

- **NumPy-only attention.** Start from the PyTorch attention code and replace each operation with its NumPy counterpart. Add extensive `print` statements for shapes and a simple timing harness using `time.perf_counter`. Reflect on where most of the time is spent.
- **Shape tracer for a tiny Transformer.** Write a small wrapper around a `torch.nn.TransformerEncoderLayer` that hooks into the forward pass and logs the shape, device, and dtype of every intermediate tensor. Use this to draw your own version of Figure 2 for the full block.
- **Precision experiments.** Implement a minimal MLP or attention layer that can switch between `float32`, `float16`, and `bfloat16`. Measure runtime and observe how outputs differ numerically; plot the differences as histograms using Matplotlib.

You do not need a large GPU cluster for any of these experiments; even a laptop CPU or a single consumer GPU is enough to see the patterns.

As you continue reading about GPTs, LLMs, and specialised AI hardware, you can now translate most high-level claims back into tensor language: shapes, ranks, contractions, and kernels. That translation skill is one of the most valuable outcomes this workbook aims to provide.