

Embeddings: giving each token a vector of meaning

A token ID tells the model which entry you meant. It does not tell the model what to do with that entry. The next step is surprisingly small: keep a table of adjustable numbers and use the ID to select a row. That row is the token's embedding. The interesting part is not the lookup. It is how training changes the table, and what the resulting geometry does and does not mean.

Why this matters

In the previous chapter, you turned text into token IDs. Stop there and you have a useful encoding, but not a useful representation for reasoning about language. The numerical distance between two IDs is arbitrary. If the tokenizer assigns neighboring IDs to unrelated pieces of text, those pieces do not suddenly become similar. A model needs adjustable representations rather than an accidental ordering imposed by its dictionary.

Embeddings solve that problem by giving the model parameters associated with each token. Downstream calculations can multiply, add and compare these parameters. Training can change them when doing so improves an objective. This is the bridge from discrete symbols to a differentiable model: the IDs stay discrete while the numbers selected by those IDs can be adjusted continuously.

You will meet this bridge while reading model code, debugging tensor shapes, sizing a local model, or building retrieval for an agent. The word embedding appears in all those settings, but it does not always name the same object. A token lookup inside a language model is not automatically a sentence representation suitable for document search. Being able to distinguish them prevents a plausible-looking demo from becoming an unreliable application.

There is also a useful defensive engineering lesson. A high similarity score is not proof that a retrieved passage is true, authorized, or safe to treat as instructions. Similarity is an ordering rule produced by a representation and a metric. An agent still needs source checks and permission boundaries outside that ordering rule. We will not build a security system here. We will build enough understanding to avoid confusing a numerical resemblance with a trust decision.

This chapter extends the Academy's random-table demonstration with a hand calculation, a gradient trace, an actual supervised training experiment, and tests that can catch a wrong explanation. The training data is deliberately synthetic. That limitation is part of the lesson, not something to hide behind a polished cosine score.

The idea in plain words

Imagine a labeled drawer for every token. Inside each drawer is a short list of numbers. Asking for a token retrieves the numbers from its drawer. No multiplication of the token ID is involved, and no dictionary definition is stored there in plain English.

The lab uses the vocabulary `cat`, `dog`, `run`, `walk`, `the`, and `a`. Each entry has a row of four floating-point numbers. With the lab's seeded random initialization, the row for `cat`, rounded for display, is `[1.927, 1.487, 0.901, -2.106]`. These numbers are not a finding about cats. They are a random starting point.

The phrase `the cat run` becomes IDs `[4, 0, 2]`, then an array with shape `(3, 4)`: a row for each token occurrence, with four coordinates per row.

That short list is a vector. The full collection of rows is a matrix. The lookup table is called an embedding layer because it places discrete token identities into a space where numerical operations are possible. In PyTorch, `nn.Embedding` is exactly this kind of learnable lookup table, as its official documentation makes explicit.

To make the vectors useful, we need a task. In the lab's small supervised experiment, the task is to reproduce supplied context-frequency patterns. The supplied pattern for `cat` resembles the one for `dog`; the patterns for `run` and `walk` resemble each other instead. Training moves the corresponding rows toward those patterns. We can then measure which rows point in similar directions.

The careful version of "a vector of meaning" is therefore: a vector of learned numbers that is useful for a particular objective. It may encode useful linguistic regularities. It is not a universal meaning barcode, and a randomly initialized vector has not learned any linguistic regularities at all.

How it actually works

Part 1: select a row, preserve the surrounding shape

Call the embedding table E . Let V be the vocabulary size and D the embedding width. E has shape V by D . A token ID i is an integer used as a row address. Its embedding is $E[i]$, a vector containing D numbers. The table has V times D adjustable entries, irrespective of how many tokens happen to appear in the current input.

Our table has shape `(6, 4)` and contains 24 parameters. Looking up a sequence of shape `(3,)` returns `(3, 4)`. Looking up a batch of shape `(2, 3)` returns `(2, 3, 4)`. The batch dimension counts examples, the sequence dimension counts token positions, and the last dimension contains the coordinates attached to each occurrence. PyTorch's shape rule is to append the embedding width to the input shape.

Repeated occurrences share the same underlying row. If `cat` appears again, the lookup returns the same values again, provided the table has not changed. This is parameter sharing, not copying independent parameters into every position. During training, feedback from all those occurrences can flow back into the shared row.

There is another way to express the same operation. Make a vector with a single one at the token's index and zeros elsewhere. This is a one-hot vector. Multiplying that vector by E selects the corresponding row, because every other row is multiplied by zero. For a sequence, stack the one-hot vectors and perform a matrix multiplication. The lab checks that this equals direct lookup exactly.

That equivalence explains the math; it is not a reason to allocate a huge one-hot array in production. Direct indexing gets the same result without explicitly storing all those zeros. The ID is an address, the one-hot vector is an optional mathematical description of that address, and the embedding is the floating-point row selected by it.

Part 2: compare directions without confusing them with lengths

A dot product multiplies matching coordinates and adds the products. For vectors u and v , written as lists of coordinates, the dot product is the sum of $u[j]$ times $v[j]$ over coordinate index j . A norm is a vector's length. The Euclidean norm is the square root of the sum of its squared coordinates.

Cosine similarity divides the dot product by the product of the lengths:

```
cosine(u, v) = dot(u, v) / (norm(u) * norm(v))
```

Here `dot` combines matching coordinates, `norm` measures length, and the denominator removes the effect of positive rescaling. For nonzero vectors, the result measures directional alignment. It is the cosine of the angle, not the angle itself. Same direction gives one, perpendicular directions give zero, and opposite directions give minus one. These are geometric statements, not automatic claims about synonyms or antonyms.

Work through the lab's vectors `u = [3, 4]` and `v = [4, 3]`. Multiply each matching pair and add to obtain a dot product of 24. Each vector has length 5. Divide 24 by the product of those lengths to obtain 0.960. The lab performs this arithmetic directly and checks it against `torch.nn.functional.cosine_similarity`.

Now compare `u` with a positively scaled copy of itself. The lab multiplies `u` by 2 and gets cosine 1.000, although the dot product is now 50.0. The direction stayed the same while the length changed. This is why a dot-product ranking and a cosine ranking can disagree if the candidate vectors have different lengths. Normalizing both the query and every candidate to unit length makes their dot products equal their cosines.

There is an important exception: the zero vector has no direction. The mathematical expression divides by zero. Our helper rejects such a row. PyTorch's cosine function instead uses a small epsilon in the denominator for numerical stability, and returns 0.0 for the lab's zero-vector comparison. That software convention does not give the zero vector a geometric angle. If zeros represent padding or missing data, handle that meaning deliberately rather than accepting a score without inspection.

The same caution applies to a token compared with itself. Any ordinary nonzero vector has self-cosine one, including an untrained random vector. Self-similarity is a useful arithmetic check, but it cannot establish that a model learned language. Our lab proves that the self-similarity check passes before any training has happened.

Part 3: discover where learning enters the lookup

A table looks static, so how can it learn? The floating-point entries can participate in calculations whose result is a loss, a number measuring how badly a task was performed. Differentiation tells us how changing each entry would change that loss. That sensitivity is called its gradient. A training update changes parameters in a direction intended to reduce the loss.

The lookup index is not what we differentiate. We differentiate the selected numbers. If an output depends on a particular row, its gradient contribution goes back to that row. Contributions from repeated occurrences add together because they refer to the same parameter.

The lab isolates this mechanism without a language model. It looks up IDs `[0, 0, 1]` and sums every selected entry. Every selected coordinate contributes directly to that sum. The gradient on every `cat` coordinate is 2.0 because that row appeared twice. Every `dog` coordinate has gradient 1.0. The other rows have zero gradient from this calculation because none of their entries contributed to the sum.

This sum is a teaching probe, not a useful language objective. It makes the routing of gradients obvious. Do not generalize the zero-gradient observation to all optimizers and architectures. Weight decay can affect parameters separately from this lookup, accumulated optimizer state can matter, and a tied

output layer can supply another gradient path. What we have proved is narrower: the plain lookup's backward pass accumulates contributions into selected rows.

Part 4: make a small table actually learn

To expose the entire learning mechanism, we supervise the embedding directly with a tiny hand-authored context fixture. Its columns are labeled `purrs`, `barks`, `quickly`, and `slowly`. These are illustrative context labels, not a measured English corpus. The deliberately artificial counts make the relationship between target data and final geometry visible.

The row for `cat` contains counts `[8, 2, 0, 0]`. Dividing by the row total produces the target `[0.8, 0.2, 0.0, 0.0]`. The row for `dog` produces `[0.7, 0.3, 0.0, 0.0]`. The motion words put their mass in the other coordinates, while `the` and `a` deliberately receive identical targets. The fixture is not claiming that cats literally bark with a measured frequency. We supplied a convenient, artificial pattern to study optimization.

Let T be the target matrix. Our loss L is half the sum of $(E[i,j] - T[i,j])$ squared over every token index i and coordinate index j . Subtraction measures the error, squaring makes positive and negative errors contribute positively, summing combines all entries, and the factor one half makes the derivative simple. For this particular loss, the gradient of an entry is just its current value minus its target.

The update rule is "new entry equals old entry minus learning rate times gradient." The learning rate controls the size of the step. Our rate is 0.2, and the loop performs 80 updates. This is plain gradient descent written out directly, not an unexplained optimizer call.

The lab prints a worked coordinate. Its starting weight is 1.926915, the target is 0.800000, the gradient is 1.126915, and the next value is 1.701532. These displayed values are rounded from the executed calculation. Read the update literally: the weight is too large, its gradient is positive, and subtracting a positive correction moves it toward the target.

After training, the maximum absolute target error is 0.000000119. The cosine between `cat` and `dog` is 0.987241, while `cat` and `run` print 0.000000 at the displayed precision. The rows for `the` and `a` have cosine 1.000000 because their supplied targets match. The nearest-other-row lookup now pairs the motion words together and pairs the articles together.

Do not conclude that we discovered language from scratch. We fitted explicit targets that we designed to have those relationships. There is no held-out language evaluation. Real language models learn embeddings through a prediction objective coupled to other layers; they are not usually given these named target coordinates. Mikolov and colleagues' work on learned word representations is a historical entry point to that broader idea. Our narrow experiment proves that a loss can train lookup parameters and thereby change measurable geometry.

Part 5: separate geometry from arbitrary labels

If IDs are only addresses, changing every address consistently should preserve the result. The lab permutes the rows and applies the corresponding inverse mapping to the IDs. Every token still retrieves its original vector. Changing just the IDs without changing the table would instead select the wrong rows. This is why a tokenizer and its model weights form a contract.

Coordinates also need not have fixed human names. The lab rotates every learned vector with the same orthogonal matrix and verifies that all pairwise cosines remain unchanged. An orthogonal transformation

preserves lengths and dot products, even though individual coordinate values change. This is a geometric demonstration, not a claim that you can rotate one layer of a complete model without adapting the rest of the model.

In our supervised fixture the coordinates have names because we imposed those names in T. Ordinary learned latent coordinates are not generally a tidy list such as “animalness, speed, politeness.” Useful properties may be distributed across coordinates. Avoid reading a single negative number as negative sentiment or a single large coordinate as proof of a specific concept.

Build it

Run from the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/02-embeddings/lab.py
```

The lab adds the shared `labs/` directory to its Python import path using its own file location. It therefore also works when invoked by absolute path from another directory. It uses the CPU, needs no API keys or network access, and does not download a model. The reusable helpers live in `labs/tutorlab/embeddings_demo.py`; their tests live in `labs/tests/test_embeddings_demo.py`.

Start with Part 1 rather than jumping to the learned scores. `torch.manual_seed` makes the random initialization reproducible in the tested environment. `nn.Embedding` creates the table. The `STOI` dictionary maps our visible token strings to integer row addresses. Passing `dtype=torch.long` constructs integer indices, rather than floating values that might mistakenly be interpreted as data coordinates.

```
ids = torch.tensor([STOI[t] for t in ['the', 'cat', 'run']], dtype=torch.long)
vectors = table(ids)
one_hot = F.one_hot(ids, num_classes=len(VOCAB)).to(table.weight.dtype)
```

The final conversion changes the one-hot tensor to the table's floating-point type before multiplication. It does not turn the token IDs into meanings. Compare the direct result with `one_hot @ table.weight`; the equality check verifies the row-selection explanation.

Our small string wrapper rejects an unknown token rather than inventing a vector. This is not a replacement for the previous chapter's byte-level tokenizer. That tokenizer can split unfamiliar text into known pieces. Our fixture has a deliberately closed vocabulary so that row addresses remain easy to inspect.

In Part 2, `cosine_matrix` divides each row by its own length and multiplies the resulting matrix by its transpose. The result contains all pairwise cosines. The helper checks shape, floating type, finite values, and nonzero row lengths first. Those checks prevent an invalid input from quietly looking like a meaningful ranking.

In Part 3, `.backward()` populates the table's `.grad` tensor. In Part 4, the training loop first clears old gradients because PyTorch accumulates them by default. It then computes the loss, differentiates, and changes the weights inside `torch.no_grad()`. That context says not to build another differentiation graph for the parameter-update operation itself. `detach().clone()` preserves the original numbers for a genuine before-and-after comparison rather than retaining a view of changing weights.

```

loss = 0.5 * (table(ids) - targets).square().sum()
table.zero_grad(set_to_none=True)
loss.backward()
with torch.no_grad():
    table.weight -= lr * table.weight.grad

```

The full loop evaluates once more after the last update so the final reported loss belongs to the final table. Its random-number context also restores the caller's random state afterward. That small library courtesy prevents this teaching experiment from unexpectedly changing later seeded experiments.

These excerpts are copied from the executed `build/lab-output.txt`:

```

ids [4, 0, 2] → shape (3, 4)
batch shape (2, 3, 4); table parameters 24
lookup equals one_hot @ weight: True
self cosine is one even before training: True
step 0 half-squared-error sum=14.9425401688
step 1 half-squared-error sum=9.5632257462
step 10 half-squared-error sum=0.1722757369
step 40 half-squared-error sum=0.0000002640
step 80 half-squared-error sum=0.0000000000
max target error=0.000000119
trained cosine(cat, dog)=0.987241
trained cosine(cat, run)=0.000000
trained cosine(the, a)=1.000000

```

The displayed final loss rounds to zero; the separate maximum-error line shows why that is not proof of exact mathematical equality. The code uses tolerances for floating-point checks. It does not replace a failed scientific claim with prettier formatting.

At the end, `check()` verifies lookup equivalence, shapes, the hand cosine, repeated-index gradients, training convergence, expected neighbors for this fixture, consistent ID relabeling, and rotation invariance. Only then does it print `CHECK PASSED`. The shared tests also exercise invalid inputs and confirm that a single update matches the analytic update rule.

Try an intervention after predicting its effect. Set the training step count to zero: self-cosine should still pass while the convergence checks fail. Replace the dog's target counts with the run counts: the learned neighborhood should reflect your changed supervision, and assertions tied to the original fixture may fail intentionally. Repeat the cat ID again in the gradient probe: predict which row changes its gradient and which rows do not. Preserve the original gated file when experimenting, so your exercise does not overwrite the verified reference.

Common mistakes and misconceptions

"An embedding layer has already learned meaning when I construct it." The default table begins randomly initialized. The lab's untrained `cat` and `dog` cosine is 0.393828, but that is not a calibrated semantic statement. Even a plausible nearest neighbor can occur by chance. Training evidence must refer to an objective and to observations beyond a self-similarity check.

"Cosine is a probability that two texts mean the same thing." It is a normalized dot product. It can be negative, and its relationship to task relevance depends on the model and training data. A threshold needs evaluation on the actual retrieval task. Our `cat` and `dog` score does not tell you how often a document search would be correct.

"Opposite meanings must point in opposite directions." Geometry does not enforce that. Opposing words can occur in similar grammatical and topical contexts. What a learned representation considers close depends on what training rewarded, not on a rule that dictionary antonyms must have negative cosine.

"One dimension has one permanent English meaning." Our named coordinates are an intentionally supervised special case. The rotation experiment preserves the similarities while changing coordinates. A nice visual axis label is an explanatory aid, not evidence that a production model learned exactly that axis.

"Every occurrence gets its own independent parameters." Occurrences of the same token share a table row. Contextual representations arise later when the model combines the lookup with position information and surrounding text. The static lookup does not by itself know which sentence supplied the token.

"Nearest means trustworthy." A search returns the highest-scoring available candidate even if every candidate is unsuitable. It can return the query itself unless you exclude that row. Our helper explicitly masks self-matches. Production retrieval also needs relevance evaluation and source handling; a numerical ranking cannot decide authority.

In the real world

The following implementation references were checked as of 2026-10-10. They describe the linked code and documentation, not undisclosed internals of a current proprietary model.

PyTorch documents `nn.Embedding` as a learnable lookup table and specifies that the embedding width is appended to the input shape. Its `padding_idx` option suppresses the lookup gradient for a designated padding row. Its `from_pretrained` convenience constructor defaults to frozen weights. If you expect a supplied table to learn, inspect whether gradients are enabled rather than assuming the word pretrained implies fine-tuning.

Karpathy's nanoGPT defines token embeddings with `nn.Embedding(config.vocab_size, config.n_embd)` and position embeddings with a separate table. The local `labs/tutorlab/gpt.py` uses the same overall arrangement. Its forward pass adds token and position embeddings, then passes the result through transformer blocks. That is the concrete boundary between this chapter's static lookup and the later context-dependent computation.

The local GPT implementation also ties its output projection weights to its token embedding weights. The same parameter table can then participate both in input lookup and in output scoring. This is why the gradient caveat above matters: "the token was absent from input" does not by itself imply that its shared row receives no training signal anywhere in the network.

Table size is real memory. The lab calculates a hypothetical vocabulary of 50,000 entries and width 768, giving 38,400,000 parameters. At float32, those weights alone require 153,600,000 bytes, or 153.6 decimal MB. This is an illustrative configuration, not a claimed parameter count for a named deployed

model. Training also needs memory for gradients, optimizer state when applicable, activations, and other layers. Lookup output size depends on batch and sequence length; table size does not.

For document retrieval, Sentence Transformers documents encoding complete texts and comparing their resulting embeddings. Those representations come from a text-encoding model, not from looking up an arbitrary single token row. Using the same vector width does not make different models' coordinate systems interchangeable. Keep the query and document encoding procedure compatible, and evaluate on actual relevance judgments rather than borrowing a cosine threshold from an unrelated system.

Go deeper

The lecture moments in `lectures.json` were verified using the supplied lookup tool on 2026-10-10. YouTube blocked transcript access from this host, so verification used creator-published chapter markers. The anchor phrases below are marker titles, not claimed verbatim spoken quotations.

- Andrej Karpathy, [Building makemore Part 2: MLP, 65:27](#), marker "visualizing the character embeddings." Compare learned character geometry with our explicitly named context targets. Ask which properties came from the training task rather than from an axis label supplied by the teacher.
- 3Blue1Brown, [Transformers, the tech behind LLMs, 12:27](#), marker "Word embeddings." Use the spatial visualization to build intuition for vectors and directions. Remember that a drawn low-dimensional picture is not the full high-dimensional table.
- Continue at [18:25](#), marker "Embeddings beyond words," to connect a token's initial vector with representations that depend on surrounding text. Attention is a later chapter; you only need to recognize what the lookup alone cannot do.

Read the PyTorch Embedding page alongside Part 1, then its cosine function documentation alongside Part 2. For historical context, Mikolov and colleagues' paper studies representations learned from large datasets and evaluated for syntactic and semantic similarity. Contrast that evidence with our intentionally limited target-fitting exercise. The gap is useful: it separates proving a mechanism from proving broad language ability.

Explain it back

Explain to someone who understands dictionaries but not neural networks how a token ID becomes a useful vector. Use the lab's row lookup and repeated-token gradient as your concrete example. Finish by explaining why self-cosine one does not establish learning and why our successful fixture fit does not establish language understanding. Dictation is fine. Your study partner will check one gap at a time, then ask you to apply the explanation to a changed target pattern before recording mastery.

Check yourself

The questions and answer rubrics are in `quiz.json`. Answers stay there so you can predict before checking. Reading this chapter or receiving its video is not mastery evidence.

1. What does an embedding lookup actually do with a token ID?
2. What shape does a batched lookup return, and why?
3. What does cosine preserve when a vector is positively rescaled?

4. Why is self-similarity not proof of learning?
5. Which relationship does this synthetic training experiment establish, and which claim does it not establish?
6. Predict the repeated-ID gradient in the sum probe.
7. Predict what remains valid if the training loop makes no updates.
8. Explain how token IDs can be relabeled without changing the vectors retrieved.
9. Explain the difference between a static token embedding and a document representation used for retrieval.
10. Apply the learning mechanism to a changed context target rather than memorizing the current nearest neighbors.

Sources

- Academy baseline, checked 2026-10-10 for coverage of lookup, shapes, cosine and untrained geometry: <https://krypteiasec.com/academy/llm-fundamentals/ch2>
- PyTorch, Embedding API, checked 2026-10-10 for initialization, shape, padding and frozen pretrained defaults: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.Embedding.html>
- PyTorch, cosine similarity API and epsilon convention, checked 2026-10-10: https://docs.pytorch.org/docs/2.11/generated/torch.nn.functional.cosine_similarity.html
- PyTorch, one-hot representation API, checked 2026-10-10: https://docs.pytorch.org/docs/stable/generated/torch.nn.functional.one_hot.html
- Karpathy, nanoGPT model source, checked 2026-10-10: <https://github.com/karpathy/nanoGPT/blob/master/model.py>
- Karpathy, Building makemore Part 2: MLP: https://www.youtube.com/watch?v=TCH_1BHY58I
- 3Blue1Brown, Transformers, the tech behind LLMs: <https://www.youtube.com/watch?v=wjZofJX0v4M>
- Mikolov, Chen, Corrado and Dean, Efficient Estimation of Word Representations in Vector Space: <https://arxiv.org/abs/1301.3781>
- Sentence Transformers, Semantic Textual Similarity, checked 2026-10-10: https://www.sbert.net/docs/sentence_transformer/usage/semantic_textual_similarity.html
- Local reproducible evidence: `lab.py`, `build/lab-output.txt`, `labs/tutorlab/embeddings_demo.py`, and `labs/tests/test_embeddings_demo.py`. The printed training values and all hypothetical memory calculations above come from this lab, not an external benchmark.
- Local model implementation referenced here: `labs/tutorlab/gpt.py`, token lookup and weight tying in the GPT constructor, position addition and blocks in its forward method. No existing shared implementation was modified for this chapter.