

1.2 — Text Becomes Numbers

Gavin · AI-assisted course design

Contents

1	Prerequisites and learning goals	3
2	1. Establish the interfaces	3
3	2. Why tokens are not words	4
	3.1 Bytes, characters, and visible marks	4
4	3. A conversation needs a representation too	5
5	4. Token IDs select embedding rows	6
	5.1 An original six-token example	6
	5.2 Coordinates require a reference frame	7
6	5. Context changes representations	7
7	6. Logits put scores on the vocabulary	7
8	7. Softmax turns relative scores into probabilities	8
9	8. Selection is a separate decision	9
10	9. Continue the sequence and repeat	9
11	Check your understanding	12
	11.1 Discussion	12
12	More Learning	12
13	References	13

Type “Paint it” into a text-generation system. Before another word appears, software must represent those characters as tokens, identify the tokens by number, retrieve numerical vectors, run a trained model, and choose a continuation. Each operation has a different job. Confusing them makes it difficult to explain even a small change in output.

This lesson follows that path through a conventional autoregressive text language model. We will specify what enters and leaves its central neural computation, while leaving attention and other Transformer internals for later lessons. The aim is to make the boundary between text processing, learned computation, and generation decisions precise.

1 Prerequisites and learning goals

Start with Lesson 1.1 — The AI Zoo, particularly its distinctions among architecture, trained weights, inference runtime, and orchestration. No programming or calculus is required here.

By the end, you should be able to:

- Distinguish a vocabulary, tokenizer, token ID, and embedding vector.
- Trace text through embedding lookup to contextual representations, logits, probabilities, and token selection.
- Explain why whitespace, Unicode, and conversation formatting affect model input.
- Compute a small lookup and softmax example.
- Separate next-token probability from factual confidence and repeated generation from training.

Math to know / refresh

Needed now: A vector is an ordered list of numbers. A matrix is a rectangular array. Indexing selects an entry or row. Exponentiation produces positive numbers, and normalization divides positive weights by their total to obtain probabilities.

Useful refresh: Zero-based indexing, matrix dimensions, and conditional probability. We write $P(a \mid b)$ for the probability of a given b . The vertical bar means “given,” not division.

Side trail: Logarithms undo exponentiation: $e^{\ln 3} = 3$. This identity makes our worked example easy to check. Dot products, gradients, and information theory become useful later; they are not prerequisites for following the main argument.

2 1. Establish the interfaces

A **vocabulary** is a finite inventory of token types, with an integer ID assigned to each. A **tokenizer** supplies the rules and data used to turn text into a sequence drawn from that inventory, and usually a way to decode IDs back into text. A **trained language model** uses numerical representations of the sequence to calculate scores for possible continuations.

The tokenizer’s vocabulary may itself have been learned from a corpus. That training is distinct from training the neural model’s weights. During ordinary use, a fixed tokenizer configuration applies its established procedure; it does not ask the language model which next word would make sense. Senten-

cePiece is one concrete toolkit that makes tokenizer training and tokenization separate operations. Kudo and Richardson, 2018

The ID-to-token mapping is part of the interface. If ID 500 refers to one token in tokenizer A and another in tokenizer B, sending A's IDs to a model trained with B changes the input's interpretation. Matching vocabulary sizes would not fix that mismatch.

Think of a laboratory sample with a numbered label. The label lets you retrieve the right sample; the number itself does not describe its chemistry. Similarly, adjacent token IDs need not have related meanings. ID 501 is not "slightly more" of whatever ID 500 represents.

Retrieval check: Which component could you inspect from tokenizer files alone: the tokenizer, the input embedding matrix, or the model's next-token logits?

3 2. Why tokens are not words

A word-level vocabulary faces a practical problem: new names, compounds, spellings, and code identifiers keep appearing. A character-level system can represent a wider variety of strings with a smaller inventory, but often needs longer sequences. Subword tokenization offers a compromise, representing frequent pieces together while permitting less common strings to use several pieces. Sennrich and colleagues demonstrated this approach for neural machine translation using byte-pair encoding, or BPE. Sennrich, Haddow, and Birch, 2016

For intuition, imagine a tokenizer whose learned inventory includes `paint`, `ing`, and `re`. It could represent repainting using several pieces. This is an invented illustration, not the output of a particular tokenizer. Actual boundaries follow the tokenizer's procedure and vocabulary; they need not match syllables, grammatical components, or a person's preferred spelling explanation.

In a basic BPE construction, frequently occurring adjacent units are repeatedly merged to form larger units. Encoding subsequently uses the learned merge information. Implementations also make choices about preprocessing, which boundaries merges may cross, and special-token handling. Knowing that two tokenizers use BPE is therefore insufficient to predict identical results.

3.1 Bytes, characters, and visible marks

GPT-2 provides a useful concrete example. Its tokenizer uses a byte-level BPE scheme built on the 256 possible byte values, with additional merged tokens and rules restricting some merges. This lets it represent text through UTF-8 bytes without needing a vocabulary entry for every possible word. The original report explains this representation in Section 2.2. Radford and colleagues, 2019

These units should stay separate in your thinking:

- A displayed symbol is what you see.
- Unicode code points specify text elements; a displayed symbol may combine several.
- UTF-8 encodes code points into bytes.
- The tokenizer groups its underlying units into tokens.

Consequently, one visible symbol need not be one token. Nor does a token need to decode into a complete, independently displayable Unicode character. Token-inspection tools may show readable

stand-ins for bytes. Those stand-ins help expose the representation, but should not be mistaken for literal extra characters inserted into the user’s sentence.

Whitespace matters too. GPT-2’s documented tokenizer treats spaces as parts of tokens, so the same written word can receive a different encoding with a preceding space. Preserve spaces and line breaks when investigating an example. [Hugging Face GPT-2 documentation](#)

The practical consequence is straightforward: count tokens using the actual tokenizer and settings. Character counts and word counts are proxies whose accuracy depends on the text. Being able to encode a script or language also establishes nothing by itself about the model’s competence in that language.

💡 Lab recommended here

Use Lab 02 — Inspect Tokenization to examine an actual GPT-2 tokenizer on your CPU. Predict boundaries for prose, code, numbers, whitespace, and Unicode, then inspect the resulting IDs and decoded text. The Lab requires tokenizer files, not model weights or a paid inference API. Return here when the distinction between the original text, displayed token pieces, and token IDs feels concrete.

4 3. A conversation needs a representation too

In a chat application, the visible user message is only one part of the input. A **chat template** serializes role-labeled messages into the model’s expected sequence. It may insert speaker markers, separators, end-of-message tokens, and an assistant-response prefix. Different models can expect different formats even when they share an underlying model family.

Special tokens are entries given particular roles by the tokenizer, model training, or runtime. Examples include beginning-of-sequence, end-of-sequence, and padding markers. Their spellings and behavior are model-specific. A role label may also be ordinary text beside a special delimiter; every visible label is not necessarily one special token. Use the supplied template and check its output, including whether a later tokenization step would add duplicate markers. [Hugging Face chat-template guide](#)

Consider an invented serialization: a user-start marker, the message `Paint it`, a message-end marker, and an assistant-start marker. The model continues after that final marker. Changing the markers changes its input even though the interface still displays the same user sentence. The template is part of the system’s behavior.

A **context window** limits the sequence a particular model/runtime configuration can process. Conversation text, inserted instructions, retrieved passages, and formatting consume positions; generated tokens also extend the sequence in the usual growing-prefix setup. For a hypothetical 100-position budget, 70 input tokens leave at most 30 positions for extension under that simple accounting rule. Actual services may impose additional limits or manage context differently.

An interface can show an old message that is absent from the current model input. To establish what the model received, inspect the assembled sequence or request trace. A tokenizer's ability to encode a long string does not establish that the neural model can process the resulting length.

5.4. Token IDs select embedding rows

The input embedding matrix holds one learned vector for each vocabulary entry. Let V be vocabulary size and d embedding dimension:

$$E \in \mathbb{R}^{V \times d}.$$

For token ID i , lookup retrieves row $E_{i,:}$. Given n token IDs, the lookup produces an $n \times d$ array. The sequence length, vocabulary size, and vector dimension answer three different questions: how many positions, how many possible token types, and how many coordinates per vector. PyTorch's embedding layer exposes precisely this lookup-table structure. [PyTorch Embedding reference](#)

5.1 An original six-token example

We will use a deliberately tiny vocabulary throughout the numerical example. Quotes show token spellings; the initial space inside several spellings is significant.

ID	Token spelling	Embedding row
0	<end>	(0, 0, 0)
1	Paint	(1, 0, 1)
2	" it"	(0, 2, -1)
3	" red"	(-1, 1, 0)
4	" blue"	(1, 1, 0)
5	.	(0, 0, 1)

This is an instructional construction, not a real model's vocabulary or measured output. For the strings used here, define the toy tokenizer to match these pieces exactly. Its end marker is reserved for stopping; we do not insert it into this example's initial input.

Paint it becomes IDs $[1, 2]$. Its lookup is:

$$X = E[[1, 2], :] = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & -1 \end{bmatrix}.$$

Here $V = 6$, $d = 3$, and $n = 2$. We selected two rows, preserving order. We did not multiply the second row by the numeric value 2. Changing the prefix to Paint it red adds ID 3 and therefore adds $(-1, 1, 0)$ as another row.

Prediction check: If the input IDs were $[2, 1, 2]$, what would the lookup array contain? Would the two occurrences of ID 2 retrieve different vectors?

The same ID retrieves the same token-embedding row in this fixed table. Later computation can produce different representations for its different occurrences. That distinction will matter throughout interpretability work.

5.2 Coordinates require a reference frame

An embedding is useful because the rest of the trained system knows how to use its coordinates. The first coordinate is not automatically “redness,” “importance,” or any other human-readable property.

Even a simple coordinate swap demonstrates the danger. In our toy table, exchanging columns one and two would turn the Paint vector into $(0, 1, 1)$. A downstream calculation that consistently exchanges its corresponding input weights could produce the same result. The coordinate labels alone cannot settle meaning.

Therefore, record the model, checkpoint, and representation location when comparing vectors. Two models with equal vector dimensions do not thereby share a coordinate system. A token embedding, a contextual hidden state, and a document-search embedding also come from different operations; the shared word “embedding” does not make them interchangeable.

6 5. Context changes representations

Lookup has supplied a starting vector at each position. The model then combines information through its learned computation, using a representation of order. In a causal text model, the prediction at a position depends on the prefix available there, rather than on future tokens. The Transformer paper describes how its decoder preserves that causal structure; the detailed mechanism belongs to Module 2. Vaswani and colleagues, 2017

Compare the prefixes Paint it and Describe it. If the same tokenizer assigns the same ID to the same piece " it", its initial token embedding is identical. Its contextual representation can differ because the available context differs. “The vector for this token” is ambiguous unless we say whether we mean the stored lookup row or an activation at a particular position and layer.

For this lesson, call the final representation at the last input position h . It contains the result of the model’s contextual computation. The embedding table alone cannot tell us h : we also need the intervening architecture, weights, and input arrangement. This is an important limit on our numerical example. We will calculate lookup and output normalization exactly, while explicitly supplying the intervening computation’s output scores.

7 6. Logits put scores on the vocabulary

The output head maps the final representation to a **logit** for each candidate next token. Logits are real-valued scores before normalization. They may be negative, positive, or zero. They do not need to sum to anything meaningful. GPT-2’s documented outputs distinguish these vocabulary scores from its hidden states. Hugging Face GPT-2 reference

A common output head can be written:

$$z = hW_{\text{out}} + b.$$

With a row-vector convention, h has d entries, W_{out} has shape $d \times V$, and z has V entries. A bias b is optional. We will study the matrix multiplication itself in Lesson 1.3; here, track the change from hidden-coordinate space to vocabulary-score space.

Some architectures **tie** input and output weights, sharing the embedding matrix with the output projection, typically through a transpose in this notation. Others use separately learned parameters. Weight tying is a design choice, not a consequence of using tokens or embeddings. Press and Wolf, 2017

For our Paint it example, suppose the full model computation returns these logits, in ID order:

$$z = (0, 0, 0, \ln 3, \ln 2, \ln 2).$$

These stipulated scores are enough to calculate the next-token distribution. They are not derivable from the displayed embedding table alone. Nor are they claims about how a real trained model would continue this phrase.

8.7. Softmax turns relative scores into probabilities

Softmax exponentiates the scores and divides each result by the sum:

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^{V-1} e^{z_j}}.$$

For finite logits in exact arithmetic, the probabilities are positive and sum to one. The denominator makes each probability depend on all candidate scores. PyTorch softmax reference

In our example, exponentiation gives:

$$e^z = (1, 1, 1, 3, 2, 2), \quad \sum_j e^{z_j} = 10.$$

Therefore:

$$p = (0.10, 0.10, 0.10, 0.30, 0.20, 0.20).$$

The token " red" has probability 0.30, and " blue" has probability 0.20. The ratio is 3/2. A zero logit produced probability 0.10 here, so a zero score clearly does not mean an impossible token.

Adding the same constant to every logit leaves the distribution unchanged: the common exponential factor cancels between numerator and denominator. Implementations can exploit this by subtracting the maximum score before exponentiation. You can verify the algebra without knowing anything about language.

What does 0.30 mean? It is probability assigned to one next token under this context and distribution. It is not a 30% probability that the whole answer will be factually correct. In this example, " red" is merely one possible continuation. Evaluating truth requires a task and evidence beyond a score attached to its first token.

A **temperature** $T > 0$ modifies the distribution by applying softmax to z/T . For the toy logits, $T = 0.5$ gives unnormalized weights $(1, 1, 1, 9, 4, 4)$, totaling 20. The probability of " red" becomes $9/20 =$

0.45. Sharpening that distribution has changed selection preferences without adding evidence about the world. Temperature is one of the controls exposed by the Hugging Face generation API.

Prediction check: If all six logits were equal, what would each probability be? If only the " blue" logit increased, would the other probabilities stay fixed?

9 8. Selection is a separate decision

Greedy selection chooses a highest-scoring token. For our original distribution it selects " red", despite that token receiving only 30% of the probability mass. **Sampling** draws according to a distribution, allowing less-probable outcomes to be selected. Generation software may first modify scores or restrict eligible candidates, as with temperature, top-k, or top-p settings. The final sampling distribution can therefore differ from the model's unmodified softmax. Hugging Face generation strategies

For a concrete sampling construction, lay out the six probabilities as consecutive intervals along $[0, 1)$ in ID order. They end at 0.10, 0.20, 0.30, 0.60, 0.80, and 1.00. A uniform draw of 0.65 lands in ID 4's interval, selecting " blue". We now have two valid procedures giving different continuations from identical logits.

For fixed scores and a fixed tie-breaking rule, greedy selection has a fixed answer. This does not establish that every deployment will reproduce identical scores across hardware, software versions, or numerical settings. Reproducibility has additional engineering requirements. PyTorch reproducibility guidance

Greedy selection also does not search all possible completed answers. Imagine one first token has probability 0.6 but its best continuation has probability 0.2, while another has probability 0.4 and its best continuation has probability 0.9. The corresponding two-token paths have probabilities 0.12 and 0.36. Taking the best immediate choice did not find the highest-probability path in this invented example.

10 9. Continue the sequence and repeat

After choosing " blue", append ID 4 to the toy prefix. The sequence becomes $[1, 2, 4]$, which decodes to Paint it blue. Run the next prediction conditioned on that extended sequence. Its distribution can differ from the preceding one because the context changed.

This is **autoregressive generation**: previous outputs become inputs for later predictions. Conceptually, the loop is predict, select, append, and repeat. Efficient implementations reuse cached computation rather than recalculating every earlier operation, but the conditioning relationship remains the same. The GPT-2 interface documents such cached state separately from the model's weights. Hugging Face GPT-2 reference

Generation stops when the runtime's conditions are met, such as a designated end token or a maximum output length. Turning IDs into displayable text is called decoding too, so distinguish **tokenizer decoding** from the **generation decoding strategy** that chooses IDs.

The current sequence and cached activations can change while the model's weights stay fixed. A long reply is therefore not evidence that the model is training as it writes. Likewise, a transcript stored by a

product becomes useful to a later request only through whatever input assembly or other mechanisms that system actually provides.

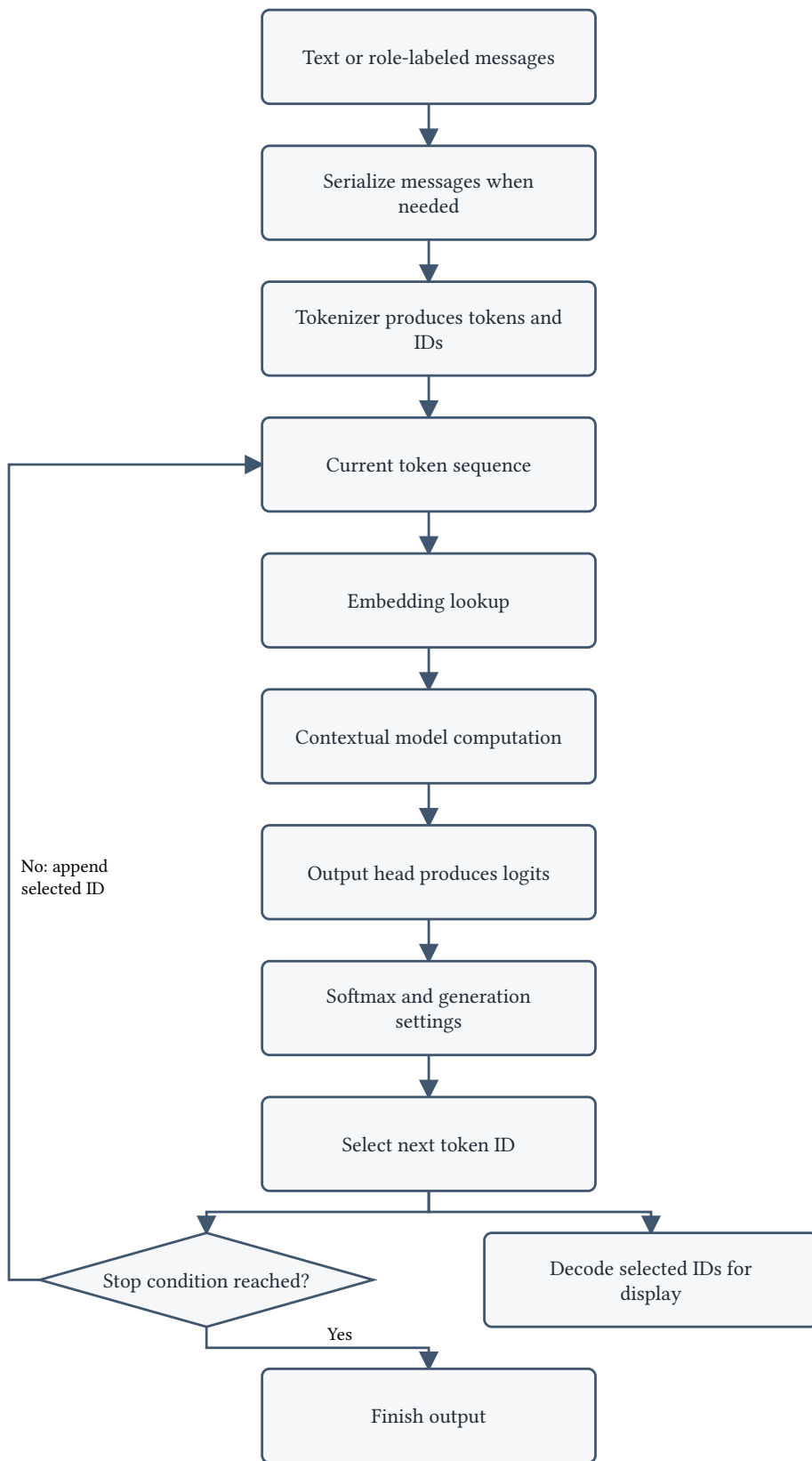


Figure 1. Original course diagram of the conceptual generation loop. The probability stage summarizes the configured selection procedure; greedy selection can choose an argmax directly without computing softmax. A runtime can stream text and reuse cached computation.

Prose alternative: Serialize messages where necessary, tokenize the resulting input, and look up vectors for its IDs. Run contextual computation and score the vocabulary. Apply the configured selection procedure. Decode selected IDs for display and, while continuation is allowed, append them to the prefix for the next prediction.

11 Check your understanding

Answer from memory before reading the discussion.

1. A tokenizer encodes an unfamiliar word. What has this established about model understanding?
2. Why does replacing the tokenizer require more care than matching vocabulary size?
3. Write the lookup rows for $[2, 1, 2]$ in the toy system.
4. Which missing component prevents computing the example logits from its embedding table?
5. Can greedy selection choose a token whose probability is below 50%?
6. Does choosing " blue" mean the next step uses the same distribution again?
7. Where would you look if a chat interface displayed an earlier instruction that the model appeared to ignore?

11.1 Discussion

1. Encoding establishes representability under that tokenizer, not comprehension or reliable performance.
2. The mapping from each ID to its learned input and output roles must agree.
3. The rows are $(0, 2, -1)$, $(1, 0, 1)$, and $(0, 2, -1)$, in that order.
4. The contextual computation and output head, including their weights, are missing.
5. Yes. Our highest probability is 30%.
6. No. The selected token extends the conditioning prefix; new scores must be calculated.
7. Inspect the actual assembled input and relevant formatting first. Visibility in the interface alone does not establish inclusion or explain the failure.

For the earlier checks: the tokenizer can be inspected from its own files; repeated IDs retrieve repeated rows; equal logits yield $1/6$ each. Raising only one logit lowers the other softmax probabilities because their shared denominator increases.

The central question remains: **Where does this behavior live?** Token boundaries come from the tokenizer; initial vectors and contextual transformations involve learned parameters; selection depends on inference settings; conversation inclusion depends on the surrounding system. We can now study the neural computations between those boundaries without conflating their responsibilities.

12 More Learning

- GPT-2 technical report — Read Section 2.2 for the byte-level representation decisions. Treat it as a documented model example, not a specification for every LLM.

- Neural Machine Translation of Rare Words with Subword Units — Read the motivation and BPE discussion. Ask which problems subword units address and which they leave to the model.
- PyTorch Embedding reference — Inspect input and output shapes. You can understand the lookup contract without running PyTorch.
- Chat templates — Compare two serialized conversations and identify the boundaries and response prefix.
- Generation strategies — Start with greedy selection and sampling; save beam search and implementation details for the inference lesson.
- Using the Output Embedding to Improve Language Models — Optional deeper reading on why sharing input and output parameters is an architectural choice.

13 References

The six-token system, calculations, path-probability counterexample, and diagram are original instructional constructions. No real-model outputs are claimed. Primary research and official documentation were checked on 7 October 2026; API defaults and model-specific formats should be verified against the version actually used.

- Kudo, T., and Richardson, J. (2018). SentencePiece.
- Sennrich, R., Haddow, B., and Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units.
- Radford, A., et al. (2019). Language Models are Unsupervised Multitask Learners.
- Vaswani, A., et al. (2017). Attention Is All You Need.
- Press, O., and Wolf, L. (2017). Using the Output Embedding to Improve Language Models.
- Hugging Face. GPT-2 reference, chat templates, generation strategies, and generation API.
- PyTorch. Embedding, softmax, and reproducibility.