Tokens and Word Vectors: Subwords, BPE, and word2vec

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

*Prerequisites: the **RNN** note §2 (tokens → IDs → embeddings); logistic regression and binary cross-entropy; PCA (Week 5). Optional companion: **TOKENS_ADVANCED**.*

This note covers how text is cut into pieces, by **byte-pair encoding** (§2–§4), and how vectors for the pieces are learned, by **word2vec** (§5–§8).

1 The Big Picture: Text In, Vectors Out

Key point:

Which pieces does the homework use where? The BPE tokenizer (§4) is a stand-alone experiment: you build it and compare it with Hugging Face's and GPT-2's. The word vectors (§6–§8) and the GPT of the TRANSFORMER note both still read HW6's *word* vocabulary (5,274 words), so their numbers stay comparable with HW6. And the GPT learns its own embedding table from scratch: the word2vec vectors are studied here, not loaded into it.

Figure 1 is HW6 Problem 3's input pipeline with one box replaced. The *tokenizer* decides what the pieces are; the *embedding table* decides what vector each piece gets. Whatever model comes next (an RNN, a Transformer) only ever sees those vectors.

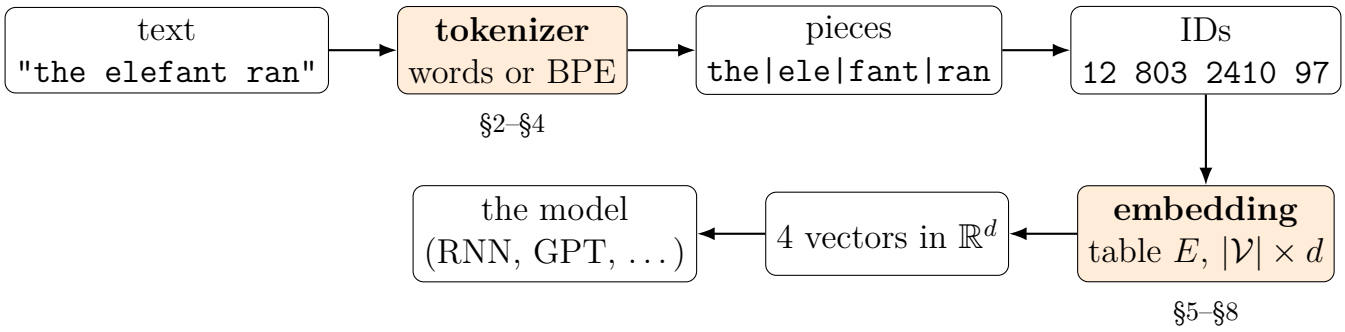


Figure 1: The input side of every language model. The two shaded boxes are this note.

Almost everything here is something you have already built (Table 1).

Part	Where you have seen it	What changes
pieces → IDs (a vocabulary)	HW6 P3: `d2l.Vocab`, `<unk>`, `<eos>`	the pieces are subwords, not words
ID → vector (`nn.Embedding`)	HW6 P3, P4	the table is now trained on its own task first
dot product and cosine as similarity	Week 2–3 k-NN; Week 5 kernels	nothing
"is this pair real?" with σ(score) and BCE	HW2 logistic regression; HW5 GAN discriminator	the score is a dot product of two *learned* vectors
2-D projection of vectors	Week 5 PCA	nothing
merging frequent pairs into new pieces	**new** (§4)	—
noise words as negative examples	**new** (§7)	—

Table 1: The parts of this note.

2 Three Ways to Cut Text: Words, Characters, Subwords

The RNN note's box explained why whole words were fine for TinyStories: the stories were written with a child's vocabulary, so 5,274 words cover 99.6% of the text. Here is what happens to the same word vocabulary on text it was not built for, next to the two alternatives.

- **Words.** Short sequences, but a fixed list: anything not on it becomes `<unk>`, and *run*, *runs*, *running* are three unrelated IDs.
- **Characters.** Nothing is ever unknown and the vocabulary is tiny, but the sequences become 4–5 times longer and the model must learn spelling before it can learn meaning.
- **Subwords.** The middle ground ("the Goldilocks zone" on the d2l slide): frequent words stay whole, rare words are split into a few frequent pieces, and every word can still be written down.

Table 2 cuts five texts three ways: by HW6's word vocabulary, by a BPE vocabulary of the *same size* (5,274 pieces) learned on the same TinyStories text, and by GPT-2's tokenizer. (The homework asks the same question about five texts of its own.)

Text	**Words (5,274)**	**BPE on TinyStories (5,274)**	**GPT-2 byte-level BPE (50,257)**
a TinyStories sentence	16 tokens, 0 unknown	the same 16 tokens	16 pieces
`The litle dinosor wanted a cookee.`	`the <unk> <unk> wanted a <unk> .`	`the lit le dino sor wanted a coo ke e .`	12 pieces, `Ġd inos or`
`Hermione and Gandalf visited Totoro.`	3 of 6 `<unk>`	`her mi one and g and al f visited to to ro .`	9 pieces, `Herm ione`, `ĠGand alf`
`Mañana vamos a la playa con mis amigos.`	7 of 10 `<unk>`	16 pieces, nothing unknown	16 pieces, nothing unknown
`x = [n * n for n in range(5)]`	4 of 7 `<unk>`; `(5)]` gone	15 pieces, 3 `<unk>` (`=`, `[`, `]`)	13 pieces; `Ġrange` whole

Table 2: Words vs subwords on five texts. `Ġ` marks a GPT-2 piece that starts with a space.

Read the table row by row and ask: *what does the model actually receive?* With words, a typo, a name, a Spanish sentence or a line of code turns into `<unk>` — the model cannot read it and cannot write it. With BPE the model still sees the spelling, and a misspelled word shares pieces with the right one (`dino sor`). The TinyStories BPE fails only on *characters* it never saw in children's stories (here `=`, `[`, `]`); GPT-2 closes that gap by working on the 256 possible bytes, so nothing is ever unknown. The price of subwords is longer sequences for unusual text (the Spanish sentence: 8 words, 16 pieces); on ordinary text it is only a few percent more tokens than words.

Remark. Splitting words into meaningful pieces is an old idea. fastText (Week 13 lecture 8/13, p. 16) represents *friend*, *friends*, *friendship* through shared character n-grams, so related words share parameters. BPE does the same thing with pieces that are *learned* from frequencies instead of all n-grams (more on fastText in TOKENS_ADVANCED).

3 Subwords in Modern Language Models

Model	Tokenizer	How pieces are chosen	Vocabulary
GPT-2 (2019), GPT-3	byte-level BPE	merge the most frequent pair	50,257
RoBERTa	byte-level BPE (GPT-2's)	same	50,265
GPT-4 / GPT-4o	byte-level BPE (`tiktoken`)	same	≈100k / ≈200k
LLaMA 1 and 2, Mistral 7B	SentencePiece BPE, byte fallback	same; a space is written _	32,000
LLaMA 3	byte-level BPE (`tiktoken`-based)	same	128,256
BERT	WordPiece	merge the pair with the best likelihood gain	30,522
T5	SentencePiece unigram	start large, prune the least useful pieces	32,100

Table 3: Tokenizers of well-known models. Sizes are the released tokenizers' vocabularies, special tokens included (read with Hugging Face `transformers`, October 2026); the two GPT-4 entries are rounded.

Key point: No modern language model reads whole words, and most of them use BPE. A fixed vocabulary of a few tens of thousands of pieces covers *any* text — any language, typo, name or program — at the cost of slightly longer sequences.

4 Byte-Pair Encoding, the Algorithm

BPE *learns* its pieces from a training text, by repeatedly gluing together the two symbols that appear next to each other most often.

Algorithm: learning BPE merges

1. Count the words of the training text; write each word as a sequence of characters.
2. Count every **adjacent pair** of symbols inside the words. Each occurrence counts as much as its word's frequency, and pairs never cross from one word to the next.
3. **Merge** the most frequent pair (a, b) into one new symbol ab, in every word, and append (a, b) to the merge list.
4. Repeat steps 2–3 a fixed number of times (HW7: 500 merges).

To encode a word (even one never seen in training): start from its characters and apply the merges *in the order they were learned*.

The ordered merge list *is* the tokenizer. The vocabulary is just the alphabet plus one new symbol per merge, but the *order* is what tells you how to split a new word: early (frequent) merges are applied first. Frequent words end up as one piece, rare words as a few frequent pieces.

Worked example: a tie-free toy corpus (the HW7 checkpoint) The corpus is `hug`×4, `hugs`×2, `pug`×2, `pun`×2, `bun`×1, `sun`×1 (12 words). Table 4 counts pairs, merges the winner, and counts again.

Step	Pair counts	Merge	Corpus after the merge
1	(u,g) 8 = 4+2+2; (h,u) 6; (p,u) 4; (u,n) 4; (g,s) 2; (b,u) 1; (s,u) 1	u+g	`h ug`, `h ug s`, `p ug`, `p u n`, `b u n`, `s u n`
2	(h,ug) 6; (u,n) 4; (ug,s) 2; (p,ug) 2; (p,u) 2; (b,u) 1; (s,u) 1	h+ug	`hug`, `hug s`, `p ug`, `p u n`, `b u n`, `s u n`
3	(u,n) 4; (hug,s) 2; (p,ug) 2; (p,u) 2; (b,u) 1; (s,u) 1	u+n	`hug`, `hug s`, `p ug`, `p un`, `b un`, `s un`

Table 4: Three BPE merges by hand. Each step has a clear winner (runner-ups 6, 4, 2), so every correct implementation, Hugging Face's included, learns exactly these merges.

Encoding unseen words with the merge list (u+g, h+ug, u+n): `sung`: start from `s u n g`; u+g and h+ug do not apply (`u` is followed by `n`); u+n gives `s un g`; `bug` → `b ug`; `pugs` → `p ug s`. Note step 1 of the count: (u,g) occurs in `hug`, `hugs` and `pug`, so its count is $4 + 2 + 2 = 8$ — the word frequencies, not the number of distinct words.

Ties, and why your merge list may part ways with the library's. When two pairs have the same count, the algorithm says nothing about which to merge; every implementation breaks ties its own way. In the reference run on TinyStories (500 merges), the hand-written BPE and Hugging Face's agree on the first 310 merges *in order* and on all 500 as a set. Merge 311 is a genuine tie — (th,is) and (ou,se) both have count 4,870 — broken differently, after which the two lists are shuffled versions of each other. So a difference is a bug only if the counts at the first difference are *not* equal; the notebook checks exactly that.

Details the book skips.

- **Word boundaries.** BPE works inside words. Some implementations mark word ends (d2l writes `_`, Sennrich et al. `</w>`), GPT-2 marks a leading space (Ġ), SentencePiece writes spaces as _ and treats them as ordinary symbols. HW7 uses no marker, like Hugging Face with a whitespace pre-tokenizer.
- **Overlapping pairs.** `a a a` contains the pair (a,a) twice in the counts, but a left-to-right merge produces `aa a`: one merge, then the scan moves past both symbols.
- **Bytes.** Byte-level BPE (GPT-2) starts from the 256 byte values instead of the characters seen in training, so no input can ever be unknown; an accented letter is two bytes and may be two pieces.

5 From IDs to Vectors

An ID is just a row number; the model needs a vector. The simplest choice, a **one-hot** vector (a 1 in position "ID", 0 elsewhere), has two problems (lecture 8/13, p. 3): it has as many dimensions as the vocabulary ("#dim = #vocab"), and every two words are equally different — the dot product of two different one-hot vectors is 0, so there is "no easy sim()".

An **embedding table** $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ fixes both. Row i is the vector of piece i, with d in the tens to thousands (HW7: $d = 100$ here, 256 in the Transformers). Looking up row i is the same as

multiplying the one-hot vector by E:

$$\text{one-hot}(i)^\top E \;=\; E_{i,:} \qquad (1 \times |\mathcal{V}|) \cdot (|\mathcal{V}| \times d) = (1 \times d),$$

which is why `nn.Embedding` is "a linear layer whose input is one-hot", implemented as an index. Similarity is now the dot product or, independent of the vectors' lengths, the **cosine** $\cos(a, b) = a^\top b / (\|a\| \, \|b\|)$. For example $(3, 3)$ has cosine 0.71 with both $(1, 0)$ and $(0, 2)$ (both are 45° away from it, whatever their lengths) and -0.63 with $(-1, 0.1)$; the dot products with the first two, 3 and 6, would wrongly call $(0, 2)$ more similar just because it is longer.

In HW6 the table started random and was trained together with the language model. The rest of this note trains it *on its own*, on a task whose only purpose is to make similar words get similar rows.

6 word2vec: a "Fake Task" That Produces Word Vectors

The idea, in the lecture's words (8/13, p. 4): 300 real numbers per word, the dot product for similarity, and a probability model whose loss is minimized only when the vectors are meaningful. The model is a tiny network (Figure 2): a one-hot input, a hidden layer with no activation, and one output per vocabulary word.

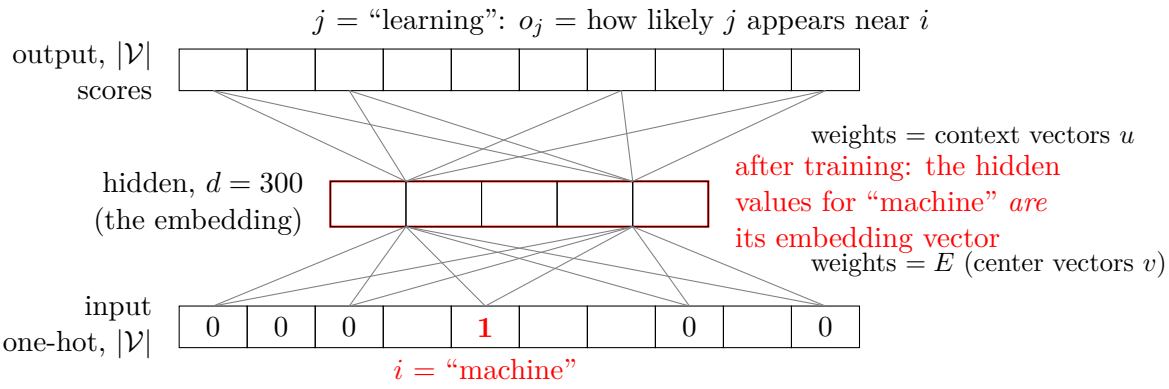


Figure 2: The word2vec network, after the board (Lecture 8/13, p. 5). The one-hot input selects one row of the input weights; that row is the word's vector.

The **skip-gram** task: given a **center** word c, predict the words around it within a window ("the black **cat** sat on" with window 2: *the, black, sat, on*). Every word w has two vectors, v_w when it is the center and u_w when it is a context word ("embed as center / embed as context"). The score of a pair is the dot product $u_o^\top v_c$, turned into a probability over the whole vocabulary by a softmax:

$$P(o \mid c) \;=\; \frac{\exp(u_o^\top v_c)}{\sum_{w \in \mathcal{V}} \exp(u_w^\top v_c)}.$$

In words: the context word o is likely near c when its vector points the same way as c's. The training loss is $-\log P(o \mid c)$ summed over every (center, context) pair in the text, treating the pairs as independent (the board's "naive assumption"). Two words that appear in similar contexts are pushed toward the same context vectors, and so end up with similar center vectors. Nobody

tells the model what a colour or an animal is; the vectors only record *which words keep the same company.*

The catch is the denominator: one sum over all $|\mathcal{V}|$ words (5,274 here, 50,000 for a real vocabulary) for *every* pair, millions of times per pass.

7 The Tricks for Computation: Negative Sampling and Subsampling

The board's fix (Lecture 8/13, p. 8): *don't compute the normalizer.* Replace "which of all words is the context?" by a yes/no question about one pair at a time — *are these two words in the same window?* — answered by a sigmoid:

$$P(D = 1 \mid c, o) \;=\; \sigma(u_o^\top v_c).$$

A yes/no question needs *no* examples too. For each real pair, draw K **noise words** $n_1, \ldots, n_K$ (HW7: $K = 5$) at random from the whole vocabulary, independently of the window, and ask the model to answer "no" for them. (Now and then a noise word happens to be a real neighbour; with thousands of words that is rare enough to ignore.) The loss of one pair is

$$\ell \;=\; -\log \sigma(u_o^\top v_c) \;-\; \sum_{k=1}^{K} \log \sigma(-u_{n_k}^\top v_c),$$

using $1 - \sigma(s) = \sigma(-s)$.

Key point: Negative sampling *is* HW2's logistic regression. Each (center, word) pair is one example; its "features" are the two vectors, its score is their dot product, the real neighbour has label 1 and each noise word label 0, and ℓ is the binary cross-entropy summed over the $1 + K$ examples. The only new thing is that the vectors themselves are the parameters being learned.

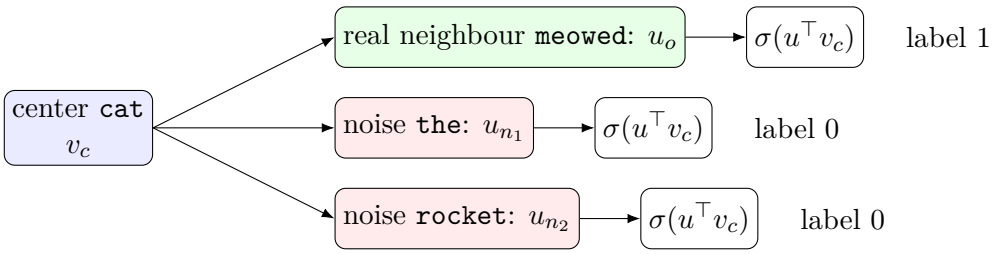


Figure 3: Skip-gram with negative sampling as logistic regression: one real pair and K noise pairs per center word.

Worked example: one pair, by hand (the HW7 checkpoint) Scores $u_o^\top v_c = 2$ for the real neighbour and $-1,\ 0.5$ for two noise words:

$$\ell = -\log \sigma(2) - \log \sigma(1) - \log \sigma(-0.5) = 0.1269 + 0.3133 + 0.9741 = 1.4143.$$

The model is already sure about the real pair ($\sigma(2) = 0.88$) and about the first noise word (it calls it real with probability $\sigma(-1) = 0.27$); the second noise word, which it calls real with probability

$\sigma(0.5) = 0.62$, contributes most. The gradient says the same thing: $\partial\ell/\partial s$ is $\sigma(2) - 1 = -0.12$ for the real pair (pull a little closer) and $\sigma(-1) = 0.27$, $\sigma(0.5) = 0.62$ for the noise words (push away, the second one hardest). At the start of training all scores are 0 and $\ell = 6 \log 2 = 4.16$ with $K = 5$; the reference run ends near 2.4.

Which noise words? They are drawn from the word frequencies raised to the power 3/4, which flattens the distribution a little: frequencies $(0.90, 0.09, 0.01)$ become noise probabilities $(0.825, 0.147, 0.028)$, so rare words are drawn somewhat more often than their frequency.

Subsampling frequent words. Words like `the` and `.` are next to everything and carry little information about their neighbours. Each occurrence of a word w with frequency $f(w)$ is *kept* with probability $\min\big(1, \sqrt{t/f(w)}\big)$, $t = 10^{-4}$ (the board writes the drop probability $\max(1-\sqrt{t/f(w)}, 0)$). In TinyStories, `the` ($f = 0.046$) is kept 4.7% of the time, `happy` ($f = 0.004$) 16%, `cat` 41%, and `dinosaur` always. This removes most of the data and makes training both faster and better.

Algorithm: skip-gram with negative sampling (SGNS), one pass

1. Subsample the token stream. For each center word, draw a window size $b \in \{1, \ldots, 5\}$ and take the words up to b steps away as its contexts — **never crossing into the next story**.
2. For each (center c, context o) pair, draw K noise words from frequency$^{3/4}$.
3. Scores: one row of V (center table) and $1 + K$ rows of U (context table), $1 + K$ dot products; shapes (n, d) and $(n, 1 + K, d) \to (n, 1 + K)$.
4. Loss ℓ averaged over the pairs; one Adam step on both tables.

After training, keep the **center** table as the word vectors.

Why stop windows at the end of a story? The last word of one story and the first words of the next have nothing to do with each other; letting them be "neighbours" would teach the model false pairs.

8 What the Vectors Learn

Two words are similar if their vectors point the same way. Table 5 lists cosine nearest neighbours from the reference run's skip-gram vectors, for words the homework does not ask about (it asks about a list of its own, and compares with gensim's word2vec, the standard implementation).

Word	6 nearest by cosine similarity
dog	barking, rex, bark, barked, leash, wagging
sad	upset, heartbroken, anymore, unhappy, angry, guilty
dad	mom, son, parents, john's, mum, dad's
jumped	jumping, sprang, leapt, dashed, straight, splashed
blue	yellow, red, bright, purple, orange, attractive
small	large, there, big, a, tiny, near

Table 5: Nearest neighbours, HW7 reference run (skip-gram with negative sampling, TinyStories, $d = 100$, 3 passes).

- **Similar or related, not synonyms.** The vectors record shared contexts, so *related* words that fill the same slot appear too: `large` and `big` next to `small` ("a _____ dog"), `leash` next to `dog`, `mom` next to `dad`. A few neighbours are just frequent words that share the slot (`a`, `there` next to `small`).
- **Against another implementation.** Two word2vec runs never give identical lists: different data pipelines, random initializations, subsamples and noise draws reorder close neighbours. Compare the *kind* of neighbours (same topic? same slot?), not the exact ranking.
- **In 2-D.** A PCA projection (Week 5) of 30 words from five groups (Figure 4): the weather, place and toy words form clusters; motion verbs and body parts are closer to each other. Reminder for the code: put one word per *row* (columns = the d features), centre each column, and take the top two rows of $V^\top$ from the SVD as the directions; a direction's sign is arbitrary, so a flipped plot is the same plot.

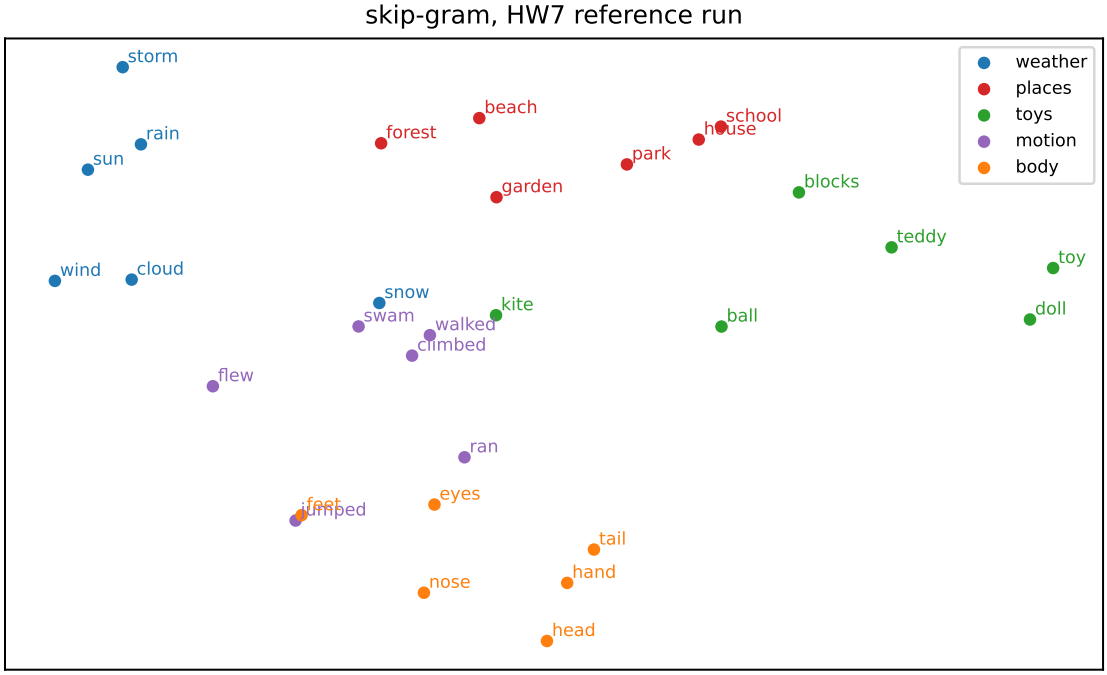


Figure 4: 2-D PCA of 30 word vectors (normalized to length 1) from the HW7 reference run's skip-gram.

Remark. These are **static** vectors: one vector per word, whatever the sentence. *bank* gets the same vector next to *river* and next to *money*. A Transformer starts from exactly such a table and then changes each vector according to its neighbours — the attention of the next note — producing *contextual* vectors.

9 HW7 Map

HW7 part	sections of this note	the effect to see
P1(A) BPE from scratch vs Hugging Face; side question	§4 (algorithm, toy, ties); §2–§3	the same merges until the first genuine tie; words turn rare text into `<unk>`, subwords and bytes do not
P1(B) SGNS scores and loss	§6, §7	the loss falls from $6 \log 2$; the checkpoint value 1.4143
P1(C) neighbours, PCA, gensim	§5, §8	clusters of related words; similar, not identical, lists from gensim
P12 (optional) larger corpora, GloVe	TOKENS_ADVANCED	—

Common bugs.

- **BPE:** counting each distinct word once instead of by its frequency (the toy's (u,g) becomes 3, not 8); counting pairs across the boundary between two words; merging an overlapping pair twice (`a a a` $\to$ `aa aa`); encoding a new word by "longest piece first" instead of the merges in learned order; treating a tie-broken difference from the library as a bug (check the counts first).
- **SGNS:** one table for both center and context vectors; the wrong sign on the noise term ($\log \sigma(+s)$ instead of $\log \sigma(-s)$); summing instead of averaging over pairs (the loss scale then depends on the batch size); taking the dot product over the wrong axis of the $(n, 1+K, d)$ tensor.
- **Neighbours:** ranking by dot product instead of cosine (long, frequent-word vectors win); returning the query word itself or `<unk>` as its own neighbour.
- **gensim not installed:** the notebook says so and uses the course's saved gensim results; nothing needs to be done.

10 Where This Goes Next

- **ATTENTION** (next lecture): every Transformer begins with exactly this embedding table, and attention turns the static vectors into contextual ones.
- **TRANSFORMER**: HW7's GPT learns its embedding table together with the model (as HW6 did) and reuses it as the output layer (weight tying).

- **TOKENS_ADVANCED** (optional): the full-softmax gradient, CBOW, GloVe, fastText, WordPiece and Unigram tokenizers, and what tokenizers do to other languages and to numbers.