Tokenization + Embeddings + Attention (Session 1)

Tokenization

- **User Input**
  "I deposited cash at the bank."

- **Tokenization**
  For this walkthrough, we will use a simplified token sequence: `["I", "deposited", "cash", "at", "the", "bank"]`
  Note: real GPT-style tokenizers often split punctuation separately and often attach leading spaces to tokens like `" bank"`.

- **Vocabulary Token ID**
  Convert each token into an ID that maps the token into the embedding matrix. In practice, the tokenizer usually does this step too.

token	id
I	0
deposited	1
cash	2
at	3
the	4
bank	5

Embeddings

- **Embedding Space**
  The embedding space is a huge multidimensional table of learned weights. For our example, the token embedding matrix looks like this:

$$E = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 2 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 2 \\ 0 & -1 \end{bmatrix}$$

- **Token Embedding Vector**
  Token IDs are used to index into the embedding matrix and retrieve token embeddings. Token embeddings turn tokens into learned vectors the model can do math with.

- In our case, `"bank"` token ID is `5`, and `E[5] = [1, 2]`. This turns `"bank"` into an embedding vector we can do math with: `[1, 2]`.
- At this stage, token embeddings do **not** disambiguate meaning. The token `"bank"` starts from the same embedding whether the sentence is about money or a riverbank.

- **Positional Information**
  For this walkthrough, we'll use the simplest version: add a position vector so the model knows word order.

$$P = \begin{bmatrix} 0.1 & 0.0 \\ 0.0 & 0.1 \\ 0.1 & 0.1 \\ 0.0 & 0.2 \\ 0.2 & 0.0 \\ 0.3 & -0.1 \end{bmatrix}$$

  Real models use a few different tricks for this.

  Tokens: `["I", "deposited", "cash", "at", "the", "bank"]`
  Positions are 0-based: `0, 1, 2, 3, 4, 5`

- **Positional Vector**
  Positional information tells the model about token order. For the token `"bank"` at position 5: Position index: `5` or `P[5] = [0.3, -0.1]`

- **Position-Aware Input Embedding (LAYER 0)**
  For each token, produce a single vector that contains both token identity and position information.

  - Token embedding for `"bank"`: `[1, 2]`
  - Positional vector for `"bank"`: `[0.3, -0.1]`
  - Combined: `[1, 2] + [0.3, -0.1] = [1.3, 1.9]`

- **Embedding Math**

  - Look up real embedding vectors for words A, B, and C
  - Compute `A - B + C` element-wise in the full embedding space
  - Brute-force scan the entire vocabulary using cosine similarity to find nearest neighbors
  - Return top matches ranked by similarity score

Attention

Attention mixes information across tokens: when we processed "bank", attention pulled in stuff from "deposited" and "cash", etc.

- **Self-Attention**
  Self-attention is one of the two main mechanisms in a transformer layer (the other is the feed-forward network). It helps the model build mathematical context-dependent representations from a sequence. In a chatbot-style decoder-only transformer, each token can look at earlier tokens and itself, but not at future tokens. Self-attention happens in many layers (for example, GPT-3 has 96 layers). In this sentence, `"bank"` is at

the end, so it can still attend to every earlier token in our example. Examining how the transformer processes the word `"bank"`:

- **Linear Projections**
  Matrix multiplication re-expresses the input embeddings into Query (Q), Key (K), and Value (V) spaces. For this simplified example, treat the input vectors as `Q = K = V`. In a real model, Q, K, and V come from separate learned projections, and the dot products are usually scaled before softmax.

  - Bank's representation vector: `[1.3, 1.9]`
  - Other representation vectors:
    - I: `[1.1, 0.0]`
    - deposited: `[2.0, 1.1]`
    - cash: `[2.1, 0.1]`
    - at: `[0.0, 1.2]`
    - the: `[0.2, 0.0]`
    - bank: `[1.3, 1.9]`

- **Dot Product**
  Dot product measures alignment: multiply corresponding vector entries and sum to a scalar. Dot products

$$\mathbf{x} = \begin{bmatrix} a & b \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} c \\ d \end{bmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}\mathbf{y} = \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = ac + bd$$

  produce attention scores between tokens.

  Example dot products of `"bank"` with every token it is allowed to attend to:

  - with I: `[1.3, 1.9] . [1.1, 0.0] = 1.3*1.1 + 1.9*0.0 = 1.43`
  - with deposited: `1.3*2.0 + 1.9*1.1 = 2.6 + 2.09 = 4.69`
  - with cash: `1.3*2.1 + 1.9*0.1 = 2.73 + 0.19 = 2.92`
  - with at: `1.3*0.0 + 1.9*1.2 = 2.28`
  - with the: `1.3*0.2 + 1.9*0.0 = 0.26`
  - with bank: `1.3^2 + 1.9^2 = 1.69 + 3.61 = 5.30` Raw attention scores for `"bank"`: `[1.43, 4.69, 2.92, 2.28, 0.26, 5.30]` From these, `"bank"` would put about 58% of its attention on itself and about 32% on `"deposited"`, with much smaller weights on other tokens.

- **Attention Head**
  A "head" is an independent self-attention mechanism running in parallel inside a layer. In this example, the head focuses mainly on two positions for `"bank"`: itself and `"deposited"` (`0.5841 + 0.3174 ~= 0.9015`, or about 90% of the attention mass). That means this head updates `"bank"` mostly using information from itself and `"deposited"`. Disambiguation emerges across many heads, many layers, and the downstream FFN.

Softmax (step 5 of 10) Softmax converts attention logits into normalized attention weights that sum to 1. Example weights:

- I: 0.0122
- deposited: 0.3174
- cash: 0.0541
- at: 0.0285
- the: 0.0038
- bank: 0.5841

Weighted Sum The weighted sum produces a context-mixed vector by mixing value vectors from different tokens using attention weights. In a real transformer, that is not the final output of the whole layer yet. The layer still combines this with the original token vector and keeps processing. Compute component-wise weighted sums:

- x = 0.0122(1.1) + 0.3174(2.0) + 0.0541(2.1) + 0.0285(0.0) + 0.0038(0.2) + 0.5841(1.3) ~= 1.52
- y = 0.0122(0.0) + 0.3174(1.1) + 0.0541(0.1) + 0.0285(1.2) + 0.0038(0.0) + 0.5841(1.9) ~= 1.50 Resulting vector ~= `[1.52, 1.50]`

Feed-Forward Network (FFN)

Unlike Attention, FFN never looks at other tokens. It takes one token vector at a time, all by itself, runs it through a transformation, and hands the result back as the new token vector for that token. Then it does the same thing to the next token.

Not in scope for today's discussion