AI for Business: Insights from Corporate Data

Topic 8: Foundations of NLP - from Tokenization to Transformer

Miao Liu

Boston College

March 23, 2026

Overview: Topic 8

1. Introduction to NLP
2. Feature Extraction: from Vectorization to Embedding
   - Simple Word Vectorization
   - Word Embedding
3. Transformer: Contextual Embeddings
   - Overview of Transformers
   - Positional Encoding
   - Attention Mechanisms

Overview: Topic 8

What is Natural Language Processing?

- **Definition:** NLP enables computers to understand, interpret, and generate human language
- **Core Goal:** Bridge the gap between human communication and computer understanding

Applications of NLP

Why Human Language is Hard

- We have an ocean of information in human-language format. We need a tool to turn such information into actionable insights.
- But this is hard - We often think of language as a set of rules that govern how words can be combined to form valid sentences, but there are countless exceptions that are hard to manage.

Why Human Language is Hard

- **Ambiguity:**
  - Natural language is inherently ambiguous. "I saw the man with the binoculars" can mean either you saw a man who was using or carrying binoculars.
  - Many words have multiple meanings depending on context
- **Ever Changing:**
  - New words are created and existing words change meanings over time (e.g., cool)
- **Contextual Information:**
  - Understanding meaning often relies on context; "I'm hungry" might imply physical hunger or an emotional state.
- **Sarcasm:**reverses the literal meaning
- **Idioms and domain-specific language:**
  - Idioms, such as "it's raining cats and dogs," convey figurative meanings that are not easily deduced from the individual words.

Overview: Topic 8

Feature Extraction: Overview

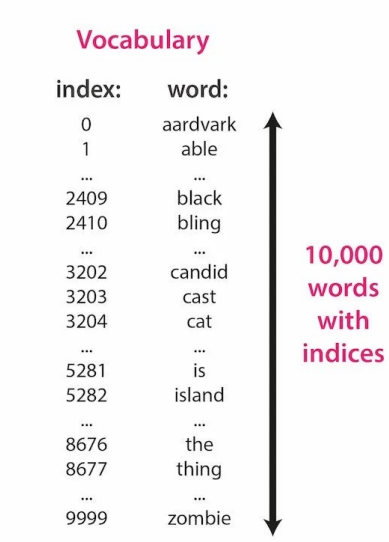
- **Objective:** Convert preprocessed text into numerical representations that computers can understand.
- **Methods:**
  - **Simple Word Vectorization:** Represents text using simple frequency counts such as Bag-of-Words and TF-ID.
  - **Word Embedding:** Creates dense, low-dimensional vectors that capture semantic relationships.

One-Hot Encoding: Introduction

- One-hot encoding is a basic method for representing words in NLP.
- Each word in the vocabulary is represented as a unique vector.
- In the vector, all elements are 0 except one position (the word's index) which is set to 1.

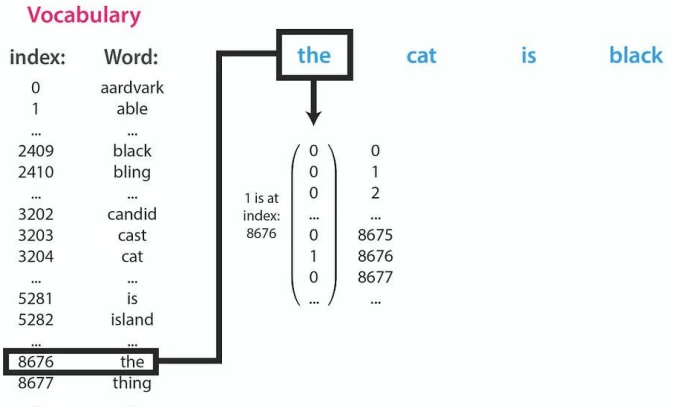
One-Hot Encoding: Vocabulary Representation

- Imagine a vocabulary of 10,000 words.
- Each word is assigned a unique integer index from 1 to 10,000.
- **Example:**
  - The word **aardvark** (first in the vocabulary) is represented as a vector with a 1 in the first position and 9,999 zeros.
  - The word **ant** (second in the vocabulary) is represented as a vector with a 1 in the second position and 9,998 zeros.



One-Hot Encoding: Vocabulary Representation

- Consider a translation task: Translating "the cat is black" into another language.
- Repeat for each word:
  - Look up the word's index in a 10,000-word vocabulary.
  - Represent the word as a 10,000-dimensional vector with 1 at the corresponding index.



One-Hot Encoding: Limitations and Importance

- **Lack of Semantic Similarity:**
  - One-hot encoding treats each word as an independent unit.
  - Words with similar meanings (e.g., "cat" and "tiger") have completely distinct representations.
- **High Dimensionality & Inefficiency:**
  - The vector size grows linearly with the vocabulary size, resulting in very high-dimensional, sparse representations.
  - This increases computational and memory demands, making it less efficient for large datasets.
- Despite these limitations, one-hot encoding provides a simple and effective way to numerically represent text, serving as a foundation for more advanced techniques like word embeddings.

Bag-of-Words (BoW) Model: Overview

- **Concept:** Represents each document as a vector of word counts.
- **Key Definitions:**
  - **Document:** A single piece of text (e.g., an article, a report, a tweet).
  - **Corpus:** The entire collection of documents used for analysis.
- **Process:**
  - Build a vocabulary from the entire corpus.
  - For each document, count the number of times each word from the vocabulary appears.
- **Relation to One-Hot Encoding:**
  - One-hot encoding represents individual words as sparse vectors.
  - BoW aggregates these individual representations into a document-level feature vector.

Bag-of-Words (BoW) Model: Example and Limitations

- **Example:**
  - Consider the sentences:
    - Sentence 1: "cat sat on the mat"
    - Sentence 2: "the cat is on the mat"
  - **Vocabulary:** [cat, sat, on, the, mat, is]
  - **BoW Vectors:**
    - Sentence 1: [1, 1, 1, 1, 1, 0]
    - Sentence 2: [1, 0, 1, 2, 1, 1]
- **Limitations:**
  - Ignores word order and context.
  - Produces high-dimensional, sparse representations.

Bag-of-Words (BoW) Model: Benefits and Transition

- **Benefits:**
  - Simple and easy to implement.
  - Effective for tasks like document classification and topic modeling.
- **Transition to Advanced Methods:**
  - BoW lays the foundation for more sophisticated techniques like TF-IDF, which weighs words by importance.
  - It also serves as a stepping stone towards word embeddings that capture semantic relationships and context.

TF-IDF: Term Frequency (TF) and Inverse Document Frequency (IDF)

- **Term Frequency (TF):**
  - Measures how frequently a term appears in a document.
  - Often normalized by the total number of terms in the document to avoid bias toward longer texts.
  - Example: In a document with 100 words, if the term "data" appears 5 times, then TF("data") $= 5/100$.
- **Inverse Document Frequency (IDF):**
  - Measures the commonality of a term across the entire corpus.
  - Formula:

$$\text{IDF}(t, D) = \log\left(\frac{N}{\text{DF}(t) + 1}\right)$$

    where N is the total number of documents and $\text{DF}(t)$ is the number of documents containing term t.
  - The "$+1$" prevents division by zero.

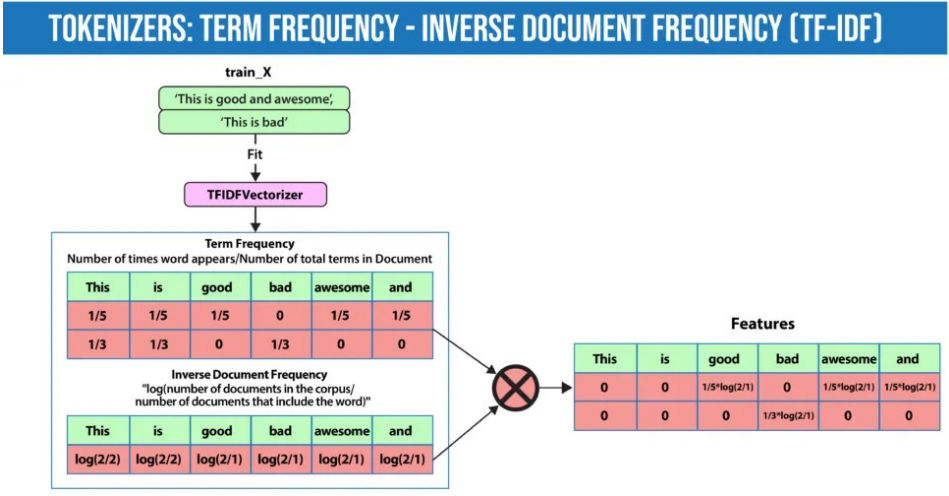
TF-IDF: Combining TF and IDF

- **Calculation:**
  - The TF-IDF score for a term t in document d is given by:

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

- **Intuition:**
  - Words that appear frequently in a document but rarely across the corpus receive higher weights.
  - This highlights important terms that are more indicative of the document's content.
- **Application:**
  - Commonly used in information retrieval and text mining to identify key topics and for document classification.

TF-IDF: an Example



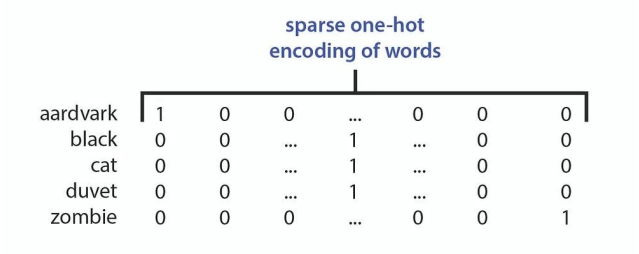
TF-IDF creates features for each document based on how often each word shows up in a document versus the entire corpus.

Word Embeddings: Introduction

- **Concept:** Generates dense, low-dimensional vectors where words with similar meanings are represented by similar vectors.
- **Approaches:**
  - Static Embeddings (e.g., Word2Vec) learn embeddings based on word co-occurrence.
  - Contextual Embeddings (e.g., BERT) adjust word representations based on surrounding context.
- **Example:**
  - Words such as "king" and "queen" will have similar vector representations that capture their related meanings.

Word Embeddings: From Sparse to Dense Representations

- Traditional one-hot encoding produces very sparse vectors where each word is represented as a high-dimensional vector with all zeros except a single 1.
- Imagine a vocabulary with words like "aardvark", "black", "cat", "duvet", and "zombie". One-hot encoding represents each word as a sparse vector.



- However, as humans, we know that words carry rich semantic meanings and relationships that one-hot encoding cannot capture.

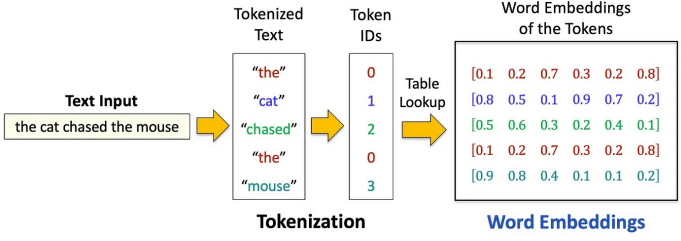
Word Embeddings: From Sparse to Dense Representations

- To capture semantic meaning, we can manually assign each word a set of features.
- Consider four semantic qualities: *animal*, *fluffiness*, *dangerous*, and *spooky*.
- Example:
  - "aardvark": high on *animal*, low on *fluffiness*, *dangerous*, and *spooky*.
  - "cat": high on *animal* and *fluffiness*, medium on *dangerous* and *spooky*.
- Each feature can be viewed as a dimension in a dense semantic space.

	animal	fluffiness	dangerous	spooky
aardvark	0.97	0.03	0.15	0.04
black	0.07	0.01	0.20	0.95
cat	0.98	0.98	0.45	0.35
duvet	0.01	0.84	0.12	0.02
zombie	0.74	0.05	0.98	0.93

Word2Vec: Embedding Dimensionality

- **Dimensionality:** The number of features used to represent each word.
- In previous example, each semantic quality (e.g., animal, spooky) is one dimension.
- In modern word embeddings, the dimensionality is typically abstract and fixed (e.g., 50 to 300 dimensions), capturing many aspects of meaning.
- Different methods yield different dimensions; higher dimensions can capture more nuance but may require more data.

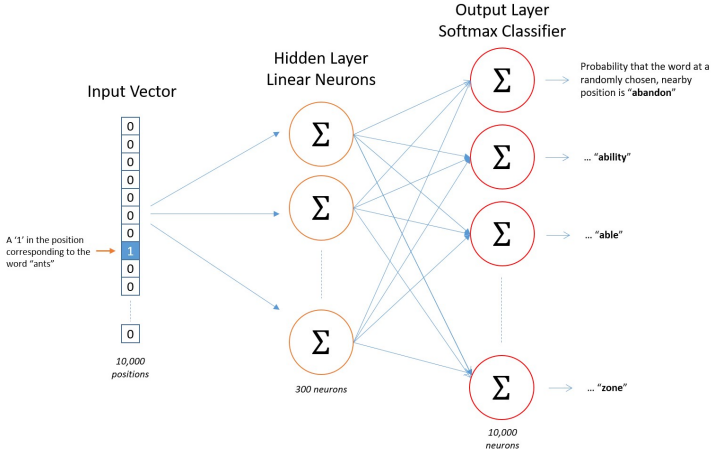


Word2Vec: Model Architecture

- **Overview:** Word2Vec learns dense, semantically-meaningful embeddings from large text corpora.
- **Architecture:** A shallow two-layer neural network
  - Input: One-hot encoded vectors of words from the training corpus.
  - Hidden Layer: Contains a fixed number of neurons equal to the desired embedding dimension (e.g., 300 neurons).
  - Output: Probability distribution over the vocabulary predicting the target word.

Word2Vec: Model Architecture

- **Architecture:**
  - Shallow two-layer neural network
  - **Input:** One-hot encoded vectors of words
  - **Hidden Layer:** Fixed number of neurons, defining embedding dimension
  - **Output:** Probability distribution of target word



- The hidden layer weights become the learned word embeddings.

Intuitive Explanation of Word2Vec Training

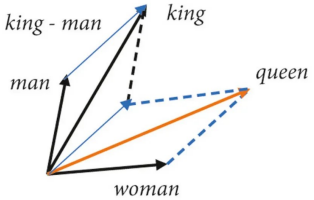
- **Objective:** Learn dense, semantically rich word embeddings.
- **Training Intuition:**
  - The network adjusts the hidden layer weights (word vectors) based on prediction accuracy.
  - Words used in similar contexts learn similar representations.
- **Outcome:**
  - Produces dense embeddings that capture semantic relationships, unlike sparse one-hot encodings.

Semantic Relationships in Word2Vec

- Word2Vec can capture intricate semantic relationships between words.
- **Example:**

$$\text{embed}(\text{"king"}) - \text{embed}(\text{"man"}) + \text{embed}(\text{"woman"}) \approx \text{embed}(\text{"queen"})$$

- Vector difference between "king" and "man" captures the concept of "royalty" minus "male," and adding "female" results in "queen."



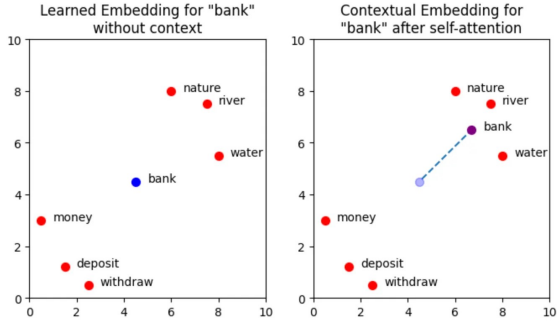
Introduction to Contextual Embeddings

- **Motivation:**
  - Static embeddings (e.g., Word2Vec) assign a single vector to each word regardless of its context.
  - In natural language, words can have different meanings depending on their surrounding context (e.g., "bank" in "river bank" vs. "bank account").
- **What Are Contextual Embeddings?**
  - They generate dynamic representations of words that change according to the words around them.
  - These embeddings capture the nuanced, context-dependent meaning of words.
- **Benefits:**
  - Enhance performance in tasks such as sentiment analysis, question answering, and machine translation.
  - Improve disambiguation of word meanings and capture complex linguistic phenomena.

Overview: Topic 8

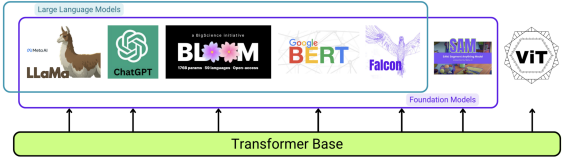
The Issue with Static Embeddings

- Traditional embedding models like Word2Vec produce static embeddings: each word is assigned one fixed vector, and fail to capture context.
- Without context, "bank" might show similarity to both financial terms (money, deposit) and geographical terms (river, nature).
- The core problem: static embeddings do not change based on surrounding words.



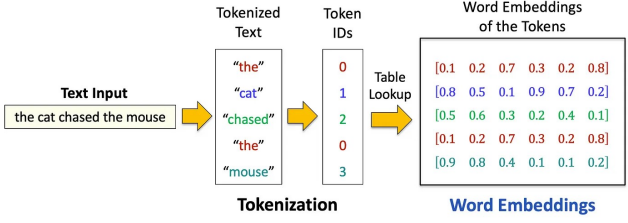
Fixing Static Embeddings with Transformers

- **Transformers:** a deep learning model that overcomes static limitations by creating **context-aware embeddings**.
- They enhance fixed word vectors by incorporating:
  - **Positional Encoding:** Provides information on word order.
  - **Self-Attention Blocks:** Evaluate relationships among all words in the sentence.
- It allows the same word to have different representations depending on context.
- Transformers form the foundation for nearly all modern large language models.



Transformer Embedding Process

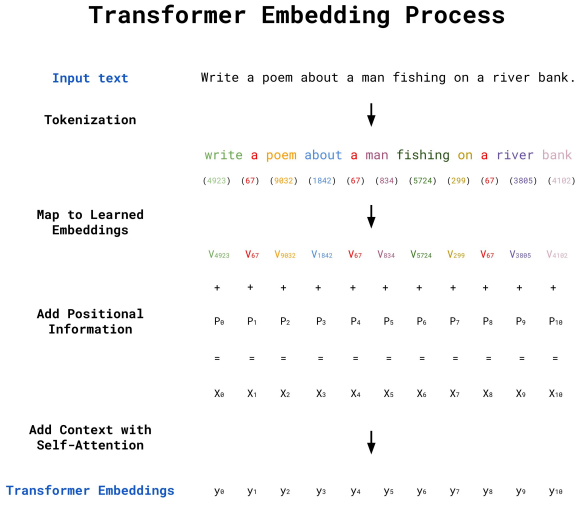
- **Step 1: Tokenization**
  - The input text is split into tokens (e.g., words or subwords).
  - Each token is assigned a unique token ID.
  - **Note:** This step is very similar to the tokenization used in Word2Vec.
- **Step 2: Mapping to Learned Embeddings**
  - Token IDs are used to extract dense vector representations from the embedding layer.
  - **Note:** This lookup process mirrors how Word2Vec obtains static embeddings.
  - Calling them "learned" emphasizes that these initial representations are a starting point, later enriched with contextual information.



Transformer Embedding Process

- **Step 3: Adding Positional Encoding**
  - Positional encoding is added to incorporate word order information.
  - **Intuition:** It provides each token with a "positional badge" so the model knows the order in which words appear.
- **Step 4: Applying Self-Attention**
  - The self-attention mechanism refines the embeddings by integrating contextual information from the entire sequence.
  - **Intuition:** It allows each word to "look at" every other word and adjust its representation based on their relevance.
- The final output is a set of **contextual embeddings** that capture both the inherent meaning of each token and its usage in context.

Transformer Embedding Process



Positional Encoding: How It's Done

- Each word in a sentence is initially represented by a vector (its embedding) that does not include any positional information.
- Positional encoding creates an additional vector for each word, acting like an "address" for the word, indicating its position in the sentence.
- The positional vector is then added to the word's embedding so that each word now carries both its meaning and its position.

Positional Encoding: Why It's Important

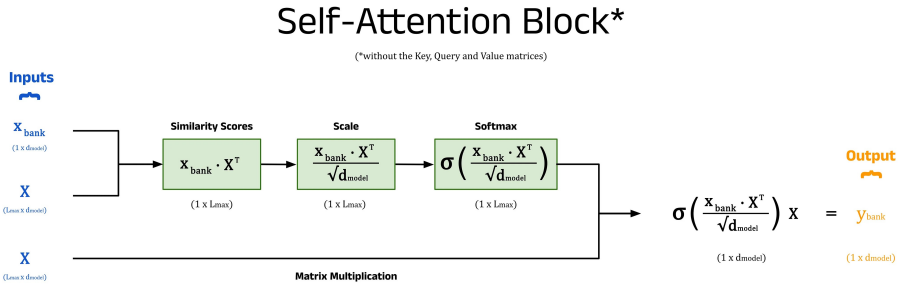
- When a transformer processes a sentence, it handles all words in parallel using the combined embeddings (word + position).
- Positional encoding tells the model "where" each word is in the sentence, allowing it to understand the sequence despite parallel processing.

Transition: From Positional Encoding to Self-Attention

- **Linking Two Key Components:**
  - Positional encoding gives each word its "address" (its position in the sentence).
  - Self-attention uses these enriched embeddings to determine relationships between all words.
- **Why Self-Attention Needs Positional Encoding:**
  - Without positional information, self-attention treats the input as an unordered bag of words.
- **Intuitive Connection:**
  - Think of positional encoding as providing each word with a location tag.
  - Self-attention then "reads" these tags along with the word meanings, so it can weigh the relevance of each word based on both content and position.

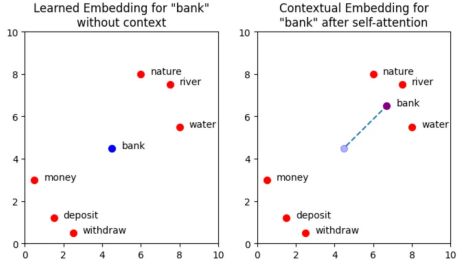
Self-Attention Mechanism: Overview

- Self-attention adjusts each word's vector to capture the context provided by all other words in the same sequence.
- The "self" in self-attention means that the mechanism uses the surrounding tokens within a single sentence for context.



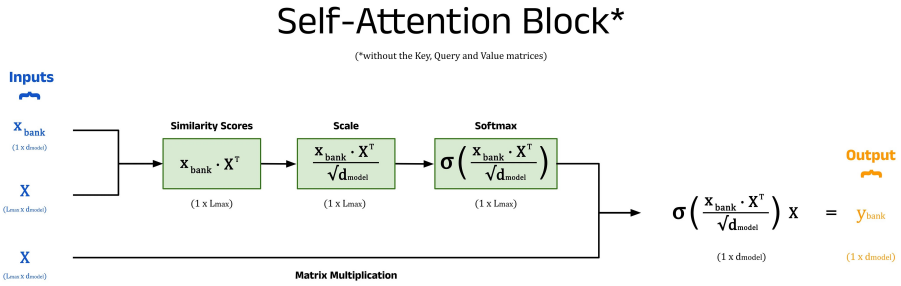
Visualizing Self-Attention in Action

- Consider a 2D plot of learned embeddings:
  - Words related to nature and rivers cluster in one region.
  - Words related to money cluster in another.
- The word **bank** starts at an ambiguous position between these clusters.
- Self-attention shifts **bank**'s embedding closer to tokens that provide the desired context (e.g., "river", "fishing").



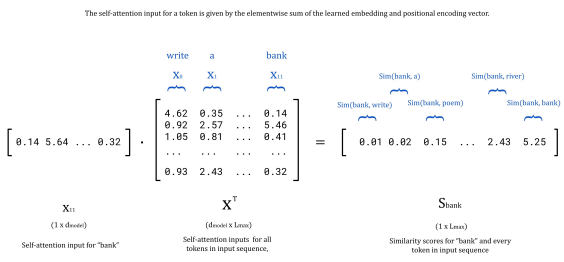
Self-Attention Mechanism: Overview

- Input: Write a poem about a man fishing on a river bank.



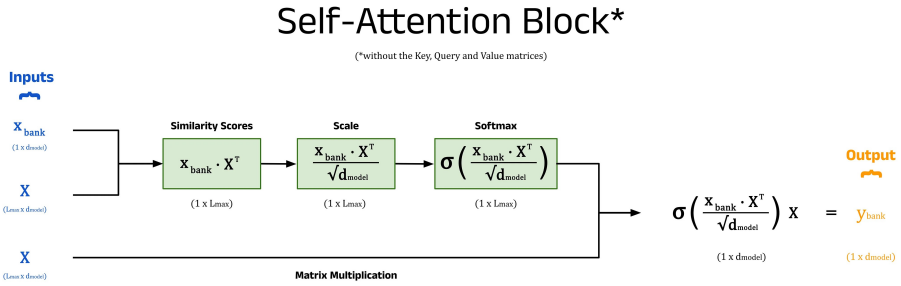
Self-Attention Algorithm: Step 1 – Calculating Similarity

- **Goal:** Update the embedding for a token (e.g., **bank**) using context.
- For each token, compute similarity scores with all other tokens via dot product
  - Similarity scores indicate how much one word should "pay attention" to another.
  - The vector for **bank** (x_{bank}) is compared with every token's vector in the input.
- These similarity scores form a vector $S_{\text{bank}} = x_{\text{bank}} \cdot X^{\top}$, where X is the matrix of learned token embeddings.



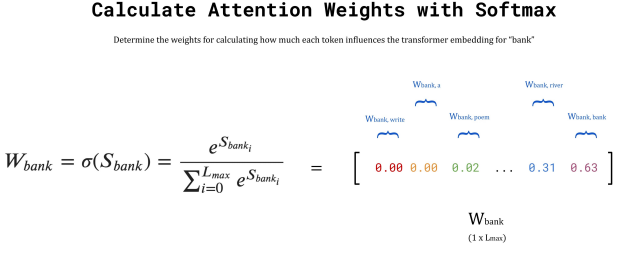
Self-Attention Mechanism: Overview

- Input: Write a poem about a man fishing on a river bank.



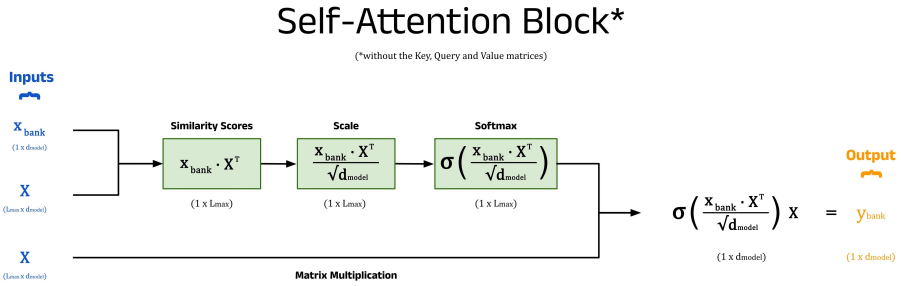
Self-Attention Algorithm: Step 2,3 – Scaling and Softmax

- **Scaling:**
  - Divide the dot product scores by $\sqrt{d_{\text{model}}}$ to control for large values.
  - Prevents the scores from becoming too large, which could skew the attention.
- **Softmax:**
  - Convert the scaled scores into a probability distribution (attention weights).
- The resulting attention weights vector, W_{bank}, tells the model how much each token contributes to updating **bank**'s embedding.



Self-Attention Mechanism: Overview

- Input: Write a poem about a man fishing on a river bank.



Self-Attention Algorithm: Step 4 – Weighted Sum

- **Step 4:** Compute the new embedding for **bank** by taking a weighted sum of all token embeddings:

$$y_{\text{bank}} = \sum_i W_{\text{bank}}(i) \cdot x_i.$$

- **Intuition:** Tokens with higher attention weights (e.g., "river" and "fishing") pull **bank**'s embedding towards the intended context.
- The final result is a **contextual embedding** y_{bank} that better reflects the meaning of **bank** in the given sentence.