Text Classification and Language Models

Chapter 5

Text Classification

Text classification categorizes text into predefined labels or classes using machine learning techniques, a core task in natural language processing.

It powers applications like spam detection, sentiment analysis, and topic labeling.

Common approaches include traditional methods like TF-IDF with Naive Bayes and modern deep learning models such as BERT.

Text Classification

Categorized text into multiple classes

- To learn which news articles are of interest
- To classify web pages by topic
- Which reviews are good and bad

Naive Bayes

Simple and baseline effective algorithm

Assume that the classification attributes are independent, they do not have any correlation between them

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Naive Bayes Classification

Learning Phase

- For a training set T of objects, let
- T_c = the set of documents from class c
- n_{wc} = # occurrences of word w in documents from Tc
- n_c = # occurences of all word in documents from Tc

We compute pc and pwc using a simple maximum likelihood estimation (MLE)

$p_c = |T_c|/|T|$ global likelihood of a class

$p_{wc} = n_{wc} / n_c$ likelihood of a word for a class

Naive Bayes Classification

Consider following artifical document collection

Doc	Class	Doc	Class
aba	A	bbaabbab	B
baabaaa	A	abbb	B
abbaa	A	bbbaab	B

- Let us predict the class for aab
  $\cdots$ A or B?
- Using Bayes Rule

$$P(A|d = aab) = \frac{P(d = aab|A)P(A)}{P(d = aab)}$$
$$= \frac{P(a|A)P(a|A)P(b|A)P(A)}{P(d = aab)}$$

$$P(A) = P(B) = \frac{1}{2}$$
$$P(a|A) = \frac{10}{15} = \frac{2}{3}$$
$$P(b|A) = \frac{5}{15} = \frac{1}{3}$$
$$P(a|B) = \frac{6}{18} = \frac{1}{3}$$
$$P(b|B) = \frac{12}{18} = \frac{2}{3}$$

Naive Bayes Classification

$$P(A|d = aab) = \frac{2}{3}\frac{2}{3}\frac{1}{3}\frac{1}{2}\frac{1}{P(d = aab)}$$

$$P(B|d = aab) = \frac{P(d = aab|B)P(B)}{P(d = aab)} = \frac{P(a|B)P(a|B)P(b|B)P(B)}{P(d = aab)}$$

$$= \frac{1}{3}\frac{1}{3}\frac{2}{3}\frac{1}{2}\frac{1}{P(d = aab)}$$

$$P(A|d = aab) > P(B|d = aab)$$

Hence, the $P(A|aab) > P(B|aab)$, the document aab lies on class **A**

Text Classification: Example 2

Doc	Class	Doc	Class
abac	A	bbaabbab	B
baabaaa	A	abbb	B
abbaa	A	bbbaab	B

- Let us predict the class for d=aabc $\cdots$ A or B?

$$P(A) = P(B) = \frac{1}{2}$$

$$P(a|A) = \frac{10}{16} = \frac{5}{8}$$

$$P(b|A) = \frac{5}{16}$$

$$P(c|A) = \frac{1}{16}$$

$$P(a|B) = \frac{6}{18} = \frac{1}{3}$$

$$P(b|B) = \frac{12}{18} = \frac{2}{3}$$

$$P(c|B) = \frac{0}{18} = 0$$

Doc	Class	Doc	Class
abac	A	bbaabbab	B
baabaaa	A	abbb	B
abbaa	A	bbbaab	B

- Let us predict the class for d=aabc $\cdots$ A or B?
- Using Bayes Rule

$$P(B|d) = \frac{P(d = aabc|B)P(B)}{P(d)}$$

$$= \frac{P(a|B)P(a|B)P(b|B)P(c|B)P(B)}{P(d)}$$

$$= 0 \quad [\because P(c|B) = 0]$$

Problem

$P(C= c|D= d) = 0$ if only one single $P(w|c) = 0$

This happens easily, when test set (d) contains a word that did not occur in the training set for class c

How to deal with zero probability problem?

Laplace Smoothing

Laplace smoothing (or add-one smoothing) is a technique used in probability estimation—commonly in Naive Bayes classifiers—to avoid zero-probability issues for unseen data.

It works by adding a small constant, typically 1 (alpha =1), to the numerator of each feature count and adding the vocabulary size (|V|) to the denominator. This ensures all features have a non-zero probability.

Laplace Smoothing

Smoothing technique that handles the problem of zero probability in Naive Bayes

Using Laplace smoothing or 1 add, we can represent P(w |c) = 0

$$P(w|c) = \frac{n_{wc} + \epsilon}{n_c + (\epsilon * K)}$$

ϵ= 1: smoothing parameter where ϵ≠ 0

K represents the number of dimensions (features) in the data

This is like adding every word ϵ times for every class

Doc	Class
abac	A
baabaaa	A
bbaabbab	B
abbb	B
abbaa	A
bbbaab	B

- Let $\epsilon = 1$ and $K = 3$ total number of term (unique word)
- Let us predict the class for d=aabc $\cdots$ A or B?

$$P(A) = P(B) = \frac{1}{2}$$

$$P(a|A) = \frac{10+1}{16+3} = \frac{11}{19}$$

$$P(b|A) = \frac{5+1}{16+3} = \frac{6}{19}$$

$$P(c|A) = \frac{1+1}{16+3} = \frac{2}{19}$$

$$P(a|B) = \frac{6+1}{18+3} = \frac{9}{21}$$

$$P(b|B) = \frac{12+1}{18+3} = \frac{13}{21}$$

$$P(c|B) = \frac{0+1}{18+3} = \frac{1}{21}$$

Using Laplace's smoothing

$$P(w|c) = \frac{n_{wc} + \epsilon}{n_c + (\epsilon \times K)}$$

Using Naive Bayes

$$P(A|d) = \frac{P(d|A)P(A)}{P(d)} = \frac{P(a|A)P(a|A)P(b|A)P(c|A)P(A)}{P(d)}$$

$$P(B|d) = \frac{P(d|B)P(B)}{P(d)} = \frac{P(a|B)P(a|B)P(b|B)P(c|B)P(B)}{P(d)}$$

Predict class based on Probability of $P(A|d)$ and $P(B|d)$

N-gram models: unigram, bigram and trigram

N-gram language models predict the probability of a word based on the preceding (n-1) words.

They are fundamental to many NLP tasks including speech recognition, machine translation, spell checking, and text generation.

Question

Calculate the probability of the sentence "I love NLP" using unigram, bigram, and trigram models.

Training Corpus:

"I love machine learning"

"I love NLP"

"NLP is amazing"

"I study NLP and machine learning"

Solution:

Assume each line is a sentence, start with "<s>" and end with "</s>" for modeling:

<s> I love machine learning </s>

<s> I love NLP </s>

<s> NLP is amazing </s>

<s> I study NLP and machine learning </s>

Vocabulary V: all distinct words in the corpus, plus <s> and </s>

List vocabulary and word counts

First, list all tokens in the corpus (split by space):

Sentence 1: <s>, I, love, machine, learning, </s>

Sentence 2: <s>, I, love, NLP, </s>

Sentence 3: <s>, NLP, is, amazing, </s>

Sentence 4: <s>, I, study, NLP, and, machine, learning, </s>

Count each word (case-sensitive, tokens only as given, NLP is a token):

<s>: 4

</s>: 4

I: 3

love: 2

machine: 2

learning: 2

NLP: 3

is: 1

amazing: 1

study: 1

and: 1

Total tokens (including <s> and </s>): 24 tokens.

Unigram Model

We want P(I love NLP) = P(I) × P(love) × P(NLP) in unigram model (no sentence boundaries in probability for the string itself unless we include start/end).

Actually, if we model sentence probability including <s> and </s>, then:

P(<s> I love NLP </s>) = P(I | <s>) × … — but for plain unigram, we just multiply probabilities of each word ignoring order (but given order, we use each unigram prob).

Unigram Model

P(<s>) = 4/24 = 1/6

P(I) = 3/24 = 1/8

P(love) = 2/24 = 1/12

P(NLP) = 3/24 = 1/8

P(</s>) = 4/24 = ⅙

P_{uni}(<s> I love NLP </s>) = ⅙ * ⅛ * 1/12 * ⅛ * ⅙ = 1/27648

P_{uni}("I love NLP" as sentence) = 1/27648

Bigram Model

We need bigram counts first.

Corpus with <s> and </s>:

1: <s> I, I love, love machine, machine learning, learning </s>

2: <s> I, I love, love NLP, NLP </s>

3: <s> NLP, NLP is, is amazing, amazing </s>

4: <s> I, I study, study NLP, NLP and, and machine, machine learning, learning </s>

Bigram Model

<s> I: 3

<s> NLP: 1

I love: 2

I study: 1

love machine: 1

love NLP: 1

machine learning: 2

learning </s>: 2

NLP </s>: 1

NLP is: 1

NLP and: 1

is amazing: 1

amazing </s>: 1

study NLP: 1

and machine: 1

For sentence <s> I love NLP </s>:

Bigram probabilities (MLE):

P(I | <s>) = count(<s> I)/count(<s>) = 3/4

P(love | I) = count(I love)/count(I) = 2/3 (count(I) as first word of bigram: I love 2, I study 1, total = 3)

P(NLP | love) = count(love NLP)/count(love) = 1/2 (count(love) as first word of bigram: love machine=1, love NLP=1, total=2)

P(</s> | NLP) = count(NLP </s>)/count(NLP) as first word = 1/3 (count(NLP as first word in bigram): NLP </s>=1, NLP is=1, NLP and=1, total=3)

P_{bi}(<s> I love NLP </s>) = P(I | <s>) * P(love | I)* P(NLP | love) * P(</s> | NLP)
= ¾ * ⅔ * ½ * ⅓ =1/12

Trigram Model

1:

<s> I love

I love machine

love machine learning

machine learning </s>

2:

<s> I love

I love NLP

love NLP </s>

3:

<s> NLP is

NLP is amazing

is amazing </s>

4:

<s> I study

I study NLP

study NLP and

NLP and machine

and machine learning

machine learning </s>

Trigram counts

<s> I love: 2

I love machine: 1

love machine learning: 1

machine learning </s>: 2

<s> I study: 1

<s> NLP is: 1

I love NLP: 1

love NLP </s>: 1

NLP is amazing: 1

is amazing </s>: 1

I study NLP: 1

study NLP and: 1

NLP and machine: 1

and machine learning: 1

Trigram

Given short sentence: <s> w1 w2 w3 </s>, in trigram:

Step1: P(w1 | <s>) — bigram

Step2: P(w2 | <s> w1) — trigram

Step3: P(w3 | w1 w2) — trigram

Step4: P(</s> | w2 w3) — trigram

Our sentence: w1=I, w2=love, w3=NLP.

P(I | <s>) = ¾

P(love | <s> I) = count(<s> I love)/count(<s> I) = ⅔

Actually for P(NLP | I love): Look at trigrams where first two words are I love. Those are: I love machine (sent1), I love NLP (sent2). So indeed there are two trigrams with I love as first two words:

(1) I love machine

(2) I love NLP

So count(I love *) = 2

count(I love NLP) = 1

Thus P(NLP | I love) = 1/2.

P(</s> | love NLP) = count(love NLP </s>)/count(love NLP)

So probability, P_{tri}(“I love NLP”) = ¾ *⅔ * ½ *1 = ¼

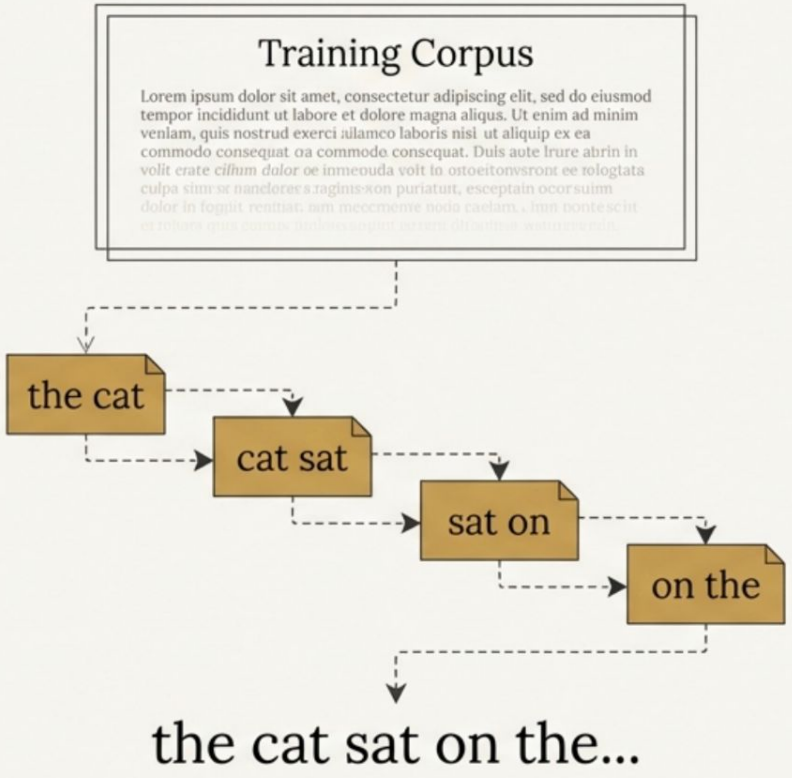
In the Beginning, Language Was a Chain of Probabilities

Key Concept: Statistical Language Models (SLMs) aimed to learn the joint probability function of word sequences. The core task was to calculate the probability of the next word given the previous ones: $P(w_t \mid w_{t-1}, w_{t-2}, \ldots)$.

The Dominant Approach: N-grams

SLMs simplified the problem by looking at a fixed context of the last *n-1* words.

Example (Trigram, n=3): To predict the next word in 'the cat sat __', the model looks up the probability P(word |'cat sat') in a massive table of pre-computed frequencies from a training corpus.



The Wall All Statistical Models Hit: The Curse of Dimensionality

The Problem Defined
With a **large vocabulary** (e.g., 100,000 words), the number of possible word combinations grows exponentially. For a 10-word sequence, there are $100{,}000^{10}$ possible combinations. It is impossible to see them all during training.

The Consequence
Any **new sentence** is likely different from every sentence seen in training. N-gram models have no way to generalize. They treat words as discrete, unrelated symbols.

The 2003 Breakthrough: What if Words Themselves Had Meaningful Representations?

The Core Idea (from Bengio et al.):

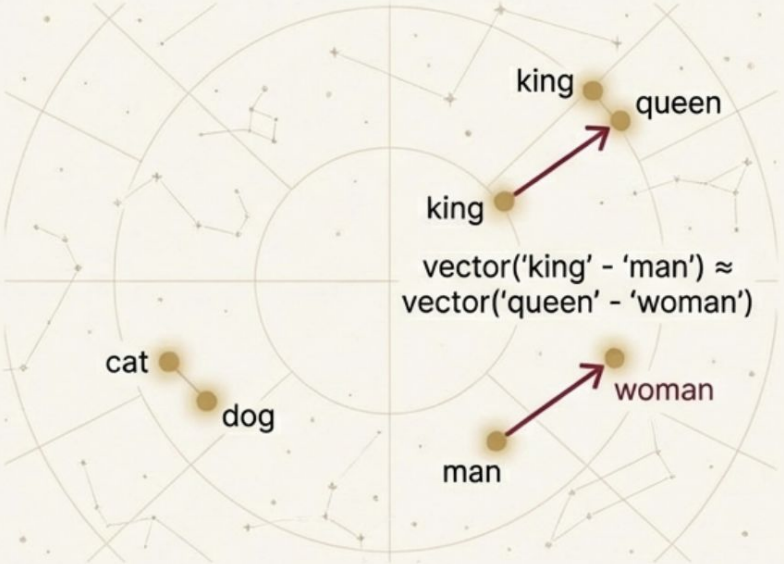
1. Associate each word with a **distributed word feature vector** (a point in a multi-dimensional space).
2. Express the probability of a word sequence in terms of these vectors.
3. Learn the vectors and the probability function **simultaneously**.

Key Insight: Instead of treating words as unique, unrelated IDs in a giant vocabulary, this approach represents them as locations in a continuous space. The number of features (e.g., 30, 60, or 100) is vastly smaller than the vocabulary size (e.g., 17,000+).

The Old Way: N-grams (Sparse Representation)

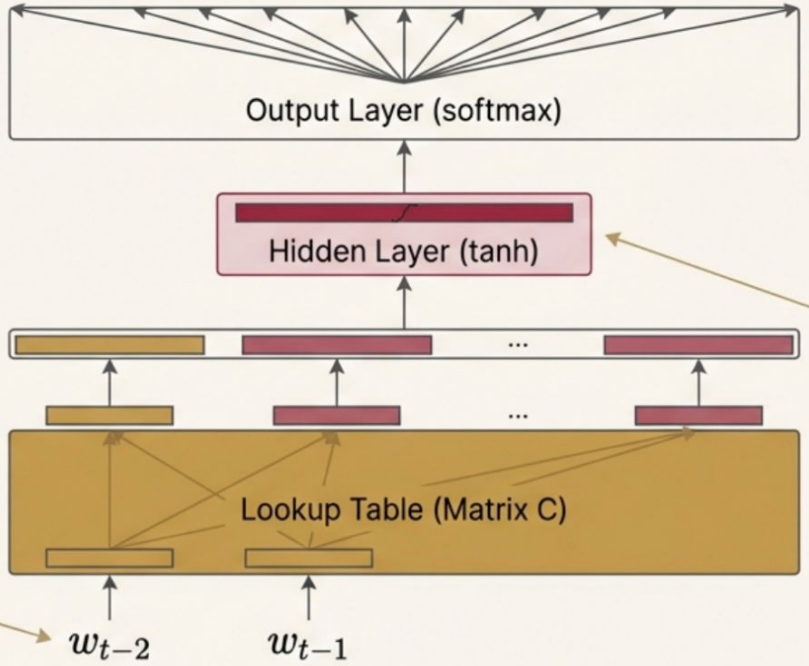
"The cat sat" tells us nothing about "A dog ran."
Words are just symbols.

The New Way: Word Vectors (Distributed Representation)



Seeing "The cat is walking" now informs the model about "A dog was running" because the model knows "cat" and "dog" are similar, and "walking" and "running" are similar. Generalization becomes possible.

The Architecture of the Solution: A Neural Probabilistic Language Model



4. The network outputs a probability distribution over the entire vocabulary for the next word, w_t.

2. Each input word is mapped to its feature vector. This is the "distributed representation" layer. Matrix C is a $|V| \times m$ matrix where these vectors are learned.

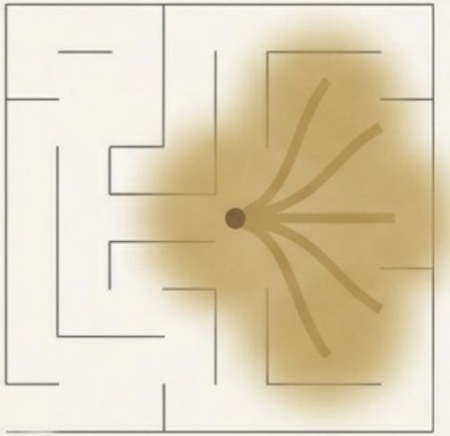
3. The concatenated feature vectors are processed. This layer learns higher-level interactions between the word features.

1. The previous *n-1* words are fed into the model.

The parameters of the model—the word feature vectors (C) and the network weights (H, U, W)—are all learned together to maximize the log-likelihood of the training data.

The Scorekeeper: How Do We Measure if a Model is 'Better'?

High Perplexity
(High Uncertainty)

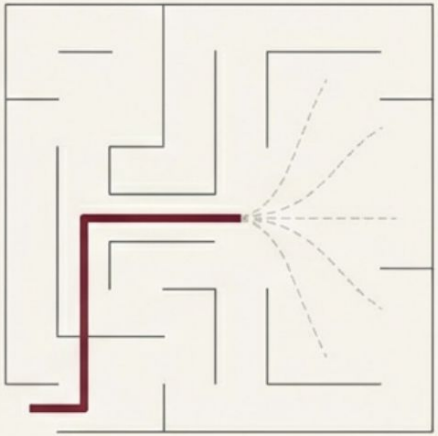


Key Metric: Perplexity (PP).
Lower is better.

The Intuitive Interpretation:
Effective Branching Factor

Perplexity measures how "perplexed" or uncertain a model is when when predicting the next word. A perplexity of 100 means the model's uncertainty is equivalent to choosing uniformly from 100 words at each step.

Low Perplexity
(Low Uncertainty)

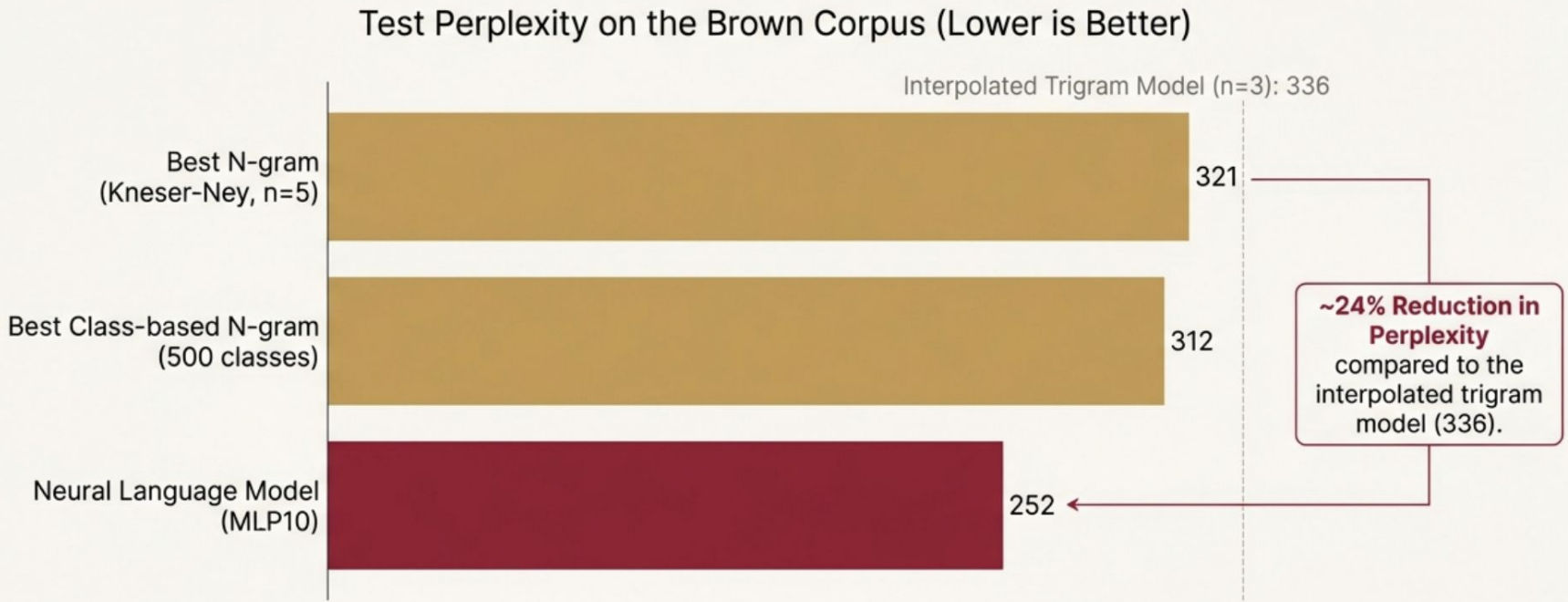


The Mathematical Foundation

Perplexity is derived from **Cross-Entropy**, the average negative log probability the model assigns to each word in a test set: $H(P, Q) = -\frac{1}{N} \sum \log_2(Q(w_i))$

The formula is $\text{Perplexity} = 2^{\text{Cross–Entropy}}$. This is equivalent to the geometric mean of the inverse probabilities of the test set words: $\text{PP}(W) = [\prod 1/P(w_i|\text{context})]^{1/N}$.

The Proof: The Neural Model Was 24% Better Than the State-of-the-Art



The breakthrough was proven, but Bengio's feed-forward network still had limitations.

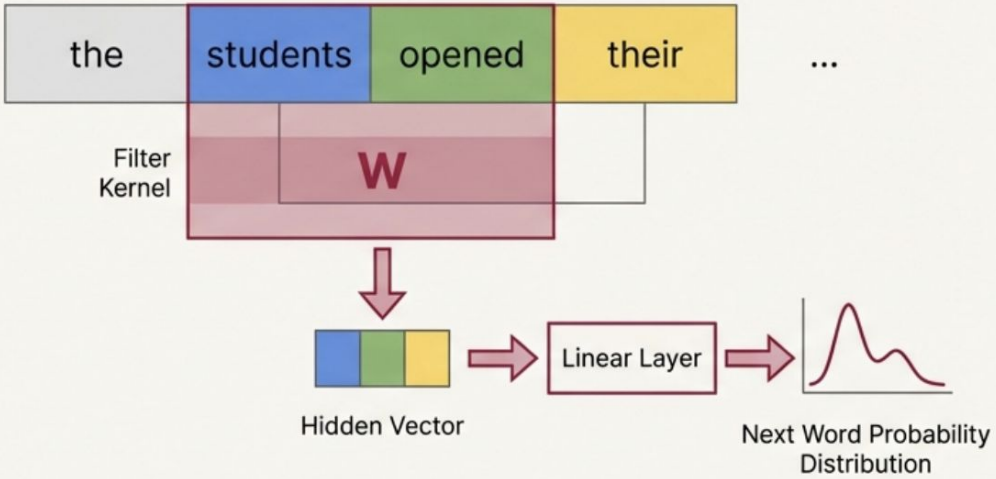
Fixed Context Window

Like n-grams, it could only look at a pre-defined number of previous words (e.g., n=5). Information from words outside this window was lost.

Disjoint Weights

The weights processing the first word in the context were different from the weights processing the second word. There was no shared understanding of position-agnostic features.

The CNN Approach: Applying a ‘Filter’ to a Sequence of Words



How it Works:

1. A sentence is represented as a matrix where each row is a word vector.
2. A “kernel” or “filter” (W) of a fixed size (e.g., 4 words) is applied to a patch of the sentence.
3. This operation produces a **hidden vector**, which is then passed to a linear layer to predict the next word.

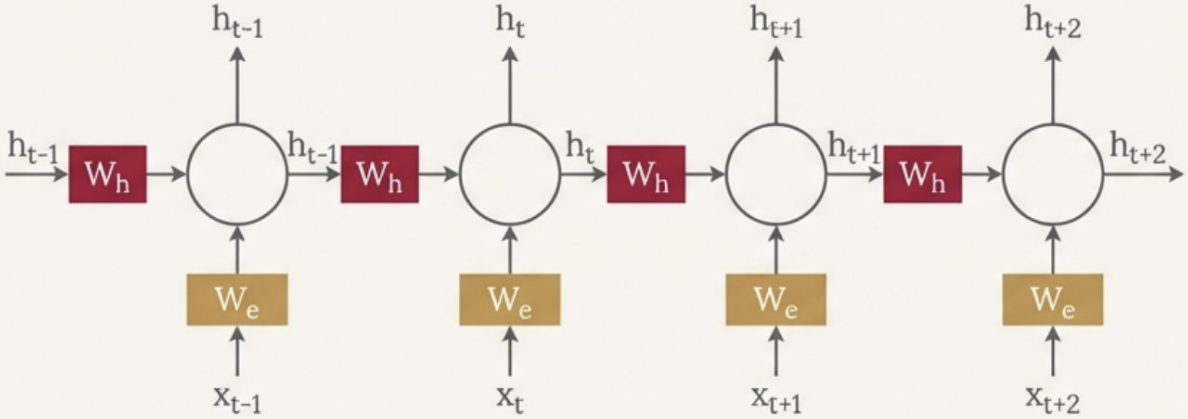
Pros:

- Handles the sparsity problem completely.
- Doesn’t need to store a massive count matrix.

Cons:

- **Still relies on a fixed window size.** Cannot capture long-distance relationships.
- The weight matrix W grows as the window size increases.

The RNN Solution: A Model with Memory



Key Advantages:

1. **Variable-Length Input:** Can theoretically process a context of any length.

2. **Parameter Efficiency:** The weight matrices (W_h and W_e) are ****shared across all time steps****. The model size does not increase with the context length.

****The Core Concept**

A Recurrent Neural Network (RNN) processes a sequence one element at a time, maintaining a hidden state **h** that contains information about the entire history seen so far.

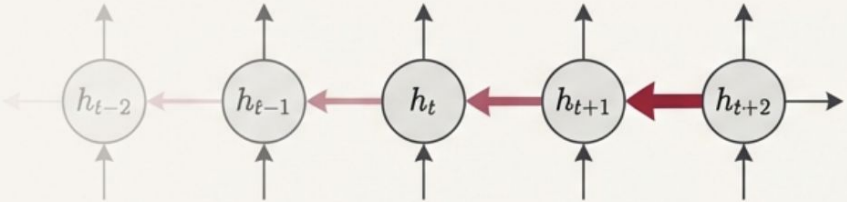
****The Mechanism**

The hidden state at the current time step ($\mathbf{h}_t$) is a function of the previous hidden state ($\mathbf{h}_{t-1}$) and the current input word vector ($\mathbf{x}_t$).

Formula: $\mathbf{h}_t = f(W_h * \mathbf{h}_{t-1} + W_e * \mathbf{x}_t + b)$

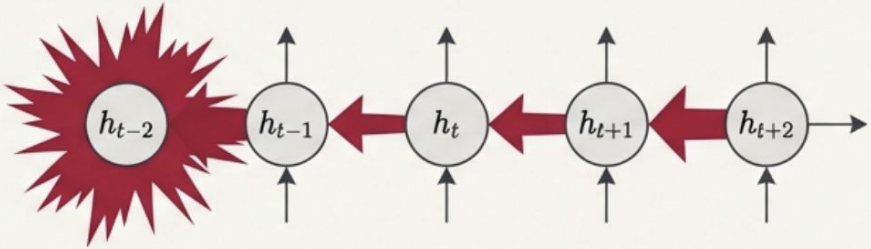
Although RNNs can theoretically remember information from the distant past, in practice, the error signal struggles to propagate through many time steps during training.

Vanishing Gradients



Vanishing Gradients: As the error signal is passed back, it can be repeatedly multiplied by small numbers, causing it to shrink exponentially. The influence of early words "vanishes," and the model fails to learn long-range dependencies.

Exploding Gradients



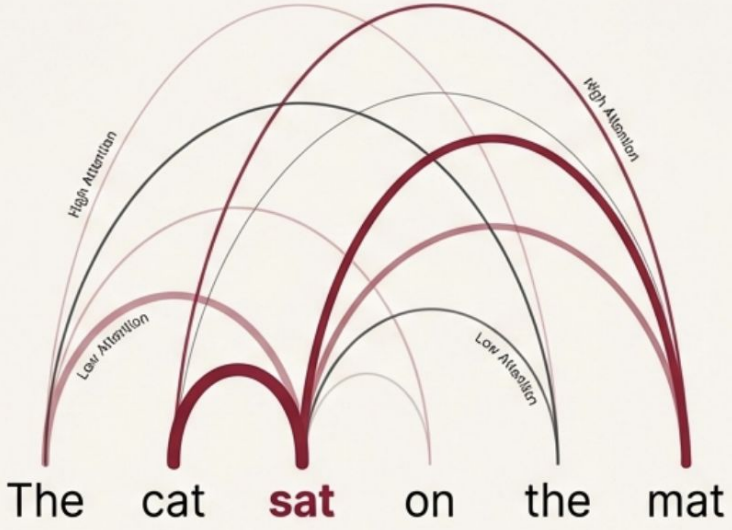
Exploding Gradients: The opposite can also occur. The gradient can be repeatedly multiplied by large numbers, growing exponentially and causing wildly unstable training updates. (This is more easily solved with gradient clipping).

A simple RNN's effective memory is much shorter than the sequence length it theoretically possesses.

The Revolution of ‘Attention is All You Need’ (2017): The Transformer dispensed with recurrence and convolutions entirely.

The Core Mechanism

Self-Attention: Instead of processing words sequentially, self-attention allows the model to directly weigh the influence of every other word in the input sequence when encoding a specific word. It creates a rich, context-aware representation by modeling all pairwise interactions simultaneously.



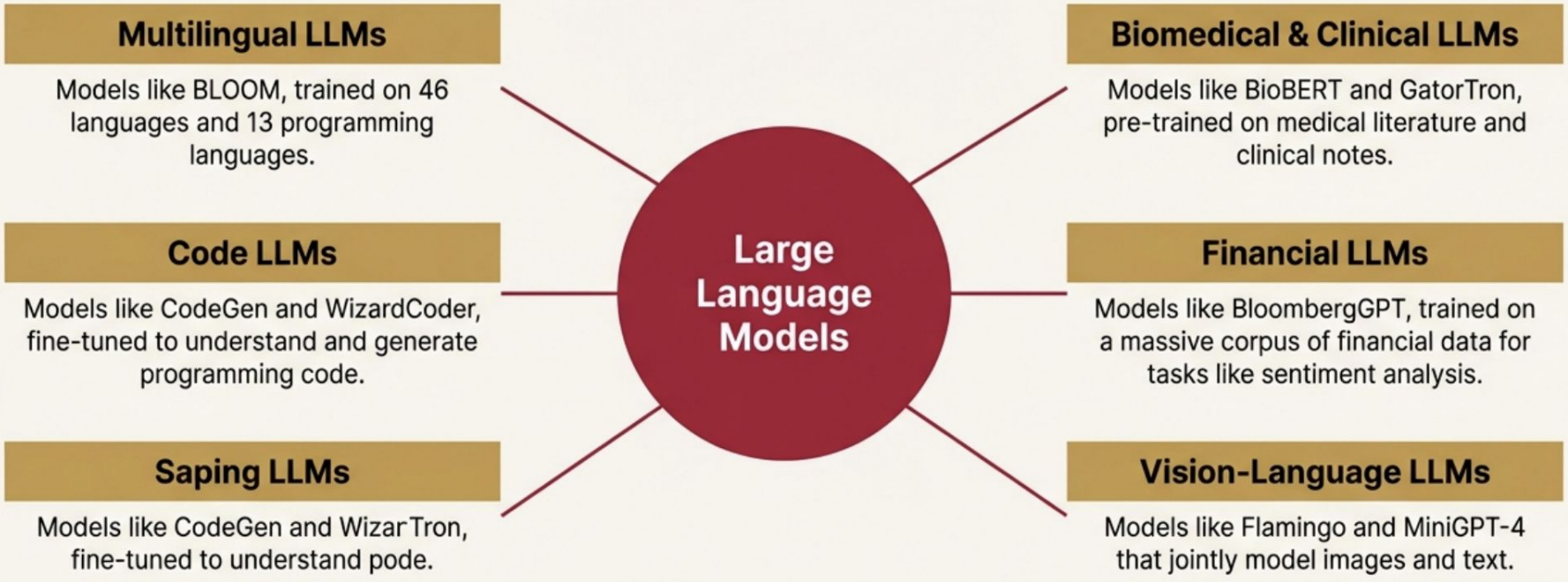
Why it Succeeded

Solved Long-Range Dependencies: The path between any two words is constant, removing the vanishing gradient problem.

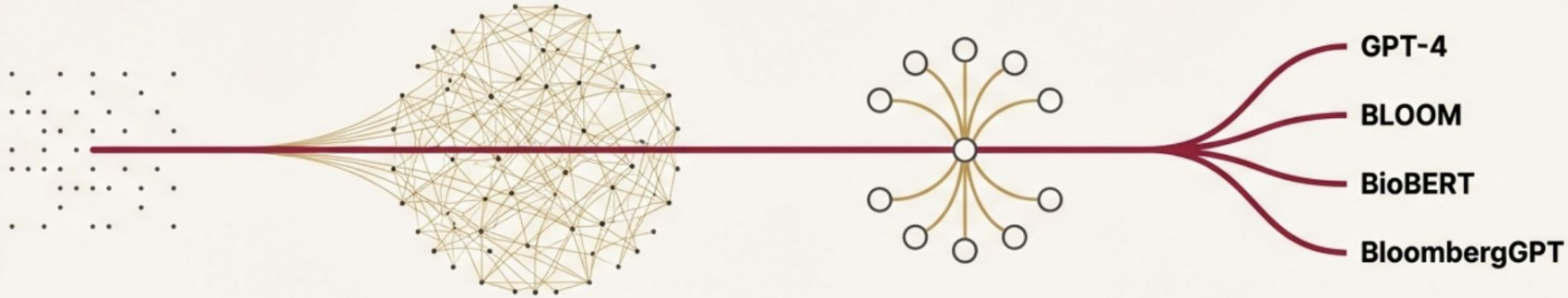
Parallelizable: Since computations aren't sequential, the entire sequence can be processed in parallel, enabling the training of vastly larger models.

The Result: An Explosion of Specialized Large Language Models

The Transformer architecture, built upon the concept of distributed word representations, unlocked the ability to create today's LLMs. This has led to a diverse ecosystem of models specialized for a variety of domains.



The Unbroken Thread: From Vector Space to the Future of AI



The Problem: Curse of Dimensionality	The Insight: Distributed Representations	The Vehicle: Transformer Architecture	The Result: Modern LLMs

The Core Journey Recapped:
- The Problem: The Curse of Dimensionality made language intractable for models that saw words as isolated symbols.
- **The Insight:** Representing words as meaningful vectors in a continuous space allowed models to generalize.
- **The Vehicle:** The Transformer architecture perfected the use of this vector space, enabling models to capture context at an unprecedented scale.

Final Takeaway: Every Large Language Model today stands on the foundation laid by the fight against the curse of dimensionality. The idea of learning a distributed representation for words remains the conceptual key that unlocked modern NLP.