



CS 394E: Contemporary Topics in LLMs

Lecture-4: Neural Language Models

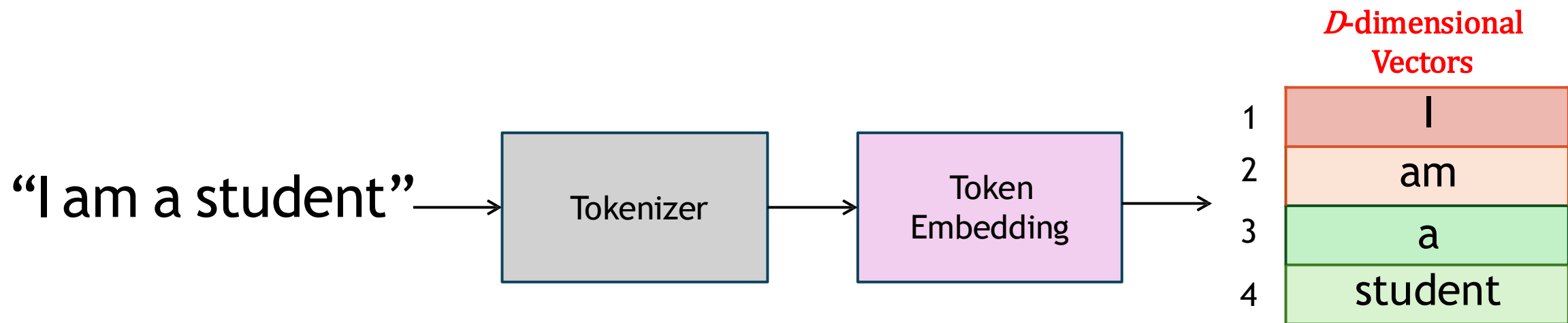
Marco Canini

Professor of Computer Science, KAUST

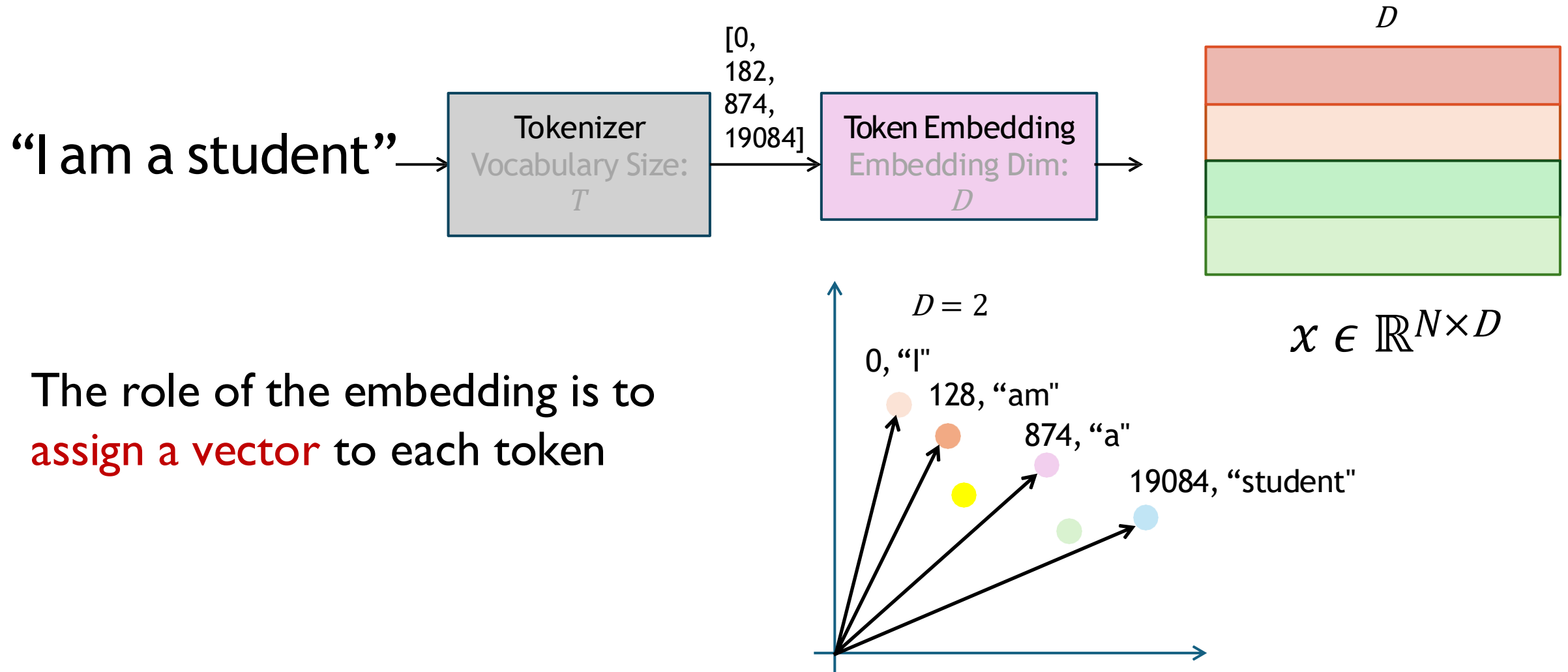
Agenda

- Recap
- Learning Vector Embeddings
- Neural Language Models

Recap: Language to Tokens to Embeddings



Recap: Language to Tokens to Embeddings



Recap: Dense Vector Representations

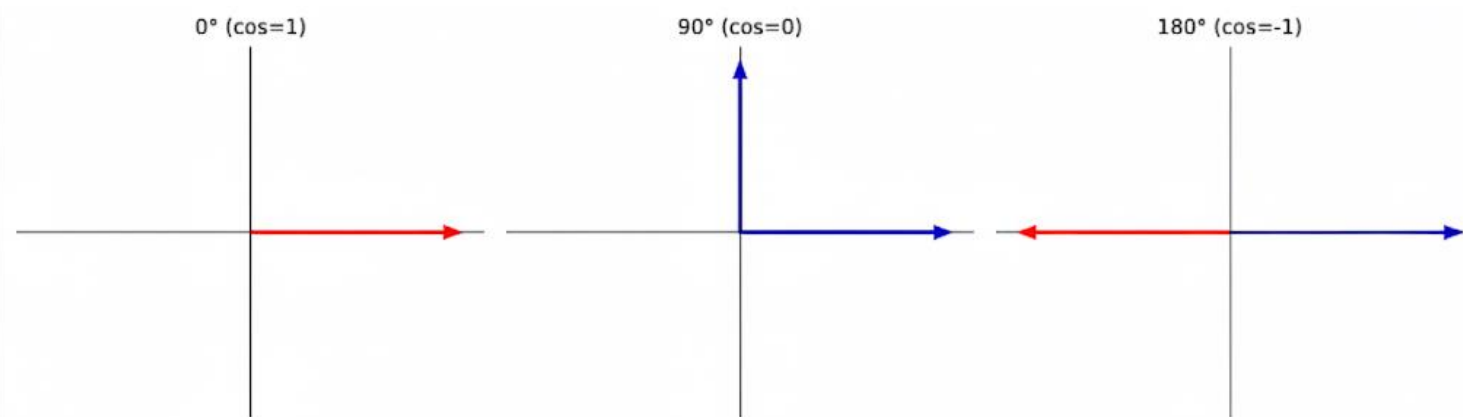
- **Vector Embeddings Solution:** Map discrete words to a continuous vector space where semantic similarity = spatial proximity
 - Objects with similar meanings are embedded close together
 - Enables the model to capture semantic relationships and generalize better to unseen data
- **Examples**
 - “cat” $\rightarrow$ [0.2, -0.4, 0.7, ...]
 - “dog” $\rightarrow$ [0.3, -0.3, 0.8, ...] $\leftarrow$ Similar!
 - “car” $\rightarrow$ [-0.5, 0.9, -0.1, ...]
 - “tree” $\rightarrow$ [0.1, 0.6, -0.8, ...]
- **Why dense embeddings are powerful?**
 - Compact representation (few hundred to a few thousand dimensions)
 - Semantic similarity preserved
 - Arithmetic properties

Recap: Cosine Similarity between Word Vectors

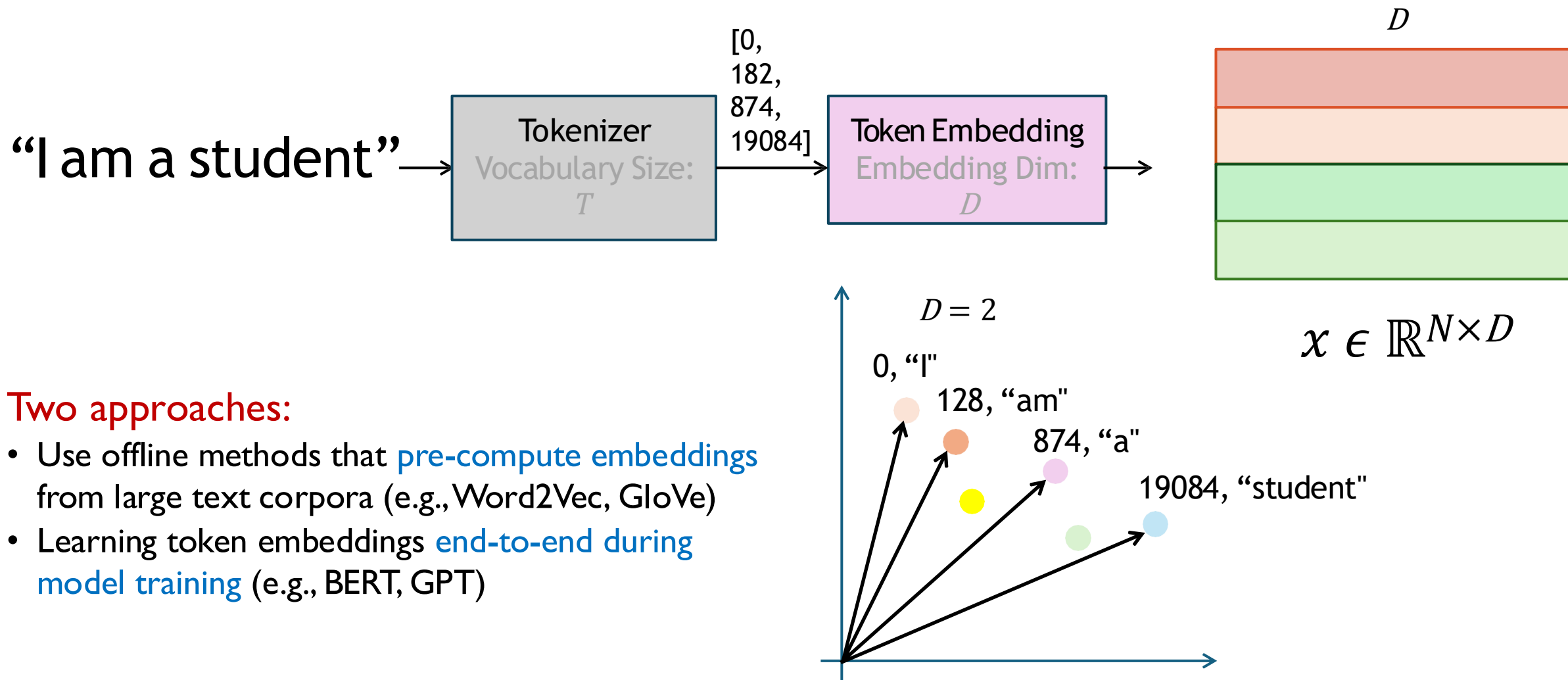
- Use cosine similarity between vectors
 - Smaller angles = higher similarity = vectors closer in embedding space.

$$\text{cosine similarity} = \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|} = \cos(\theta)$$

θ	Cosine Similarity	Meaning
0°	1	Perfectly aligned (same meaning)
90°	0	Unrelated
180°	-1	Opposite meaning



Language to Embeddings



Word2Vec Training

- Word2Vec creates **word embeddings** by training a model on a large text corpus to predict words. It uses two primary methods to do this:
 - **Continuous Bag-of-Words (CBOW)**
 - This model predicts a target word using its surrounding context words
 - It's like filling in the blank. Given “The ___ is wearing a crown”, the model learns to predict “king”
 - **Skip-gram**
 - This model does the reverse: it uses a target word to predict its surrounding context words
 - It's like guessing the words around a keyword. Given “king”, the model learns to predict words like “The”, “is”, “wearing”, and “crown”
- Both methods improve their predictions by continuously updating the word vectors to reduce errors

Example: Context Words and Target Word

- Consider the sentence “The king is wearing a crown,” and assume the target word is king, and the context words are the ones surrounding it
 - Let's assume the context window is 2 (1 word to the left and 2 words to the right of the target word)

Context (input): ["The", "is", "wearing"]

Target (output): "king"

- Suppose the embeddings of these words are:
 - The = [0.12, 0.34, -0.45]
 - king = [-0.78, 0.56, 0.10]
 - is = [0.23, -0.47, 0.75]
 - wearing = [-0.65, 0.43, -0.22]
 - a = [0.19, -0.29, 0.31]
 - crown = [-0.34, 0.22, 0.58]

CBOW Model: Context Averaging

- In the CBOW model, the embeddings of the context words are **averaged to form a context vector**
 - The model takes the embeddings of the context words ["The", "is", "wearing"], sums them up, and divides by the number of context words

1
"The": [0.12, 0.34, -0.45]

"is": [0.23, -0.47, 0.75]

"wearing": [-0.65, 0.43, -0.22]

2
$$\text{Combined Context Vector} = \frac{1}{3} ([0.12, 0.34, -0.45] + [0.23, -0.47, 0.75] + [-0.65, 0.43, -0.22])$$

3
$$\text{Combined Context Vector} = \frac{1}{3} ([-0.30, 0.30, 0.08]) = [-0.10, 0.10, 0.03]$$

Cosine similarity: $\cos(\theta) = 0.137 / (0.1445 * 0.9655) \approx 0.137 / 0.1395 \approx 0.9820$

During training, the model adjusts the embeddings so that the combined context vector becomes closer to the actual embedding of "king", which is initially **[-0.78, 0.56, 0.10]**.

"The": [0.05, 0.32, -0.40]

"is": [0.28, -0.45, 0.70]

4
"wearing": [-0.60, 0.40, -0.25]

After training, this combined context vector **[-0.09, 0.09, 0.02]** will be much closer to the trained embedding of **"king" [-0.75, 0.60, 0.12]**, allowing the model to correctly predict "king" given the context words.

How Close Is the Context Vector to "king"?

- Cosine similarity measures how **aligned** two vectors are (I = same direction)
 - $\mathbf{c} = [-0.10, 0.10, 0.03]$ — the combined context vector, i.e. the average of "The", "is", "wearing"
 - $\mathbf{k} = [-0.78, 0.56, 0.10]$ — the *initial* embedding of the target word "king"
- $\cos(\theta) = (\mathbf{c} \cdot \mathbf{k}) / (||\mathbf{c}|| \times ||\mathbf{k}||)$

STEP 1 — dot product $\mathbf{c} \cdot \mathbf{k}$

$$(-0.10)(-0.78) + (0.10)(0.56) + (0.03)(0.10) = 0.078 + 0.056 + 0.003 = \mathbf{0.137}$$

STEP 2 — vector lengths $||\mathbf{c}||$ and $||\mathbf{k}||$

$$||\mathbf{c}|| = \sqrt{(0.01 + 0.01 + 0.0009)} = 0.1445 \quad ||\mathbf{k}|| = \sqrt{(0.6084 + 0.3136 + 0.01)} = \mathbf{0.9655}$$

STEP 3 — divide

$$\cos(\theta) = 0.137 / (0.1445 \times 0.9655) = 0.137 / 0.1395 \approx \mathbf{0.9820}$$

0.9820 is the similarity BEFORE training. After training, $\mathbf{c} = [-0.09, 0.09, 0.02]$ and $\mathbf{k} = [-0.75, 0.60, 0.12]$, giving $\cos(\theta) \approx 0.9936$ — the vectors have moved closer.

Predicting the Target Word

- Now that we have the **context vector** $[-0.09, 0.09, 0.02]$, the CBOW model uses this to predict the target word, **king** $[-0.78, 0.56, 0.10]$
 - Model tries to find the word whose embedding is most similar to the context vector
 - In this case, we compare the cosine similarity between the context vector and the embedding of the word king

$$\text{cosine similarity} = \frac{c \cdot k}{\|c\| \cdot \|k\|}$$

c is the **context vector**

k is the **target word's embedding**

$c \cdot k$ is the dot product

$\|c\|$ and $\|k\|$ are the vector norms

$$\text{Cosine similarity} = \frac{0.1226}{0.1288 \cdot 0.9655} \approx 0.9863$$

So, the **cosine similarity** between the context vector and the word king is **0.9863**.

Interpretation: This is a high similarity value, indicating that the context words ["The", "is", "wearing"] provide a strong contextual match to the word king.

In this case, the CBOW model would predict that king is the most likely word for the given context.

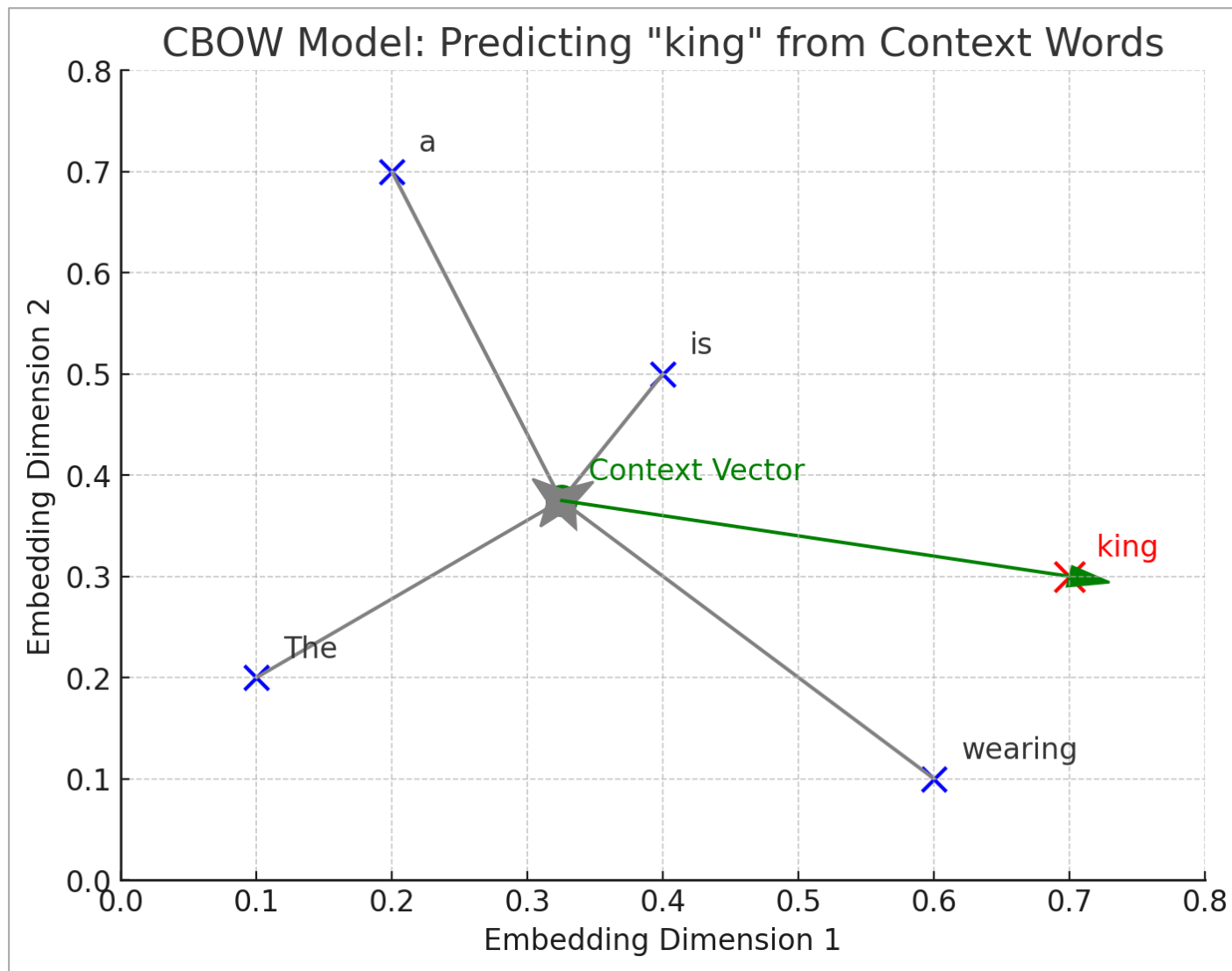
What Happens During Training?

- During training, the model **adjusts the word embeddings using gradient descent**
 - The objective is to **increase the cosine similarity** between the context vector and the actual target word, while **minimizing similarity with incorrect words**
 - Over time, embeddings for words that frequently appear together (like king and crown) will move closer in the embedding space, **reducing the angle between their vectors and increasing cosine similarity**

Mechanism Overview

- Goal: How do we move word vectors so that context words become better at predicting the target word?
- Mechanism
 - Build a context vector by averaging word embeddings
 - Score how well this context predicts the target word
 - Compute an error
 - Use gradient descent to update both:
 - Context word embeddings
 - Target word embeddings

Visualization



CBOW Review

The **king** is wearing a crown

Initial Vectors

Starting word vectors

The: [0.10, 0.20, 0.30]

king: [0.40, 0.50,
0.60]

is: [0.70, 0.80, 0.90]

wearing: [0.20, 0.30,
0.40]

a: [0.50, 0.60, 0.70]

crown: [0.80, 0.90,
1.00]

CBOW Review

The **king** is wearing a crown

Select Context

Choose words within the context window

The: [0.10, 0.20, 0.30]

king: [0.40, 0.50,
0.60]

is: [0.70, 0.80, 0.90]

wearing: [0.20, 0.30,
0.40]

a: [0.50, 0.60, 0.70]

crown: [0.80, 0.90,
1.00]

CBOW Review

The **king** is wearing a crown

Create Context Vector

Average the vectors of context words

The: [0.10, 0.20, 0.30]

king: [0.40, 0.50, 0.60]

is: [0.70, 0.80, 0.90]

wearing: [0.20, 0.30, 0.40]

a: [0.50, 0.60, 0.70]

crown: [0.80, 0.90, 1.00]

Context Vector:
[0.33, 0.43, 0.53]

CBOW Review

Predict Target

Use context vector to predict the target word

The: [0.10, 0.20, 0.30]

king: [0.40, 0.50, 0.60]

is: [0.70, 0.80, 0.90]

wearing: [0.20, 0.30, 0.40]

a: [0.50, 0.60, 0.70]

crown: [0.80, 0.90, 1.00]

Context Vector:

[0.33, 0.43, 0.53]

Prediction:

0.6615

Formula: $\sigma(\text{context_vector} \cdot \text{target_vector})$

CBOW Review

Prediction:

0.6615

Formula: $\sigma(\text{context_vector} \cdot \text{target_vector})$

Error:

0.3385

Formula: $\text{error} = \text{target} - \text{prediction}$

*The gradient vector that indicates how much the model's parameters should be adjusted to reduce the error during training. By multiplying the error with the context vector, the gradient provides the **direction** and **magnitude** of adjustment for the weights.*

Context Vector:

[0.33, 0.43, 0.53]

Prediction:

0.6615

Formula: $\sigma(\text{context_vector} \cdot \text{target_vector})$

Error:

0.3385

Formula: $\text{error} = \text{target} - \text{prediction}$

Gradient:

[0.11, 0.15, 0.18]

Formula: $\text{gradient} = \text{error} * \text{context_vector}$

CBOW Review

Context Vector:

[0.33, 0.43, 0.53]

Prediction:

0.6615

Formula: $\sigma(\text{context_vector} \cdot \text{target_vector})$

Error:

0.3385

Formula: $\text{error} = \text{target} - \text{prediction}$

Gradient:

[0.11, 0.15, 0.18]

Formula: $\text{gradient} = \text{error} * \text{context_vector}$

Update Formulas:

Context words: $v_{\text{context}} = v_{\text{context}} + \text{learning_rate} * \text{gradient}$

Target word: $v_{\text{target}} = v_{\text{target}} + \text{learning_rate} * \text{error} * \text{context_vector}$

This update formula adjusts the context word embeddings, so they become more predictive of the target word over time.

*The target word embedding is also updated by adjusting it in the **direction of the context vector** and **scaled by the error**.*

This process is repeated for every word in the corpus.

What are the limitations of the CBOW Model?

- **Unweighted averaging** → all words are equally important
- **Fixed context window** → challenges with capturing long-range dependence

What if we make the context window infinite?

- **Increased computational complexity**
 - Model would need to consider every other word in the corpus for each target word
 - This would dramatically increase the computational cost, making training
- **Loss of local context**
 - CBOW relies on the assumption that nearby words are more relevant for predicting a target word
 - An infinite window would give equal weight to all words, regardless of their distance from the target

Neural Language Model

Language Model $\rightarrow$ Joint Probability Distribution

- Language Models aim to find the **joint probability**

$$P(W) = P(w_1, w_2, \dots, w_n)$$



$$P(W) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdot \dots \cdot P(w_n | w_1, \dots, w_{n-1})$$



N-gram

$$P(w_n | w_1, \dots, w_{n-1}) \approx P(w_n | w_{n-N+1}, \dots, w_{n-1})$$

Curse of Dimensionality

- A very large number of word sequences given a realistic vocabulary
- A word sequence on which the model is tested is likely to be **different from word sequences during training**
- **Generalization** on test sets is hard!

Neural Language Models

- Bengio et al., 2003 pioneered **neural language models**, where the probability distribution $P(w_n | w_1, \dots, w_{n-1})$ is given by a neural network
 - NLMs can better capture **long-range dependencies** and **semantic relationships**

Journal of Machine Learning Research 3 (2003) 1137–1155

Submitted 4/02; Published 2/03

A Neural Probabilistic Language Model

Yoshua Bengio
Réjean Ducharme
Pascal Vincent
Christian Jauvin

*Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada*

BENGIOY@IRO.UMONTREAL.CA
DUCHARME@IRO.UMONTREAL.CA
VINCENTP@IRO.UMONTREAL.CA
JAUVINC@IRO.UMONTREAL.CA



Key Idea

- Instead of **estimating raw probabilities**, let's use a neural network to fit the probability distribution of language!
 - $P(w \mid \text{I am a good})$
 - $P(w \mid \text{I am a great})$
- Key ingredient: word embeddings $e(\text{good}) \approx e(\text{great})$

Fighting the Curse of Dimensionality

- Learning a **distributed representation of words**
 - Allows each training sentence to inform the model about an exponential number of semantically neighboring sentences
- The models **jointly** learns:
 1. Distributed representation of words, i.e., vector embeddings
 2. **Probability function** for word sequences expressed in terms of these representations

Fighting the Curse Dimensionality

1. associate with each word in the vocabulary a distributed *word feature vector* (a real-valued vector in $\mathbb{R}^m$),
2. express the joint *probability function* of word sequences in terms of the feature vectors of these words in the sequence, and
3. learn simultaneously the *word feature vectors* and the parameters of that *probability function*.

A fundamental problem that makes language modeling and other learning problems difficult is the *curse of dimensionality*. It is particularly obvious in the case when one wants to model the joint distribution between many discrete random variables (such as words in a sentence, or discrete attributes in a data-mining task). For example, if one wants to model the joint distribution of 10 consecutive words in a natural language with a vocabulary V of size 100,000, there are potentially $100\,000^{10} - 1 = 10^{50} - 1$ free parameters. When modeling continuous variables, we obtain generalization more easily (e.g. with smooth classes of functions like multi-layer neural networks or Gaussian mixture models) because the function to be learned can be expected to have some local smoothness properties. For discrete spaces, the generalization structure is not as obvious: any change of these discrete variables may have a drastic impact on the value of the function to be esti-

The Case for Better Generalization

A useful way to visualize how different learning algorithms generalize, inspired from the view of non-parametric density estimation, is to think of how probability mass that is initially concentrated on the training points (e.g., training sentences) is distributed in a larger volume, usually in some form of neighborhood around the training points. In high dimensions, it is crucial to distribute probability mass where it matters rather than uniformly in all directions around each training point. We will show in this paper that the way in which the approach proposed here generalizes is fundamentally different from the way in which previous state-of-the-art statistical language modeling approaches are generalizing.

Components of the NLM Approach

I) Word Embeddings

- Words represented as continuous-valued vectors instead of discrete tokens
 - Embeddings enable generalization: similar words get similar representations
- Joint learning of word probabilities & embeddings during training
- Advantage over N-grams: Generalizes to unseen sequences through word similarity

second it is not taking into account the “similarity” between words. For example, having seen the sentence “The cat is walking in the bedroom” in the training corpus should help us generalize to make the sentence “A dog was running in a room” almost as likely, simply because “dog” and “cat” (resp. “the” and “a”, “room” and “bedroom”, etc...) have similar semantic and grammatical roles.

Components of the NLM Approach by Bengio et al.

2) Input Layer

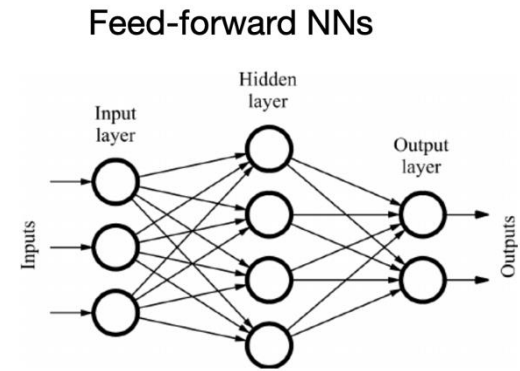
- Word embeddings ($w_1, w_2, \dots, w_{n-1}$) concatenated into single vector
- Transforms discrete words into continuous representation

3) Hidden Layer

- These layers apply **non-linear transformations**, enabling the model to capture complex patterns and dependencies in the data

4) Output Layer

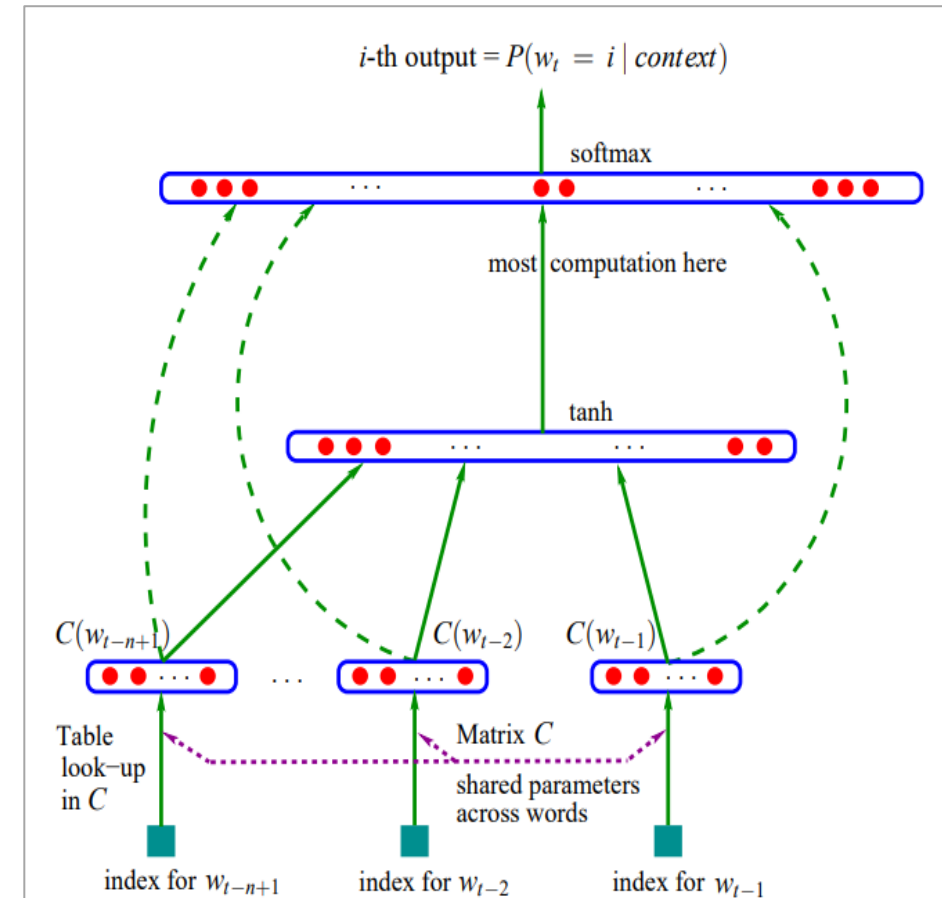
- Softmax activation generates **probability distribution**
 - Predicts next word w_n from entire vocabulary; Outputs sum to 1
- First successful neural approach to language modeling using distributed word representations



Components of the NLM Approach

5) Training

- Minimize **cross-entropy loss** between predicted and actual next words
- **Joint optimization** of word embeddings and network weights
- **Objective**: Maximize likelihood of training sequences

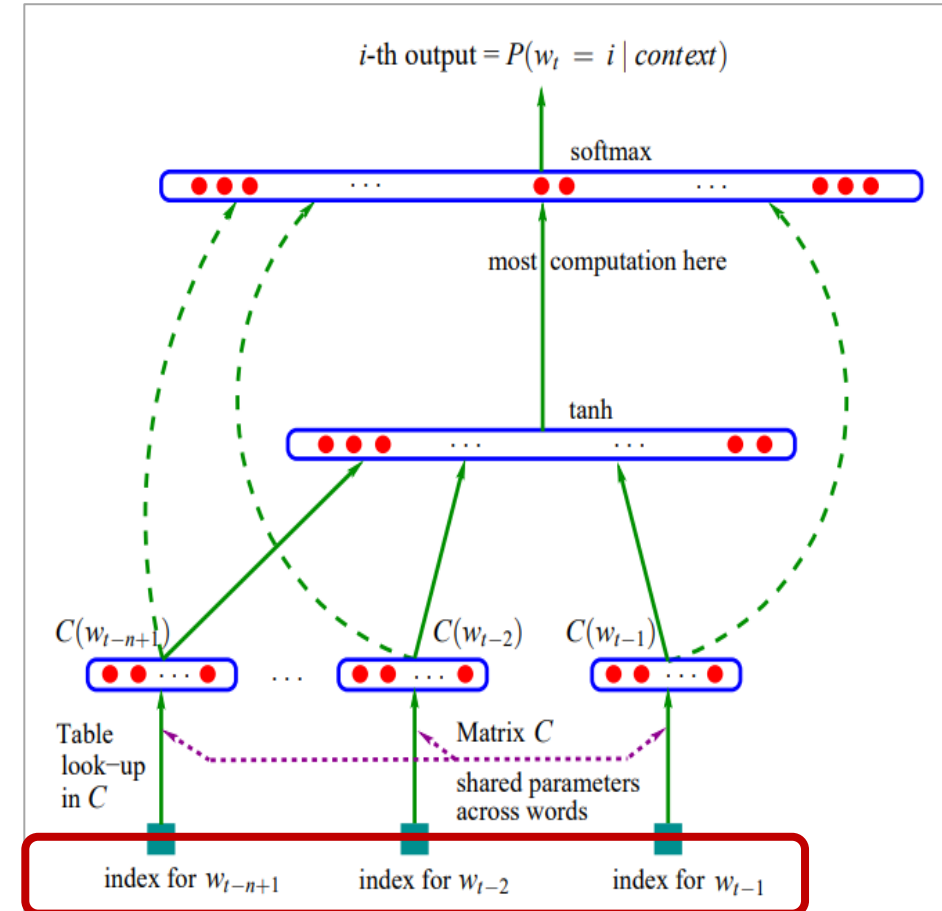


Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

Components of the NLM Approach

5) Training

- Minimize **cross-entropy loss** between predicted and actual next words
- **Joint optimization** of word embeddings and network weights
- **Objective**: Maximize likelihood of training sequences

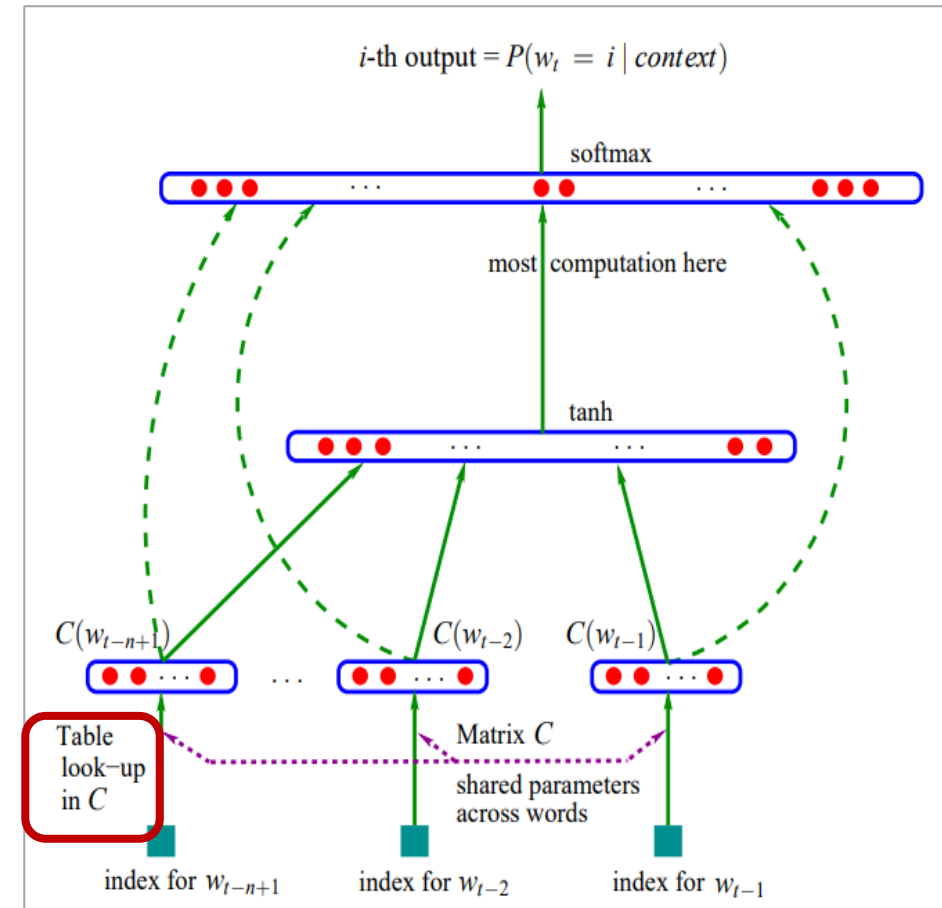


Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

Components of the NLM Approach

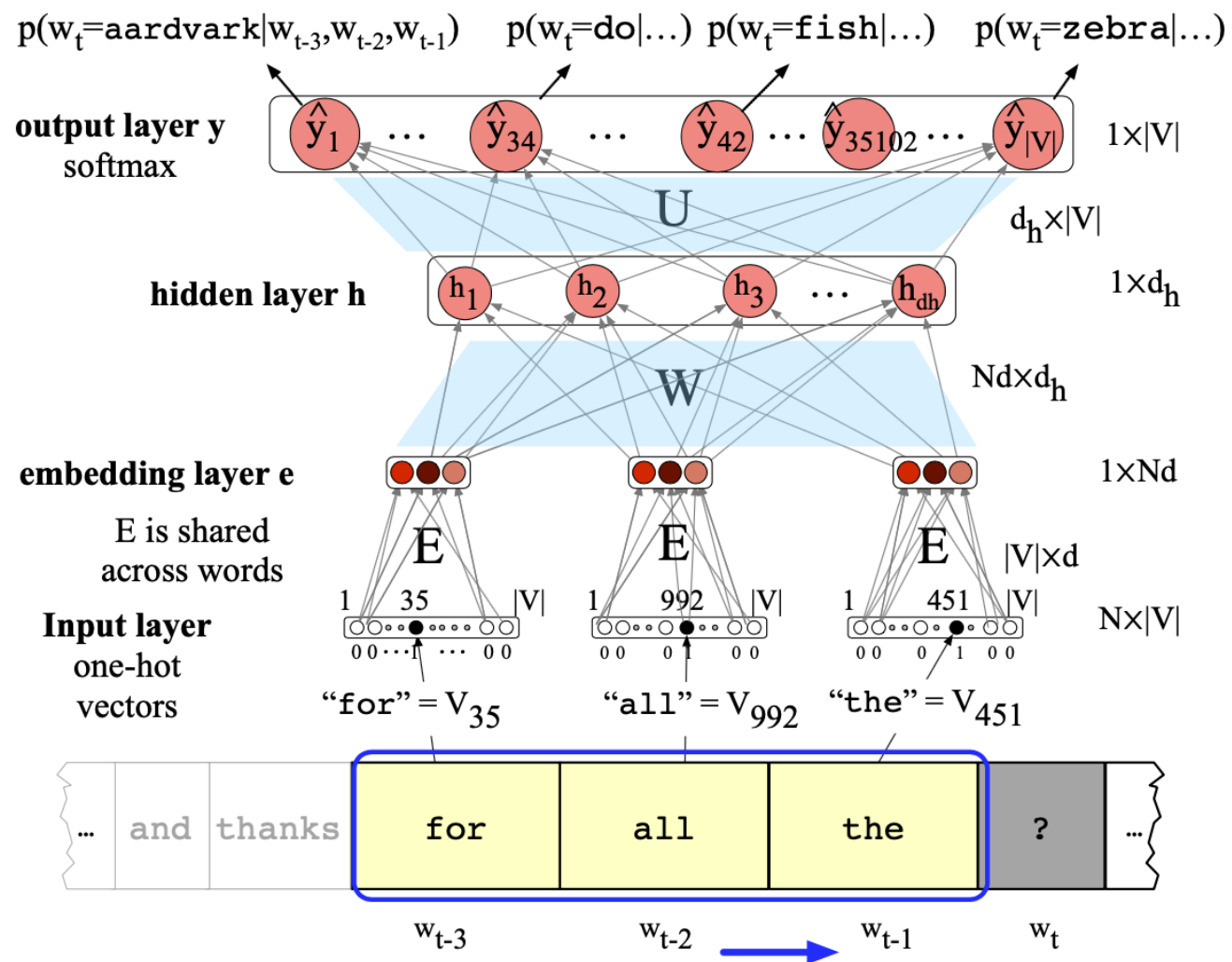
5) Training

- Minimize **cross-entropy loss** between predicted and actual next words
- **Joint optimization** of word embeddings and network weights
- **Objective**: Maximize likelihood of training sequences



Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

Inference in an NLM



$$\mathbf{e} = [\mathbf{Ex}_{t-3}; \mathbf{Ex}_{t-2}; \mathbf{Ex}_{t-1}]$$

$$\mathbf{h} = \sigma(\mathbf{We} + \mathbf{b})$$

$$\mathbf{z} = \mathbf{Uh}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

Source: "Speech and Language Processing", Book by Dan Jurafsky and James Martin

Putting it Together

Feedforward neural language models approximate the probability based on the previous m (e.g., 5) words - m is a hyper-parameter!

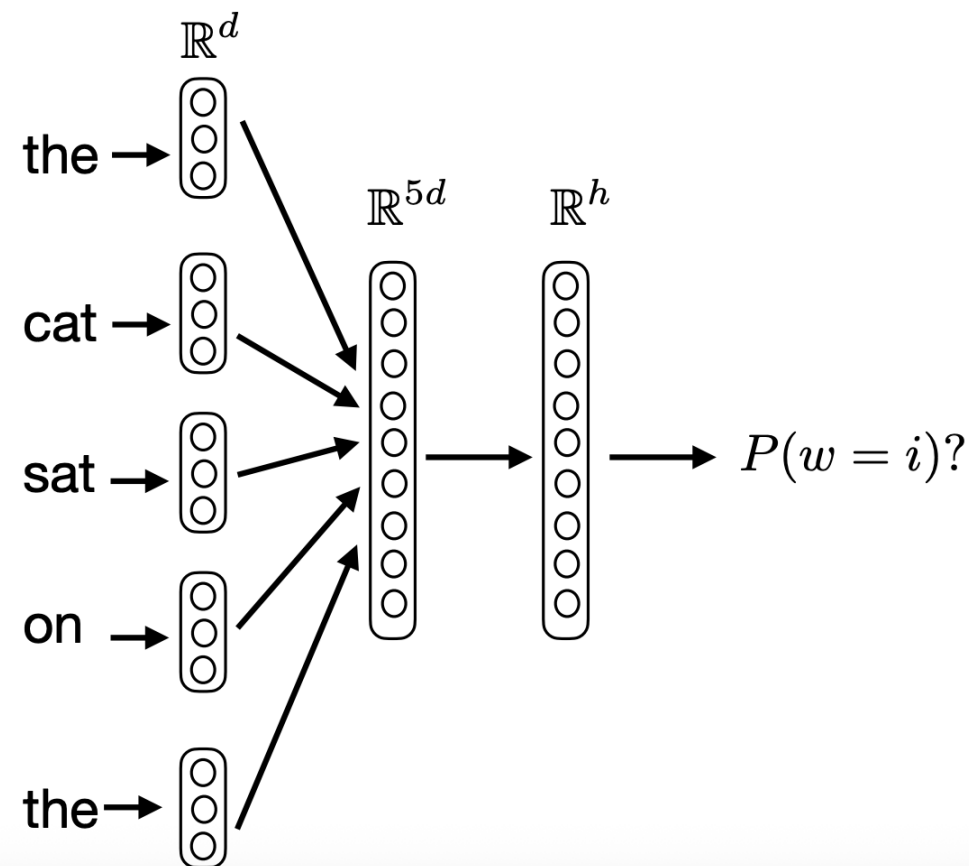
$$P(x_1, x_2, \dots, x_n) \approx \prod_{i=1}^n P(x_i \mid x_{i-m+1}, \dots, x_{i-1})$$

$P(\text{mat} \mid \text{the cat sat on the}) = ?$

d : word embedding size

h : hidden size

It is a $|V|$ -way classification problem!



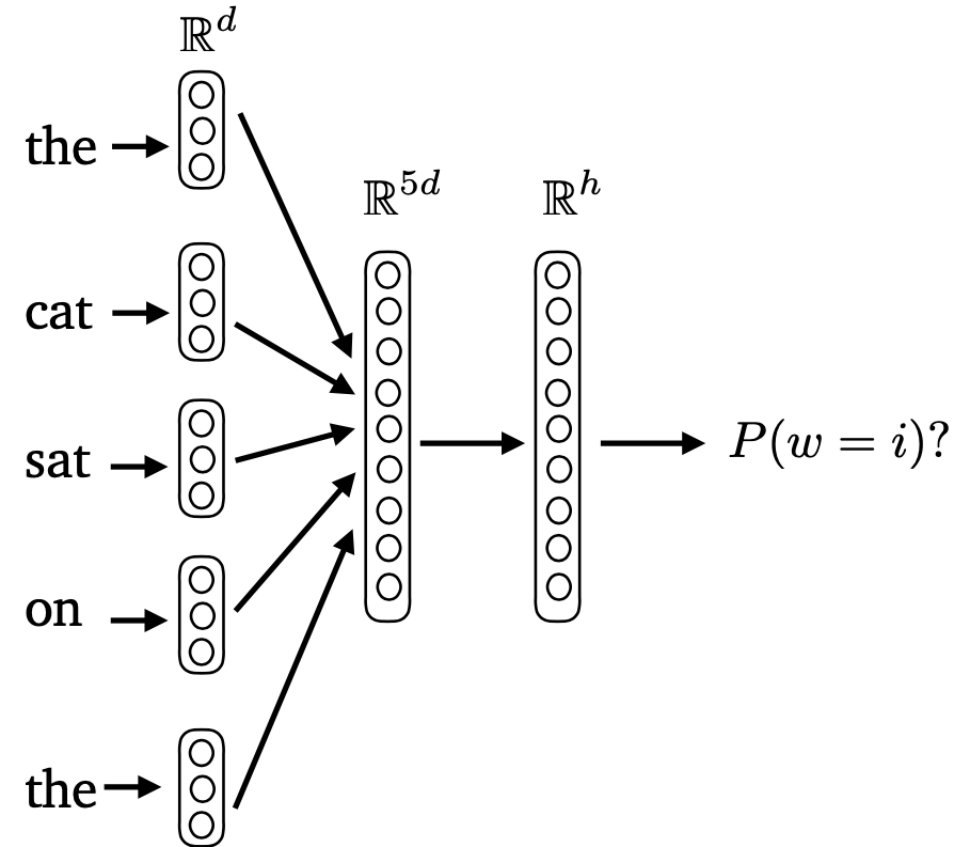
Example

$P(\text{mat} \mid \text{the cat sat on the}) = ?$ d : word embedding size h : hidden size

- Input layer ($m=5$):
 $\mathbf{x} = [e(\text{the}); e(\text{cat}); e(\text{sat}); e(\text{on}); e(\text{the})] \in \mathbb{R}^{md}$
- Hidden layer:
 $\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^h$
- Output layer $\mathbf{U}$ is the **output layer weight matrix**
 $\mathbf{z} = \mathbf{U}\mathbf{h} \in \mathbb{R}^{|V|}$

$$P(w = i \mid \text{the cat sat on the})$$

$$= \text{softmax}_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$



Why Neural LMs Generalize Better than N-grams?

- N-grams treat contexts as discrete symbols → no similarity sharing
- Neural LMs embed words in a continuous space
 - Similar words & contexts → similar representations
 - Parameters are shared across all contexts in
 - Enables generalization to unseen n-grams

The Paper's Impact...

- The 2003 paper by Bengio et al. introduced the concept of **learnable word embeddings**, which became a cornerstone in many NLP applications
- While the idea of distributed word representations existed before but they were among the first to formalize and successfully apply them within NLMs
- Later models, such as Word2Vec, GloVe, and Transformer-based models like BERT and GPT, all build upon this

Pros & Cons of early NLMs

- Advantages

- Better generalization: Predicts unseen word sequences via learned semantic patterns
- Semantic similarity: Words with similar meanings have similar vector representations
- Vocabulary scaling: Dense embeddings handle large vocabularies more efficiently than sparse N-gram approaches

- Challenges

- Computational cost: Slow training from large # of parameters & full-vocabulary softmax
- Context limitations: Fixed input windows restrict long-range dependency modeling
- Reduced interpretability: Less transparent than N-gram probability tables

Let's step back and ask the following

- Are these vector embeddings **static**?
 - In the **simple neural language model** we discussed, the representation of a word's meaning is always the same vector, irrespective of the context
- *Examples*
 - *I sat on the **bank** of the river and watched the water flow*
 - **bank = water bank**
 - *I went to the **bank** to deposit some money*
 - **bank = financial institution**
 - *The plane made a sharp **bank** to the left*
 - **bank = tilt/maneuver**
- **Same token, different senses, but Word2Vec & early NLMs give it one vector**

From Word2Vec/Early NLMs to Attention

CBOW provided us a way to understand a word by looking at its neighbors through a fixed-size window.

Now, what if, instead of

- (i) treating all context words equally, **we could learn which words are more important** for understanding each word?
- (ii) restricting ourselves to a fixed window, we could **attend over the entire sentence?**

Self-attention generalizes this idea by computing a context-dependent representation for each word, using learned, input-dependent weights over all other words in the sequence.

Big Picture: From Word2Vec to Attention

- In CBOW, we **average the vectors of context words**
- In self-attention, we replace this average with a **weighted sum, where the weights (attention scores) are learned**
 - These weights determine how much each word in the sentence contributes to the representation of the current word
- While CBOW asks *'What's the average meaning of the context?'*, self-attention asks, *'How important is each word in the sentence for understanding this particular word?'*
 - The latter allows for more nuanced and context-specific representations, capturing complex relationships between words regardless of their distance in the sentence

Transformer Models: BERT and GPT

- In **transformer models** like BERT and GPT, the context is taken into account through the self-attention mechanism
- **Self-Attention**
 - The attention mechanism allows the model to **focus on different parts of the input** (i.e., context words) when making predictions
 - Each word's embedding is computed **in relation to every other word** in the sequence
 - E: For the sentence “*The king is wearing a crown*,” the word king’s embedding will be computed in relation to **The**, **is**, **wearing**, and **crown**, capturing the full context
- BERT takes **bidirectional context into account**, meaning it looks both to the left and the right of a word to understand its context, ...
 - while **GPT processes context in a unidirectional** (left-to-right) manner

Attention Mechanism

Idea: Identify and attend to what is important in the input.

Questions

Next Lecture

- Attention and Large Language Models