CSCI E-89B: Introduction to Natural Language Processing
Lecture 04: Bag of Words, N-grams, and Convolutional Neural Networks

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-89B: Introduction to Natural Language Processing
■ **Week:**	Lecture 04
■ **Instructor:**	Dmitry Kurochkin
■ **Objective:**	Master bag of words representations, n-gram features, embedding mechanics, and introduction to Convolutional Neural Networks for NLP

Contents

1 Quiz Review: Text Processing Fundamentals

Lecture Overview

This lecture covers key concepts in text representation for NLP, including bag of words, n-grams, and embeddings. We also introduce Convolutional Neural Networks (CNNs) as an alternative to RNNs for capturing local context in text.

1.1 Tokenization

Definition: Tokenization

Tokenization is the process of splitting text into specific units of information called **tokens**. These can be:

- **Words**: Most common choice
- **Subwords**: For handling unknown words and knowledge transfer
- **Characters**: Maximum flexibility, requires more data

When to Use Which Token Type

- **Words**: Best for specific tasks on your own dataset (classification, sentiment)
- **Characters**: Better for knowledge transfer between datasets; more data needed
- **Subwords**: Balance between vocabulary size and handling unknown words

Example: NLTK Smart Tokenization

NLTK recognizes initials and keeps them together:

```
from nltk.tokenize import word_tokenize
text = "J. Wright in 1903"
tokens = word_tokenize(text)
# Result: ['J.', 'Wright', 'in', '1903']
# Note: 'J.' stays together because NLTK recognizes it as an initial!
```

NLTK doesn't just split on spaces and punctuation—it understands linguistic patterns.

1.2 Stemming vs. Lemmatization

Aspect	Stemming	Lemmatization
Definition	Remove prefixes and suffixes	Reduce to dictionary base form
Speed	Very fast (rule-based)	Slower (requires POS tagging)
Output	May not be a real word	Always a real word
Example	"octopus" → "octop"	"octopus" → "octopus"
Use case	Classification, sentiment	Translation, text generation

Stemming Produces Non-Words

The main disadvantage of stemming: it doesn't produce real dictionary words. "Octopuses" becomes "octop," which isn't a word. For tasks like translation or caption generation where you need valid words, use lemmatization.

2 Bag of Words Representation

2.1 What is Bag of Words?

Definition: Bag of Words (BoW)

A text representation where each document is converted to a fixed-length vector. Each position corresponds to a word in the vocabulary, and the value is the word's frequency (or binary presence) in that document.

Key characteristic: Word order is completely discarded—only frequency matters.

2.2 Creating Bag of Words

1. **Tokenize**: Split all documents into tokens
2. **Build vocabulary**: Collect unique tokens from training corpus
3. **Count**: For each document, count occurrences of each vocabulary word
4. **Create vector**: Vector length = vocabulary size

Example: Bag of Words Construction

Corpus:

- Doc 1: "Henry Ford introduced Model T"
- Doc 2: "Ford T was revolutionary"

Vocabulary (order of first occurrence): [Henry, Ford, introduced, Model, T, was, revolutionary]

Vectors:

- Doc 1: [1, 1, 1, 1, 1, 0, 0] (Ford=1, T=1, etc.)
- Doc 2: [0, 1, 0, 0, 1, 1, 1] (no Henry, Ford=1, T=1, was=1)

If "Ford" appeared twice in Doc 2, its value would be 2 (frequency count).

2.3 Bag of Words with sklearn

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "Henry Ford introduced Model T",
    "Ford T was revolutionary"
]

# Create and fit vectorizer
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

# View vocabulary
print(vectorizer.get_feature_names_out())
# ['ford', 'henry', 'introduced', 'model', 'revolutionary', 't', 'was']

```

```
# View bag of words representation
print(X.toarray())
# [[1 1 1 1 0 1 0]
#  [1 0 0 0 1 1 1]]
```

2.4 Handling New Documents

Critical: Vocabulary is Fixed

When processing new documents (test data):

- The vocabulary is **fixed** from training
- New words not in vocabulary become **Out-of-Vocabulary (OOV)**
- OOV words can be ignored or counted in a special OOV position

Never add new words to vocabulary during testing!

```
# Transform new document using trained vectorizer
new_doc = ["The impact of self-driving cars"]
X_new = vectorizer.transform(new_doc)
print(X_new.toarray())
# Mostly zeros because new words aren't in vocabulary!
```

2.5 Limitations of Bag of Words

Critical Limitations

1. **Order is lost**: "dog bites man" = "man bites dog"
2. **Sparse representations**: Large vocabulary ⇒ many zeros
3. **No semantic meaning**: Words are independent features
4. **Storage/computation**: Large vocabularies are expensive

2.6 Applications

Despite limitations, bag of words is effective for:

- **Text classification**: Topic categorization
- **Sentiment analysis**: When word presence matters more than order
- **Document similarity**: Comparing document content
- **Baseline models**: Quick prototyping

When BoW Works Well

Bag of words is surprisingly effective for classification tasks where the mere presence of certain words strongly indicates the class. For example, a movie review containing "boring," "waste," "terrible" is likely negative, regardless of word order.

3 Understanding Embeddings

3.1 The Problem with Sparse Representations

Consider a vocabulary of 10,000 words. With one-hot encoding:

- Each word is a 10,000-dimensional vector
- Only one position is 1, rest are 0
- Enormous storage and computation waste
- No semantic relationships captured

3.2 What Embeddings Really Do

Definition: Embedding

An embedding is a learned mapping from sparse, high-dimensional space to dense, low-dimensional space:

$$\text{Embedding} : \mathbb{R}^V \to \mathbb{R}^d$$

where V = vocabulary size (e.g., 10,000) and d = embedding dimension (e.g., 128).

Example: Geometric Intuition

Consider mapping 2 classes (male/female) to 1D:

- One-hot: female = [1, 0], male = [0, 1] (2D space)
- Embedding: female = 1, male = 0 (1D space)

For 3 classes with one-hot [1,0,0], [0,1,0], [0,0,1]:

- We can map to 1D: class 1 $\to w_1$, class 2 $\to w_2$, class 3 $\to w_3$
- These w values are **learned during training**!

3.3 Why Index-Based Input Works

Key Summary

Key insight: When input is one-hot encoded (e.g., [0, 1, 0, 0]):

- Matrix multiplication with one-hot vector just **selects one row**
- No need to store the full one-hot vector
- Just use the index directly to look up the corresponding row

This is why embedding layers take integer indices, not one-hot vectors!

3.3.1 Mathematical Equivalence

For vocabulary size 3, embedding dimension 2:

$$\text{Weight matrix } W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

$$\text{One-hot for word 2: } [0, 1, 0] \times W = [w_{21}, w_{22}]$$

$$\text{Index lookup: } W[1] = [w_{21}, w_{22}] \quad \text{(same result!)}$$

Embedding vs Dense Layer

An embedding layer is mathematically equivalent to a dense layer with:

- Linear activation (no non-linearity)
- No bias term
- One-hot input

But embedding uses efficient lookup instead of matrix multiplication!

3.4 Embedding Layer Parameters

```python
from tensorflow.keras.layers import Embedding

# Embedding(input_dim, output_dim)
# input_dim = vocabulary size (how many unique indices)
# output_dim = embedding dimension

embedding = Embedding(input_dim=10001, output_dim=128)
# Parameters: 10001 * 128 = 1,280,128
```

Example: Parameter Calculation

For `Embedding(200, 3)`:

- Input: index from 0 to 199 (200 possible words)
- Output: 3-dimensional dense vector
- Parameters: $200 \times 3 = 600$ (the weight matrix)

3.5 Embedding vs Bag of Words

Aspect	**Bag of Words**	**Embedding + Sequence**
Time/Order	Lost completely	Preserved
Representation	Sparse (many zeros)	Dense (meaningful values)
Repeated words	Counted (frequency)	Each occurrence kept
Downstream model	Dense layers (no RNN)	RNN, LSTM, Transformer
Memory	High for large vocab	Lower (embedding dimension)

4 N-grams: Capturing Local Context

4.1 What are N-grams?

Definition: N-gram

An n-gram is a contiguous sequence of n tokens from text:

- **Unigram** (n=1): Single words (standard tokenization)
- **Bigram** (n=2): Two consecutive words
- **Trigram** (n=3): Three consecutive words

4.2 Why N-grams Matter

Example: Sentiment Analysis

Consider the sentence: "This movie was not funny"

Unigrams: [This, movie, was, not, funny]

- "funny" suggests positive sentiment
- Completely misses the negation!

Bigrams: [This movie, movie was, was not, not funny]

- "not funny" captures the negation
- Much better for sentiment analysis!

4.3 Creating N-grams

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = ["Henry Ford introduced the Model T"]

# Bigrams only
vectorizer_2gram = CountVectorizer(ngram_range=(2, 2))
X = vectorizer_2gram.fit_transform(corpus)
print(vectorizer_2gram.get_feature_names_out())
# ['ford introduced', 'henry ford', 'introduced the',
#  'model t', 'the model']

# Both bigrams and trigrams
vectorizer_mixed = CountVectorizer(ngram_range=(2, 3))
# Includes all 2-grams AND all 3-grams
```

4.4 N-gram Trade-offs

Vocabulary Explosion

As n increases, vocabulary size grows dramatically:

- Unigrams: V words
- Bigrams: Up to V^2 possible combinations
- Trigrams: Up to V^3 possible combinations

In practice, most combinations don't appear, but vocabulary still grows significantly.

Practical Recommendations for Sentiment Analysis

- Unigrams alone: Often insufficient (misses negation)
- Bigrams + Trigrams: Usually best performance
- Beyond 3-grams: Vocabulary becomes too large, diminishing returns

Recommended: Use `ngram_range=(2, 3)` for sentiment analysis.

4.5 Limitations of N-grams

1. **Only local context**: Cannot capture long-range dependencies
2. **Increased sparsity**: Even more zeros in representation
3. **Storage**: Larger vocabulary requires more memory
4. **Still loses global order**: Bag of n-grams is still a "bag"

5 Convolutional Neural Networks (CNNs)

5.1 From Images to Text

CNNs were originally designed for images but work well on any array-structured data:

Text as an Array

A sentence represented as embeddings becomes a 2D array:

- Rows: Time steps (words in sequence)
- Columns: Embedding dimensions

This array structure is similar to a single-channel image!

5.2 Key CNN Concepts

5.2.1 1. Filters (Kernels)

Definition: Filter/Kernel

A small matrix of learnable weights that slides across the input, computing a weighted sum at each position. The filter learns to detect specific patterns.

For text: A filter of size $(3, d)$ looks at 3 consecutive word embeddings at a time—similar to trigrams but with learned weights!

5.2.2 2. Convolution Operation

Example: 2D Convolution

Input: 3×3 array, Filter: 2×2

$$\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 2 & 2 \\ \hline \end{array} + \text{bias } 100$$

Top-left position: $1 \times 1 + 2 \times 1 + 4 \times 2 + 5 \times 2 + 100 = 121$

5.2.3 3. Weight Sharing

Critical: CNN Weight Sharing

Unlike fully connected layers, CNNs use the **same weights** for every position:

- Dramatically reduces parameters
- Position-invariant feature detection
- Same pattern detected anywhere in input

Similar to how RNNs share weights across time, CNNs share weights across space.

5.3 CNN Architecture Components

5.3.1 Padding

Padding	Effect
`valid`	No padding; output smaller than input
`same`	Zero padding added; output same size as input (if stride=1)

Output size formula (no padding): output = input − filter + 1

Example: Input 28, filter 3: Output = $28 - 3 + 1 = 26$

5.3.2 Strides

Stride determines how many positions the filter moves at each step:

- Stride 1: Move one pixel at a time (default)
- Stride 2: Skip every other position (reduces output size)

5.3.3 Max Pooling

Definition: Max Pooling

Downsampling operation that takes the maximum value from each region:

- Reduces spatial dimensions
- Provides slight translation invariance
- No learnable parameters

2×2 max pooling with stride 2 halves each dimension.

Example: Max Pooling

5	3	2	1
1	7	4	6
9	2	8	3
4	5	1	2

$\xrightarrow{\text{2×2 max pool}}$

7	6
9	8

5.4 Complete CNN Example

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([
    # Input: 32x32x3 (RGB image)
    Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
    # Output: 30x30x32 (lost 2 pixels, gained 32 feature maps)

    MaxPooling2D((2,2)),
```

```
    # Output: 15x15x32 (halved dimensions)

    Conv2D(64, (3,3), activation='relu'),
    # Output: 13x13x64

    MaxPooling2D((2,2)),
    # Output: 6x6x64

    Conv2D(64, (3,3), activation='relu'),
    # Output: 4x4x64

    Flatten(),
    # Output: 1024 (4*4*64 = 1024)

    Dense(64, activation='relu'),
    Dense(10, activation='softmax')  # 10-class classification
])
```

5.5 Understanding Feature Maps

Multiple Filters Create Depth

- Each filter produces one **feature map**
- `Conv2D(32, ...)` means 32 filters $\Rightarrow$ 32 feature maps
- Next layer's filters have depth matching previous output
- Example: After Conv2D(32), next filter is $(3, 3, 32)$—32 sub-filters

5.6 Flatten Layer

Definition: Flatten

Reshape multi-dimensional array to 1D vector for dense layers:

- $(4, 4, 64) \rightarrow 1024$
- No learnable parameters
- Just reshaping, not computation

5.7 CNN vs RNN for NLP

Aspect	CNN	RNN/LSTM
Context	Local (filter size)	Sequential (can be long)
Parallelization	Highly parallelizable	Sequential processing
Training speed	Faster	Slower
Long dependencies	Limited by filter size	Better (LSTM)
Common use in NLP	Text classification	Sequence generation

5.8 Special Filter: 1×1 Convolution

What Does 1×1 Convolution Do?

A 1×1 filter doesn't look at spatial neighbors—it only:

- Collapses channels/colors together (weighted average per pixel)
- Reduces dimensionality without losing spatial resolution
- Used in inception modules to reduce computation

Think of it as: applying a dense layer to each spatial position independently.

6 Practical Considerations

6.1 The Sigmoid Loss Explosion Problem

Log of Zero Problem

When using sigmoid output with cross-entropy loss:

- If model is very confident but wrong: $\hat{y} \approx 0$ or $\hat{y} \approx 1$
- Loss $= -\log(\hat{y})$ or $-\log(1-\hat{y})$
- $\log(0.0001) \approx -9.2$, $\log(0.00001) \approx -11.5$
- Loss explodes to infinity!

Solution: Add dropout to prevent overconfident predictions.

6.2 Stop Words

Definition: Stop Words

Common words with little semantic value: "the," "a," "is," "are," "in," etc.

Removing them:

- Reduces vocabulary size
- Speeds up training
- May improve classification (less noise)

```
from sklearn.feature_extraction.text import CountVectorizer

# Remove English stop words
vectorizer = CountVectorizer(stop_words='english')

# Or provide custom list
custom_stops = ['the', 'a', 'an', 'is', 'are']
vectorizer = CountVectorizer(stop_words=custom_stops)
```

6.3 When to Use Deep Learning

Data Size Matters

- $<$ **1000 samples**: Classical ML (logistic regression, random forest)
- **1000–10,000 samples**: Consider simple neural networks
- $>$ **10,000 samples**: Deep learning becomes viable
- $>$ **100,000 samples**: Deep learning often outperforms

Deep learning has many parameters—needs sufficient data to train properly.

7 One-Page Summary

Bag of Words

Concept: Count word frequencies, ignore order
Steps: Tokenize → Build vocabulary → Count → Create vector
Limitations: Loses order, sparse, no semantics

N-grams

Concept: Token = n consecutive words
Benefit: Captures local context ("not funny" vs "funny")
Cost: Vocabulary explosion
Recommendation: Use (2,3) for sentiment analysis

Embeddings

What: Learned mapping from sparse to dense vectors
Why index works: One-hot × matrix = row selection
Parameters: vocab_size × embedding_dim
Key insight: Same as dense layer but uses efficient lookup

CNN Key Concepts

Filter	Sliding window of learnable weights
Stride	How many positions to move
Padding	valid (shrinks) or same (maintains size)
Max Pool	Take max from each region
Flatten	Reshape to 1D for dense layers

CNN Architecture Flow

Input → [Conv → ReLU → Pool] $\times N$ → Flatten → Dense → Output
Weight sharing: Same filter weights at all positions
Feature maps: Number of filters = depth of output

8 Glossary

Term	**Definition**
Bag of Words	Text representation counting word frequencies without order
Bigram	Two consecutive tokens from text
Convolution	Operation sliding a filter across input, computing weighted sums
Embedding	Learned mapping from sparse indices to dense vectors
Feature Map	Output of applying one filter to entire input
Filter/Kernel	Small matrix of learnable weights in CNN
Flatten	Reshape multi-dimensional array to 1D vector
Max Pooling	Downsampling by taking maximum in each region
N-gram	Sequence of n consecutive tokens
One-hot Encoding	Sparse vector with single 1 indicating category
Padding	Adding zeros around input to control output size
Sparse	Vector with mostly zeros
Stop Words	Common words removed before analysis (the, a, is)
Stride	Step size when sliding filter across input
Trigram	Three consecutive tokens from text
Unigram	Single token (standard word)
Vocabulary	Set of unique tokens from training corpus
Weight Sharing	Using same weights at different positions (CNN, RNN)