CSCI E-89B: Introduction to Natural Language Processing
Lecture 03: Text Preprocessing and NLP Pipelines

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-89B: Introduction to Natural Language Processing
■ **Week:**	Lecture 03
■ **Instructor:**	Dmitry Kurochkin
■ **Objective:**	Master text preprocessing techniques including tokenization, stemming, lemmatization, and embeddings for building NLP classification systems

Contents

1 Quiz Review: RNN Architecture Deep Dive

Lecture Overview

This lecture begins with an in-depth review of recurrent neural network architecture, focusing on parameter calculations, LSTM mechanics, and bidirectional networks. Understanding these fundamentals is essential for applying RNNs to natural language processing tasks.

1.1 Simple RNN Parameter Calculation

Consider a Simple RNN layer defined as:

```
SimpleRNN(2, activation='tanh', input_shape=(20, 4))
```

Understanding the Input Shape

The input shape `(20, 4)` means:

- **20**: Number of time steps (sequence length)—how many vectors in the sequence
- **4**: Dimensionality of each input vector—each time step receives a 4-dimensional vector

The number **2** specifies 2 hidden neurons in the recurrent layer.

1.1.1 Time Steps Don't Affect Parameters

A crucial insight: **time steps do not affect the number of trainable parameters**. Whether you have 20 or 100 time steps, the weight matrices remain the same size because:

- The same weights are **shared across all time steps**
- During training, the network unrolls for T time steps, but uses identical weights at each step
- You can even specify `None` for time steps during model definition

Example: RNN Parameter Formula

For a Simple RNN with:

- $n_{in} = 4$ (input dimension)
- $n_h = 2$ (hidden units/neurons)

The number of parameters per neuron:

$$\text{Parameters per neuron} = n_{in} + n_h + 1 = 4 + 2 + 1 = 7$$

Total parameters:

$$\text{Total} = 7 \times n_h = 7 \times 2 = \boxed{14}$$

The breakdown:

- $n_{in} = 4$: weights from input vector x_t
- $n_h = 2$: weights from previous hidden state h_{t-1} (recurrent connections)
- 1: bias term

1.2 RNN Visual Understanding

The Simple RNN can be visualized as a network that **unfolds through time**:

Compact View (Memory Cell)
Input: x_t (4-dim) $\rightarrow$ [2 neurons] $\rightarrow$ Output: y_t (2-dim)
$\uparrow$ Recurrent connection from y_{t-1}

When unfolded for $T = 20$ time steps:

- Time step 1: $x_1 \rightarrow$ [2 neurons] $\rightarrow y_1$
- Time step 2: $x_2 + y_1 \rightarrow$ [2 neurons] $\rightarrow y_2$
- $\vdots$
- Time step 20: $x_{20} + y_{19} \rightarrow$ [2 neurons] $\rightarrow y_{20}$

The Unfolded View is for Understanding

While we draw 20 separate boxes for visualization, the **weights are shared across all time steps**. This is what makes RNNs efficient for sequence processing—they don't need separate parameters for each position in the sequence.

2 RNN Output Architectures

Key Summary

A recurrent neural network can operate in different modes depending on how we use its outputs. The claim "RNN can only map sequence to sequence" is **FALSE**—there are multiple valid architectures.

2.1 Many-to-Many (Sequence to Sequence)

- **Input**: Sequence of vectors $[x_1, x_2, \ldots, x_T]$
- **Output**: Sequence of vectors $[y_1, y_2, \ldots, y_T]$
- **Use case**: Part-of-speech tagging, sequence labeling
- **Keras setting**: `return_sequences=True`

2.2 Many-to-One

- **Input**: Sequence of vectors $[x_1, x_2, \ldots, x_T]$
- **Output**: Single vector y_T (only the last output)
- **Use case**: Sentiment analysis, text classification
- **Keras setting**: `return_sequences=False` (default)

Example: Sentiment Classification

Given a movie review as input:

- Process entire sequence through RNN
- Take only the final hidden state h_T
- Pass through dense layer with sigmoid to get positive/negative probability

We don't need predictions at every word—just a single classification at the end.

2.3 One-to-Many

- **Input**: Single vector x (or one input followed by zeros)
- **Output**: Sequence of vectors $[y_1, y_2, \ldots, y_T]$
- **Use case**: Image captioning—input an image, output a sentence

Image Captioning Architecture

For generating captions from images:

1. Process image through CNN to get a feature vector
2. Use this vector as initial input to RNN
3. Feed zeros (or learned tokens) at subsequent time steps
4. RNN generates word sequence describing the image

Note: RNN was not great for *generating* images from text. That required transformers and diffusion models.

2.4 Encoder-Decoder Architecture (Sequence to Sequence)

A special many-to-many architecture with a **bottleneck**:

1. **Encoder**: Process input sequence, compress into a single "context" vector
2. **Bottleneck**: The final encoder hidden state represents the entire input
3. **Decoder**: Generate output sequence from the context vector

Definition: Autoencoder for Sequences

An autoencoder maps input to itself through a bottleneck:

- **Training objective**: Reconstruct input from compressed representation
- **Bottleneck benefit**: Forces the network to learn the most important features
- **Application**: The encoder alone can create meaningful sentence embeddings

Example: Sentence Compression

Consider compressing a sentence into a single vector:

- Input: Sequence of word vectors representing a sentence
- Process through RNN encoder
- Final hidden state h_T = "sentence embedding"
- This vector lives in, say, 128-dimensional space

Analogy: We live comfortably in 3D (or 4D with time). Why can't sentences "live" in 128D or 1000D space? There's plenty of room for every unique sentence!

2.4.1 Applications Beyond Translation

While translation was a major application of encoder-decoder RNNs (2014–2017), the architecture has other uses:

- **Denoising autoencoders**: Train on (noisy image, clean image) pairs. The bottleneck learns to ignore noise.
- **Image restoration**: Scratched or damaged images can be recovered
- **Compression**: Learn efficient representations of data

Translation Note

For translation, even during the RNN era, people didn't typically pretrain on the same language (autoencoder style). They trained directly on (source language, target language) pairs because the nuances of translation require learning from actual parallel data.

3 LSTM: Long Short-Term Memory

3.1 LSTM Cell State and Hidden State Dimensions

Key Summary

Quiz Question: Do the cell state C and hidden state H in an LSTM have the same dimensions?
Answer: **TRUE**. By the mathematical design of LSTM, C and H must have identical dimensions.

3.1.1 Mathematical Proof from LSTM Operations

Looking at how H is computed from C:

$$H_t = O_t \odot \tanh(C_t) \tag{1}$$

Where:

- $\odot$ denotes element-wise (Hadamard) multiplication
- $\tanh(C_t)$ is applied element-wise to C_t
- O_t is the output gate

Key insight: Element-wise multiplication requires both operands to have the **same dimension**. Therefore:

$$\dim(H_t) = \dim(O_t) = \dim(C_t)$$

LSTM Sub-Networks

When you specify `LSTM(5)`, it means:

- H is 5-dimensional
- C is 5-dimensional
- Each of the 4 sub-networks (forget gate, input gate, candidate, output gate) has 5 neurons
- Total neurons inside: $4 \times 5 = 20$

3.2 Understanding LSTM Gates

Definition: LSTM Components

- **Forget Gate (f_t)**: Controls what to remove from cell state
- **Input Gate (i_t)**: Controls what new information to add
- **Candidate Values ($\tilde{C}_t$)**: New candidate information
- **Output Gate (o_t)**: Controls what to output from cell state

Each gate is a fully connected layer with its own weights, but all produce outputs of dimension n_h (the number of hidden units).

3.2.1 Key Difference from Simple RNN

- Simple RNN: Inputs enter each neuron directly
- LSTM: Inputs enter each **sub-network** (4 of them)
- LSTM has two recurrent paths:
  - H goes to all 4 sub-networks (like Simple RNN's recurrence)
  - C flows through the cell state "highway" (mostly bypasses sub-networks)

Example: LSTM Parameter Count

For `LSTM(32)` with input dimension 128:

$$\begin{aligned}
\text{Parameters} &= 4 \times [(n_{in} + n_h + 1) \times n_h] && (2)\\
&= 4 \times [(128 + 32 + 1) \times 32] && (3)\\
&= 4 \times [161 \times 32] && (4)\\
&= 4 \times 5152 = 20608 && (5)
\end{aligned}$$

The factor of 4 comes from the 4 sub-networks (gates + candidate).

4 Bidirectional RNNs

4.1 Why Bidirectional?

Sometimes the **beginning** of a sequence is more important than the end. Sometimes the **end** is more important. And sometimes **both matter**.

Example: German Negation

In German, negation often appears at the end of a sentence. To understand if a statement is positive or negative, you need to process the entire sentence, including the end. A forward-only RNN might struggle because by the time it reaches the negation, important early context may have faded.

4.2 Bidirectional Architecture

Definition: Bidirectional RNN

Process the sequence in **both directions** simultaneously:

1. **Forward pass**: Process $[A, B, C, D, E]$ left to right
2. **Backward pass**: Process $[E, D, C, B, A]$ (reversed) left to right
3. **Concatenate**: Combine outputs from both directions

If each direction has 2 hidden units, the concatenated output has 4 dimensions.

4.3 Parameter Count for Bidirectional Layers

Critical: Bidirectional Parameter Doubling

Quiz Question: If LSTM has 8,320 parameters, how many does Bidirectional LSTM have?
Answer: Exactly **double** = 16,640 parameters.
The forward and backward LSTMs are **completely independent** networks with their own weights. They don't share anything except the input data (one gets original, one gets reversed).

4.3.1 What About the Next Layer?

Consider this network:

```
model.add(Bidirectional(LSTM(32)))  # Output: 64 dimensions
model.add(Dense(1, activation='sigmoid'))  # Input: 64, Output: 1
```

- LSTM(32) produces 32-dimensional output
- Bidirectional concatenates: $32 + 32 = 64$ dimensions
- Dense layer receives 64-dimensional input
- Dense parameters: $64 \times 1 + 1 = 65$ (not 33!)

Not Everything Doubles

The bidirectional layer's parameters double because two independent networks run. But the **next layer's parameters don't double**—they just need to account for the concatenated (doubled) input dimension, plus one shared bias.

5 What is Natural Language Processing?

Lecture Overview

Natural Language Processing (NLP) is the field of making computers understand, interpret, and generate human language. It sits at the intersection of **linguistics** and **artificial intelligence**.

5.1 AI, ML, DL, and NLP: Understanding the Hierarchy

Definition: The Nested Relationship

- **Artificial Intelligence (AI)**: The broadest category—any technique that makes computers behave intelligently. Includes rule-based systems where experts hardcode decisions.
- **Machine Learning (ML)**: A subset of AI where computers learn rules from data rather than having rules programmed. Given (data + labels), the algorithm learns the mapping.
- **Deep Learning (DL)**: A subset of ML using multi-layer neural networks. Excels at learning complex patterns from unstructured data.
- **NLP**: An *application domain* that intersects with all the above. NLP uses traditional AI (linguistic rules), ML, and DL to process language.

Example: The Key Difference: AI vs ML

Traditional AI (Rule-Based):

- Input: Data + **Rules** (hardcoded by experts)
- Output: Predictions/Decisions
- Example: Doctor's decision rules stored in computer, nurse looks up treatment

Machine Learning:

- Input: Data + **Labels/Results**
- Output: **Rules** (learned patterns)
- Example: Linear regression: given (X, Y) pairs, learn coefficients β

5.2 Why NLP Needs More Than Just ML

NLP problems are so complex that we often need linguistic knowledge **in addition to** machine learning:

- **Stemming/Lemmatization**: Linguistic rules that "run," "ran," "running" are the same word
- **Syntax parsing**: Grammar rules for sentence structure
- **Named Entity Recognition**: Understanding that "Paris" is a city, not just a word

This is why NLP is truly **multidisciplinary**—combining linguistics, statistics, and computer science.

5.3 NLP Application Areas

Application	**Description**
Text Classification	Spam detection, topic categorization, sentiment analysis
Named Entity Recognition	Identifying names, organizations, locations in text
Sentiment Analysis	Determining emotional tone (positive/negative)
Information Retrieval	Search engines, TF-IDF, BM25
Optical Character Recognition	Converting images of text to digital text
Machine Translation	Converting text between languages
Text Summarization	Condensing documents to key points
Speech Recognition	Converting audio to text
Question Answering	Providing specific answers to questions
Chatbots	Conversational AI systems
Topic Modeling	Discovering themes in document collections
Language Generation	Creating human-like text

5.4 NLP Challenges

Ambiguity is Everywhere

Consider the headline: “Court to try shooting defendant”

What’s happening? Is the court:

1. Going to **try** (in a legal sense) the defendant who did the shooting?
2. Going to **try shooting** the defendant?

Humans use world knowledge to disambiguate. Teaching computers this is extremely hard!

6 Text Preprocessing: The Foundation of NLP

Key Summary

Before feeding text to any neural network, we must transform it into numbers. This involves multiple preprocessing steps that can significantly impact model performance.

6.1 The Preprocessing Pipeline

1. **Tokenization**: Split text into units (tokens)
2. **Normalization**: Stemming or lemmatization
3. **Vocabulary Building**: Create word-to-index mapping
4. **Encoding**: Convert tokens to numerical representations
5. **Padding**: Make all sequences same length

6.2 Tokenization

Definition: Tokenization

The process of segmenting text into individual units called **tokens**. A token can be:

- A word (most common)
- A subword (for handling unknown words)
- A character
- A sentence

6.2.1 Word Tokenization Example

```python
from nltk.tokenize import word_tokenize

text = "Henry Ford's innovation, the assembly line process."
tokens = word_tokenize(text)
# Result: ['Henry', 'Ford', "'s", 'innovation', ',', 'the',
#           'assembly', 'line', 'process', '.']
```

Why Use Libraries?

Don't just split on spaces! Libraries like NLTK handle edge cases:

- Contractions: "can't" should stay together
- Punctuation: "end." should separate the period
- Possessives: "Ford's" might become ["Ford", "'s"]

6.2.2 Sentence Tokenization

Useful when sentences are your unit of analysis:

```python
from nltk.tokenize import sent_tokenize

text = "He created assembly lines. This revolutionized production."
sentences = sent_tokenize(text)
# Result: ['He created assembly lines.', 'This revolutionized production.']
```

6.3 Stop Words

Definition: Stop Words

Common words that carry little meaning for analysis: "the," "a," "is," "are," "in," etc.

Removing them:

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

When NOT to Remove Stop Words

For tasks like translation or language modeling, stop words are essential! "I am *not* happy" loses crucial meaning without "not."

7 Stemming and Lemmatization

7.1 The Problem of Word Variants

Consider: "running," "runs," "ran"—all forms of "run." Should our model treat them as three different words or one?

Benefits of reducing to base form:

- Smaller vocabulary
- Better generalization (model learns one representation)
- Improved performance on classification tasks

7.2 Stemming

Definition: Stemming

Mechanically remove word endings using rules. Fast but imprecise.

- "running" → "run" (good!)
- "transportation" → "transport" (good!)
- "electric" → "electr" (not a word!)
- "Henry" → "henri" (changed name!)

7.2.1 Popular Stemmers

Stemmer	Characteristics
Porter Stemmer	Classic (1979), widely used, moderate aggression
Snowball Stemmer	Porter's improvement, supports multiple languages
Lancaster Stemmer	Most aggressive, cuts more from words

```python
from nltk.stem import PorterStemmer, SnowballStemmer

porter = PorterStemmer()
snowball = SnowballStemmer("english")

words = ['running', 'runs', 'profoundly', 'driving']

porter_results = [porter.stem(w) for w in words]
# ['run', 'run', 'profoundli', 'drive']

snowball_results = [snowball.stem(w) for w in words]
# ['run', 'run', 'profound', 'drive']
```

7.3 Lemmatization

Definition: Lemmatization

Use linguistic knowledge to find the **dictionary form** (lemma) of a word. Slower but more accurate.

- "running" → "run" (correct verb lemma)
- "cars" → "car" (correct noun lemma)
- "driving" → "drive" (not "driv"!)
- "was" → "be" (irregular verb handled correctly)

7.3.1 Lemmatization Libraries

Library	**Characteristics**
NLTK WordNet	Academic standard, requires POS tag, English only
SpaCy	Industry standard, fast, multi-language, handles new words

```python
import spacy

nlp = spacy.load("en_core_web_sm")
text = "The cars were driving quickly"
doc = nlp(text)

lemmas = [token.lemma_ for token in doc]
# ['the', 'car', 'be', 'drive', 'quickly']
```

7.4 Stemming vs. Lemmatization: When to Use Which?

Factor	**Stemming**	**Lemmatization**
Speed	Fast	Slower
Accuracy	Lower	Higher
Output	May not be valid word	Always valid word
Language support	Good	Varies
For classification	Often sufficient	May not add much
For translation	Inadequate	Necessary

Practical Recommendation

For text classification (sentiment, topic):

1. Try no stemming/lemmatization first
2. Try stemming
3. Try lemmatization
4. Compare validation accuracy

Often stemming is "good enough" and faster!

8 From Words to Vectors: Embeddings

8.1 The Problem with One-Hot Encoding

Representing words as one-hot vectors:

- "cat" = [1, 0, 0, 0, ..., 0]
- "dog" = [0, 1, 0, 0, ..., 0]
- "table" = [0, 0, 1, 0, ..., 0]

Problems:

1. **High dimensionality**: 10,000-word vocabulary = 10,000-dimensional vectors
2. **Sparse**: Almost all zeros, wasted computation
3. **No semantic meaning**: "cat" and "dog" are as different as "cat" and "table"

8.2 What is an Embedding?

Definition: Word Embedding

A mapping from sparse, high-dimensional one-hot vectors to dense, low-dimensional vectors where **semantic similarity** is captured by **vector proximity**.

$$\text{One-hot: } [0, 0, 1, 0, \ldots, 0] \text{ (10,000 dimensions)}$$
$$\text{Embedding: } [0.23, -1.5, 0.87, \ldots] \text{ (128 dimensions)}$$

8.3 How Embeddings Work

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

# Vocabulary size: 10001 (10000 words + 1 OOV token)
# Embedding dimension: 128
embedding_layer = Embedding(input_dim=10001, output_dim=128)
```

What happens internally:

1. Layer has a weight matrix of shape (10001, 128)
2. Each row corresponds to one word's embedding
3. Input: word index (integer)
4. Output: corresponding row from weight matrix
5. **Weights are learned during training**

Example: Embedding Lookup

Input index: 5 (representing "cat")

The embedding layer simply looks up row 5 of its weight matrix:

```
# Conceptually:
weight_matrix[5] = [0.23, -1.5, 0.87, ...]  # 128 numbers
```

No matrix multiplication needed—just a lookup! This is much faster than multiplying a one-hot vector by a weight matrix.

8.4 Why Embeddings are Trainable

Learning Semantics

Unlike fixed mappings like [1, 0] for female and [0, 1] for male, embedding values are **learned from data**.

During training:

- Words appearing in similar contexts get similar embeddings
- "king" - "man" + "woman" ≈ "queen"
- Relationships are encoded as vector arithmetic!

8.5 Embedding Layer vs. Dense Layer

	Embedding Layer	**Dense Layer**
Input	Integer indices	Continuous vectors
Operation	Table lookup	Matrix multiplication
Bias	No bias	Has bias
Activation	Linear (none)	Any
Efficiency	Very fast	Slower for sparse input

Mathematically, an embedding layer is equivalent to a dense layer with:

- Linear activation
- No bias
- One-hot input

But the lookup implementation is much more efficient!

9 Complete Text Classification Pipeline

Lecture Overview

This section walks through building a neural network for text classification, using the 20 Newsgroups dataset to classify posts as either "hockey" or "for sale."

9.1 Data Preparation

```python
from sklearn.datasets import fetch_20newsgroups

# Select two categories for binary classification
categories = ['rec.sport.hockey', 'misc.forsale']

# Load training and test data
train_data = fetch_20newsgroups(subset='train', categories=categories)
test_data = fetch_20newsgroups(subset='test', categories=categories)

# train_data.data: list of text documents
# train_data.target: list of labels (0 or 1)
```

9.2 Building the Vocabulary

```python
from nltk.tokenize import word_tokenize
from collections import defaultdict

MAX_FEATURES = 10000  # Keep only top 10,000 words

# Count word frequencies across all training documents
word_freq = defaultdict(int)
for text in train_data.data:
    tokens = word_tokenize(text.lower())
    for token in tokens:
        word_freq[token] += 1

# Sort by frequency, keep top MAX_FEATURES
sorted_words = sorted(word_freq.items(), key=lambda x: x[1], reverse=True)
word_index = {'<OOV>': 0}  # Index 0 reserved for out-of-vocabulary
for i, (word, _) in enumerate(sorted_words[:MAX_FEATURES]):
    word_index[word] = i + 1
```

9.3 Converting Text to Sequences

```python
def text_to_sequence(text, word_index):
    """Convert text to sequence of word indices."""
    tokens = word_tokenize(text.lower())
    sequence = [word_index.get(token, 0) for token in tokens]  # 0 for OOV
```

```
    return sequence

# Convert all documents
train_sequences = [text_to_sequence(text, word_index)
                   for text in train_data.data]
```

9.4 Padding Sequences

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

MAX_LENGTH = 500  # Maximum sequence length

# Pad sequences to same length (truncate if longer, pad if shorter)
X_train = pad_sequences(train_sequences, maxlen=MAX_LENGTH, padding='post')
X_test = pad_sequences(test_sequences, maxlen=MAX_LENGTH, padding='post')
```

9.5 Building the Model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

def build_model():
    model = Sequential([
        # Embedding: 10001 words -> 128 dimensions
        Embedding(input_dim=MAX_FEATURES + 1, output_dim=128),

        # Dropout on embeddings
        Dropout(0.2),

        # LSTM layer
        LSTM(128),

        # Dropout before output
        Dropout(0.2),

        # Binary classification output
        Dense(1, activation='sigmoid')
    ])

    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',  # For binary classification
        metrics=['accuracy']
    )
    return model
```

9.6 Why Binary Cross-Entropy?

Definition: Binary Cross-Entropy Loss

For binary classification with sigmoid output:

$$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Where:

- $y \in \{0, 1\}$: true label
- $\hat{y} \in [0, 1]$: predicted probability

Mean Squared Error would have terrible gradients for sigmoid output—cross-entropy is essential!

9.7 Training and Results

```python
model = build_model()
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.1
)

# Evaluate on test set
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
```

Results with different preprocessing:

Preprocessing	Test Accuracy
Tokenization only	∼93.5%
Tokenization + Stemming	∼97%
Tokenization + Lemmatization	∼95–96%

Interpreting Results

Stemming performed best here, possibly because:

- Reduced vocabulary helps with limited data
- Word variants unified (“hockey,” “Hockey” become same)
- The classification task doesn’t require precise word forms

Always experiment! Different tasks and datasets may favor different preprocessing.

10 Practical Considerations

10.1 Out-of-Vocabulary (OOV) Handling

The OOV Problem

Test data often contains words not seen during training. What do we do?

Option 1: Ignore unknown words (skip them)

Option 2: Map to special OOV token (index 0)

Option 2 is usually better—the model knows "there was a word here I don't recognize."

Why Not Use All Words?

Q: Why not add all English dictionary words to vocabulary?

A: Because the model only learns representations for words it sees during training! Adding "elephant" to vocabulary doesn't help if no training document mentions elephants—the model has no learned embedding for it.

The vocabulary should come from training data, not external dictionaries.

10.2 Dropout for Regularization

Definition: Dropout

During training, randomly set a fraction of neurons' outputs to zero. This prevents **overfitting** by:

- Preventing co-adaptation of neurons
- Creating an ensemble effect
- Forcing redundant representations

10.2.1 Dropout in RNN Models

Three places to apply dropout:

1. **Input dropout**: Applied to embedding outputs
2. **Recurrent dropout**: Applied to recurrent connections (special handling)
3. **Output dropout**: Applied to layer outputs

```
# Dropout on inputs/outputs
Dropout(0.2)  # 20% of values set to 0

# Recurrent dropout inside LSTM
LSTM(128, dropout=0.2, recurrent_dropout=0.2)
```

Example: How Dropout Works

With `Dropout(0.2)` on a sequence:

```
Original:  [1.7,  0.9, -1.3,  2.1, 0.5]
After:     [1.7,  0.0, -1.3,  2.1, 0.0]  # 20% zeroed
```

Each mini-batch gets different random zeros. This variability prevents overfitting even with many epochs.

11 One-Page Summary

RNN Architecture

Parameter Counting:

- Simple RNN: $(n_{in} + n_h + 1) \times n_h$
- LSTM: $4 \times (n_{in} + n_h + 1) \times n_h$
- Bidirectional: $2\times$ base parameters

LSTM: Cell state C and hidden state H have **same dimensions**.

Text Preprocessing Pipeline

Tokenization → **Normalization** → **Vocabulary** → **Encoding** → **Padding**

Stemming	Fast, rule-based, may produce non-words
Lemmatization	Slower, dictionary-based, always valid words

Word Embeddings

One-Hot: Sparse, high-dimensional, no semantics

Embedding: Dense, low-dimensional, learned semantics

Embedding layer = lookup table with trainable weights

Model Architecture for Text Classification

```
Embedding(vocab_size, embed_dim)
-> Dropout(0.2)
-> LSTM(hidden_units)
-> Dropout(0.2)
-> Dense(1, activation='sigmoid')
```

Loss: Binary Cross-Entropy | Optimizer: Adam

Key Formulas

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

Binary Cross-Entropy: $L = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$

LSTM Cell Update:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{(forget gate)}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad \text{(cell state)}$$
$$h_t = o_t \odot \tanh(C_t) \quad \text{(hidden state)}$$

12 Glossary

Term	Definition
Autoencoder	Neural network trained to reconstruct input through a bottleneck, learning compressed representations
Bidirectional RNN	RNN that processes sequences in both forward and backward directions, concatenating outputs
Binary Cross-Entropy	Loss function for binary classification comparing predicted probabilities to true labels
Dropout	Regularization technique that randomly zeros neurons during training to prevent overfitting
Embedding	Learned dense vector representation of discrete items (words) capturing semantic relationships
Encoder-Decoder	Architecture with two parts: encoder compresses input, decoder generates output
Lemmatization	Reducing words to dictionary form using linguistic knowledge ("drove" → "drive")
LSTM	Long Short-Term Memory—RNN variant with gates to control information flow, handling long sequences
One-Hot Encoding	Sparse vector with single 1 indicating category, all other positions 0
OOV (Out-of-Vocabulary)	Words not present in the training vocabulary, mapped to special token
Padding	Adding zeros to sequences to make them equal length for batch processing
Stemming	Reducing words to root form by removing affixes ("running" → "run")
Stop Words	Common words ("the," "is") often removed as they carry little semantic content
Tokenization	Splitting text into individual units (tokens) like words or sentences