■ **Course:** CSCI E-89B: Introduction to Natural Language Processing

■ **Lecture:** Lecture 01 – Neural Networks Foundations

■ **Instructor:** Dmitry Kurochkin

■ **Objective:** Understand the fundamentals of neural networks including architecture, activation functions, loss functions, cost functions, and optimization algorithms for NLP applications

Contents

1 Introduction: Course Overview and Learning Roadmap

Lecture Overview

Welcome to CSCI E-89B: Introduction to Natural Language Processing. This course covers the intersection of deep learning and language processing. Before diving into NLP-specific techniques, we must first establish a strong foundation in neural networks.

Key Learning Objectives:

- Understand the fundamental architecture of neural networks
- Learn about activation functions and their role in introducing non-linearity
- Distinguish between loss functions (individual error) and cost functions (aggregate error)
- Master gradient descent and its variants for optimization
- Apply these concepts using Keras/TensorFlow

1.1 Why Neural Networks for NLP?

Key Information

The Feature Engineering Problem

Traditional machine learning requires **manual feature engineering**—humans must design the features that the model uses.

Example: Polynomial Regression

$$\hat{y} = w_0 + w_1 \cdot x + w_2 \cdot x^2 + w_3 \cdot x^3 \tag{1}$$

Here, x, x^2, and x^3 are features we *manually* created. This works for simple problems, but:

- Images have millions of pixels with complex relationships
- Text has sequences of words with semantic meaning
- How do you manually design features for "the meaning of a sentence"?

Neural networks solve this by learning features automatically from data.

1.2 Course Communication Channels

Channel	Purpose	Notes
Piazza	Official Q&A forum	Instructors monitor and respond; share with class
WhatsApp	Informal student discussions	Not officially monitored; for peer support
Canvas Inbox	Private communication	For personal matters with instructors/TAs

Table 1: *Course Communication Channels*

1.3 Weekly Schedule

Session	Day	Focus
Lecture	Tuesday	Theory and core concepts
TA Session 1	Wed/Thursday	Theory review, problem solving
Instructor's Section	Friday	Python implementations, code examples
TA Session 2	Sat/Sunday	Additional problems (different content from Session 1)

Table 2: *Weekly Session Schedule*

Warning

Important Policy Notes:

- **Quizzes**: No late submissions allowed. Solutions are discussed immediately after the deadline.
- **Assignments**: Due Sundays 11:59 PM Boston time. Late penalty: 10% per day.
- **Final Project**: Strict deadline—extensions require official Extension School paperwork.
- **No dropping**: Unlike some courses, the lowest quiz/assignment is NOT dropped.

1.4 Grading Breakdown

Component	Weight
Weekly Assignments	65%
Weekly Quizzes	20%
Final Project	15%

Table 3: *Grade Distribution*

2 From Linear Regression to Neural Networks

2.1 The Limitations of Linear Models

Definition:

Linear Regression **Linear regression** fits a model that is *linear in its parameters*:

$$\hat{y} = w_0 + w_1 \cdot u_1 + w_2 \cdot u_2 + w_3 \cdot u_3 \tag{2}$$

where u_1, u_2, u_3 can be transformed versions of the input (e.g., $u_1 = x$, $u_2 = x^2$, $u_3 = x^3$).

The Problem:

- Features (u_1, u_2, u_3) must be **manually designed**
- Works for simple data where you can visualize and understand relationships
- Fails for high-dimensional data (images, text, audio)

Why not make the powers learnable?

You might think: "Let's learn the optimal power p in x^p"

$$\hat{y} = w_0 + w_1 \cdot x^{p_1} + w_2 \cdot x^{p_2} + w_3 \cdot x^{p_3} \tag{3}$$

This is a *terrible idea* because:

- The function becomes highly non-linear in parameters
- Optimization becomes extremely difficult (many bad local minima)
- Derivatives with respect to p are complex

2.2 The Neural Network Solution

Important:

The Key Insight Neural networks introduce non-linearity through **activation functions** applied to **linear combinations**.

$$u_1 = f(w_0 + w_1 x_1 + w_2 x_2) \tag{4}$$

Why this specific form?

- Linear combinations are easy to differentiate
- Chain rule applies cleanly
- We can use gradient descent efficiently

The derivative:

$$\frac{\partial u_1}{\partial w} = f'(\cdot) \cdot \frac{\partial}{\partial w}(w_0 + w_1 x_1 + w_2 x_2) \tag{5}$$

The derivative of the linear part is trivial, and f' is usually simple too.

3 Feedforward Neural Networks

Definition:

Feedforward Neural Network (FNN) A **feedforward neural network** is a composition of functions where information flows in one direction—from input to output. Mathematically:

$$\hat{y} = f^{(L)}\left(f^{(L-1)}\left(\dots f^{(1)}(x)\right)\right) \tag{6}$$

Each function $f^{(l)}$ typically consists of a linear transformation followed by a non-linear activation.

3.1 A Simple Two-Layer Network

Consider a network with:

- 2 inputs: x_1, x_2
- 2 hidden neurons: u_1, u_2
- 1 output: $\hat{y}$

Hidden Layer Computation:

$$u_1 = f\left(w_{01}^{(1)} + w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2\right) \tag{7}$$

$$u_2 = f\left(w_{02}^{(1)} + w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2\right) \tag{8}$$

Output Layer Computation:

$$\hat{y} = f\left(w_0^{(2)} + w_1^{(2)}u_1 + w_2^{(2)}u_2\right) \tag{9}$$

Key Information

Understanding the Notation:

- $w_{ij}^{(l)}$: Weight connecting input i to neuron j in layer l
- $w_{0j}^{(l)}$: Bias term for neuron j in layer l
- f: Activation function

3.2 The Role of the Bias Term

Example:

Why Do We Need Bias? The bias term w_0 acts as a **threshold shifter**.
Consider a biological analogy: A neuron "fires" when the cumulative input exceeds a threshold. Without bias, this threshold is fixed at zero. With bias, we can adjust where the activation "turns on."
Mathematically, instead of:

$$f(w_1x_1 + w_2x_2) \quad \text{(threshold at 0)} \tag{10}$$

We have:

$$f(w_0 + w_1x_1 + w_2x_2) \quad \text{(adjustable threshold)} \tag{11}$$

The bias allows the decision boundary to shift away from the origin.

3.3 Universal Approximation Theorem

Theorem 3.1 (Universal Approximation)**.** A feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given an appropriate activation function.

Implications:

- Theoretically, one layer is "enough" for any function
- In practice, deeper networks learn more efficiently
- The choice of activation function matters less theoretically, but significantly in practice

4 Activation Functions

Definition:

Activation Function An **activation function** is a non-linear function applied to the output of a neuron. Without activation functions, stacking linear layers would result in just another linear transformation—unable to learn complex patterns.

4.1 Why Non-linearity is Essential

If we had no activation function:

$$u = W^{(1)}x + b^{(1)} \tag{12}$$

$$\hat{y} = W^{(2)}u + b^{(2)} = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)} \tag{13}$$

This collapses to:

$$\hat{y} = \underbrace{W^{(2)}W^{(1)}}_{W'}x + \underbrace{W^{(2)}b^{(1)} + b^{(2)}}_{b'} \tag{14}$$

A single linear transformation! No matter how many layers, without non-linearity, we cannot learn complex patterns.

4.2 Biological Inspiration: The Step Function

Warning

Historical Note: The Step Function

Early neural networks mimicked biological neurons using the **Heaviside step function**:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{15}$$

Problem: The derivative is zero everywhere (except at $z = 0$ where it's undefined).
This means the cost function becomes **piecewise constant**—gradient descent doesn't know which direction to move! This activation is NOT used in practice.

4.3 Common Activation Functions

4.3.1 ReLU (Rectified Linear Unit)

$$f(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{16}$$

Advantages:

- Computationally efficient

Function	Formula	Range	Use Case
ReLU	$f(z) = \max(0, z)$	$[0, \infty)$	Hidden layers (most common)
Leaky ReLU	$f(z) = \max(\alpha z, z)$	$(-\infty, \infty)$	Hidden layers (avoids dead neurons)
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	$(0, 1)$	Binary classification output
Softmax	$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$	$(0, 1)$, sum $= 1$	Multi-class classification output
Tanh	$f(z) = \tanh(z)$	$(-1, 1)$	Hidden layers, RNNs

Table 4: *Common Activation Functions*

- Doesn't saturate for positive values
- Widely used and well-studied

Disadvantage:

- "Dead neurons": If $z < 0$, gradient is 0, neuron stops learning

4.3.2 Leaky ReLU

$$f(z) = \max(\alpha z, z) \quad \text{where } \alpha \approx 0.01 - 0.1 \tag{17}$$

The small slope for negative values prevents dead neurons. However, α is a **hyperparameter** that must be chosen.

4.3.3 Softmax for Classification

Definition:

Softmax Function For a vector $\mathbf{z} = [z_1, z_2, \ldots, z_M]$:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}} \tag{18}$$

Properties:

- Each output is in $(0, 1)$
- All outputs sum to exactly 1
- Outputs can be interpreted as probabilities

Example:

Softmax Calculation Given logits $\mathbf{z} = [11, 10]$:

$$\text{softmax}(z_1) = \frac{e^{11}}{e^{11} + e^{10}} = \frac{e^{11}}{e^{11}(1 + e^{-1})} \approx 0.73 \tag{19}$$

$$\text{softmax}(z_2) = \frac{e^{10}}{e^{11} + e^{10}} \approx 0.27 \tag{20}$$

The larger input gets the higher probability. The "winning" class is amplified.

4.4 Keras Implementation

```
import keras
from keras import models, layers

model = models.Sequential()

# Hidden layer: 16 neurons with ReLU activation
# Input shape: 900 features
model.add(layers.Dense(16, activation='relu', input_shape=(900,)))

# Output layer: 2 neurons with Softmax (for binary classification)
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

Listing 1: Building a Neural Network in Keras

5 Loss Functions and Cost Functions

Definition:

Loss vs Cost **Loss Function** $L^{(i)}(w)$: Measures the error for a **single** data point $(x^{(i)}, y^{(i)})$.
Cost Function $J(w)$: The **average** loss over the entire dataset:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(w) \tag{21}$$

Analogy:

- Loss = How rotten is *one* apple?
- Cost = On average, how rotten is the entire box of apples?

5.1 Loss Functions for Regression

When predicting continuous values:

5.1.1 Squared Error (SE)

$$L(\hat{y}, y) = (\hat{y} - y)^2 \tag{22}$$

Properties:

- Penalizes large errors more heavily
- Differentiable everywhere
- Sensitive to outliers

When averaged: **Mean Squared Error (MSE)**:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \tag{23}$$

5.1.2 Absolute Error (AE)

$$L(\hat{y}, y) = |\hat{y} - y| \tag{24}$$

Properties:

- Less sensitive to outliers
- Not differentiable at $\hat{y} = y$
- Constant gradient magnitude

When averaged: **Mean Absolute Error (MAE)**

Key Information

MSE vs MAE: When to Use Which?

Both have the same minimum (the correct prediction), but their **gradients behave differently**:

- **MSE**: Gradient gets smaller as you approach the minimum (self-adjusting step sizes)
- **MAE**: Constant gradient (fixed step size regardless of distance to minimum)

The shape of the cost function affects which local minimum you converge to!

5.2 Loss Functions for Classification

5.2.1 Why Not Use MSE for Classification?

Warning

MSE is Terrible for Classification!

Consider classifying images as Cat (1,0) or Dog (0,1):

- True label: $y = [1, 0]$ (Cat)
- Prediction: $\hat{y} = [0.9, 0.1]$

MSE computes: $(0.9 - 1)^2 + (0.1 - 0)^2 = 0.01 + 0.01 = 0.02$

The Problem:

- y lives in a discrete space: $\{[1, 0], [0, 1]\}$
- $\hat{y}$ lives on a continuous line (probabilities summing to 1)
- Computing distance between discrete and continuous spaces creates a highly non-convex cost function
- Result: You get stuck in bad local minima and fail to converge

5.2.2 Cross-Entropy Loss

Definition:

Cross-Entropy For a classification problem with M classes:

$$L(\hat{y}, y) = -\sum_{j=1}^{M} y_j \log(\hat{y}_j) \tag{25}$$

For binary classification (with one-hot encoding):

$$L = -y_1 \log(\hat{y}_1) - y_2 \log(\hat{y}_2) \tag{26}$$

Since y is one-hot encoded (e.g., $[1, 0]$), only one term contributes.

Example:

Cross-Entropy Calculation **Case 1: Good Prediction**

- True: $y = [1, 0]$ (Cat)
- Predicted: $\hat{y} = [0.99, 0.01]$

- Loss: $-1 \cdot \log(0.99) - 0 \cdot \log(0.01) \approx 0.01$

Case 2: Bad Prediction

- True: $y = [1, 0]$ (Cat)
- Predicted: $\hat{y} = [0.01, 0.99]$
- Loss: $-1 \cdot \log(0.01) - 0 \cdot \log(0.99) \approx 4.6$

Cross-entropy heavily penalizes confident wrong predictions!

Key Information

Why Cross-Entropy Works:

- When prediction matches truth: $\log(1) = 0$ (zero loss)
- When prediction is wrong: $\log(\epsilon)$ becomes very negative (high loss)
- The cost function has a much nicer shape for optimization
- Softmax ensures predictions are never exactly 0, avoiding $\log(0)$

5.3 Keras Implementation

```
model.compile(
    optimizer='adam',                    # Optimization algorithm
    loss='categorical_crossentropy',     # Cross-entropy for multi-class
    metrics=['accuracy']                 # What to display during training
)

# For binary classification with sigmoid output:
# loss='binary_crossentropy'

# For regression:
# loss='mse' or loss='mae'
```

Listing 2: Specifying Loss and Optimizer in Keras

6 Optimization: Gradient Descent

Definition:

Gradient Descent **Gradient descent** is an iterative algorithm to find the minimum of a function by repeatedly moving in the direction of steepest descent (negative gradient).

Update Rule:

$$w_{\text{new}} = w_{\text{old}} - \alpha \nabla J(w_{\text{old}}) \tag{27}$$

Where:

- α: Learning rate (step size)
- ∇J: Gradient of the cost function

6.1 The Gradient

Definition:

Gradient The **gradient** ∇J is a vector of partial derivatives:

$$\nabla J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_d} \right] \tag{28}$$

It points in the direction of **steepest increase** of J. We move in the **opposite** direction to minimize.

6.2 Variants of Gradient Descent

Variant	Batch Size	Pros	Cons
Gradient Descent (GD)	All m samples	Stable, true gradient	Very slow for large datasets
Stochastic GD (SGD)	1 sample	Fast updates, escapes local minima	Very noisy, slow convergence
Mini-batch GD	n samples (e.g., 32)	Best of both worlds	Requires tuning batch size

Table 5: *Gradient Descent Variants*

6.2.1 Batch Gradient Descent

$$\nabla J = \frac{1}{m} \sum_{i=1}^{m} \nabla L^{(i)} \tag{29}$$

Issues:

- Must compute gradient over ALL data points before one update
- Very expensive for large datasets
- Deterministic—no randomness to escape local minima

6.2.2 Stochastic Gradient Descent (SGD)

$$\nabla J \approx \nabla L^{(i)} \quad \text{(single random sample)} \tag{30}$$

Properties:

- Very fast per update
- Highly stochastic—helps escape local minima
- Too noisy—bounces around a lot
- Cannot be easily parallelized (each step depends on previous)

6.2.3 Mini-batch Gradient Descent

$$\nabla J \approx \frac{1}{n} \sum_{i \in \text{batch}} \nabla L^{(i)} \tag{31}$$

The Sweet Spot:

- Typical batch sizes: 32, 64, 128, 256
- Still stochastic (can escape local minima)
- Parallelizable (all samples in batch use same weights)
- The "soup and spoon" analogy: You don't need a bigger spoon for a bigger pot

Key Information

The Soup and Spoon Analogy

To check if soup is salty enough:

- You don't need to drink the entire pot
- A small spoon gives you a good estimate
- The spoon size doesn't need to scale with pot size

Similarly, a mini-batch of 32-64 samples is enough to estimate the gradient, regardless of whether your dataset has 10,000 or 10 million samples!

6.3 The Non-Convex Optimization Challenge

Important:

Why Deep Learning Optimization is Hard Neural network cost functions are **highly non-convex**:

- Many local minima (not just one global minimum)
- Saddle points (flat regions with zero gradient)
- The landscape depends on architecture, data, and loss function

Implications:

- Different random initializations → different solutions
- Different students solving the same problem may get different results
- This is why we share student solutions—to see what different approaches find!

6.4 Learning Rate Considerations

Warning

Choosing the Learning Rate α:

- **Too large**: Algorithm diverges (overshoots the minimum)
- **Too small**: Convergence is extremely slow
- **Just right**: Fast convergence to a good minimum

MSE has a nice property: As you approach the minimum, the gradient gets smaller, so steps automatically become smaller. This is "self-adjusting."

MAE is trickier: The gradient magnitude is constant, so step sizes don't decrease near the minimum.

6.5 Advanced Optimizers

Optimizer	Description
SGD	Basic stochastic gradient descent
SGD + Momentum	Accumulates past gradients for smoother updates
Adagrad	Adapts learning rate per parameter based on history
RMSprop	Improves Adagrad by using moving average
Adam	Combines momentum + adaptive learning rates; default choice

Table 6: *Common Optimizers*

```
from keras.optimizers import SGD, Adam

# Basic SGD
model.compile(optimizer=SGD(learning_rate=0.01), loss='mse')

# SGD with momentum
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9), loss='mse')

# Adam (recommended default)
model.compile(optimizer=Adam(learning_rate=0.001), loss='
    categorical_crossentropy')

# Training with mini-batch
history = model.fit(
    X_train, y_train,
    batch_size=64,       # Mini-batch size
    epochs=50,           # Number of full passes through data
    validation_data=(X_val, y_val)
)
```

Listing 3: Using Different Optimizers in Keras

7 Putting It All Together: Training a Neural Network

7.1 The Complete Training Pipeline

1. **Initialize weights** randomly
2. **Forward pass**: Compute predictions $\hat{y}$
3. **Compute loss**: Measure error using loss function
4. **Backward pass**: Compute gradients via backpropagation
5. **Update weights**: Apply optimizer (e.g., Adam)
6. **Repeat** until convergence or max epochs

7.2 Complete Keras Example

```python
import keras
from keras import models, layers
from keras.optimizers import Adam

# 1. Build the model
model = models.Sequential([
    layers.Dense(16, activation='relu', input_shape=(900,)),  # Hidden layer
    layers.Dense(2, activation='softmax')                      # Output layer
])

# 2. Compile with loss, optimizer, and metrics
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# 3. Train the model
history = model.fit(
    X_train, y_train,
    batch_size=128,
    epochs=35,
    validation_data=(X_test, y_test)
)

# 4. Evaluate
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

Listing 4: Complete Neural Network Training Example

7.3 Understanding the Output

- **Training Loss**: Should decrease over epochs
- **Validation Loss**: Should also decrease; if it increases, you're overfitting
- **Accuracy**: Metric for monitoring, not for optimization

Glossary

Term	Definition
Neural Network	A model inspired by biological neurons that learns features from data
Activation Function	Non-linear function applied to neuron outputs (e.g., ReLU, Softmax)
Loss Function	Measures error for a single data point
Cost Function	Average loss over the entire dataset
Gradient Descent	Optimization algorithm that follows the negative gradient
Learning Rate	Step size for weight updates (α)
Mini-batch	Subset of data used for each gradient update
Epoch	One complete pass through the training data
One-Hot Encoding	Representing categories as binary vectors (e.g., Cat $\rightarrow$ [1,0])
Hyperparameter	Parameters set before training (e.g., learning rate, batch size)
Cross-Entropy	Loss function for classification problems
Softmax	Activation that converts logits to probabilities

One-Page Summary

CSCI E-89B Lecture 01: Neural Networks Foundations

1. Why Neural Networks?

- Manual feature engineering doesn't scale to complex data (images, text)
- NNs learn features automatically through nested non-linear transformations

2. Architecture

- Layers of neurons: Input → Hidden → Output
- Each neuron: $u = f(w_0 + w_1x_1 + w_2x_2 + \ldots)$
- Linear combination + non-linear activation

3. Activation Functions

- Hidden layers: ReLU $\max(0, z)$ or Leaky ReLU
- Binary classification output: Sigmoid $\frac{1}{1+e^{-z}}$
- Multi-class output: Softmax $\frac{e^{z_i}}{\sum e^{z_j}}$

4. Loss Functions

- Regression: MSE $(y - \hat{y})^2$ or MAE $|y - \hat{y}|$
- Classification: Cross-Entropy $-\sum y_j \log(\hat{y}_j)$
- Never use MSE for classification!

5. Optimization

- Gradient Descent: $w_{\text{new}} = w_{\text{old}} - \alpha \nabla J$
- Mini-batch GD: Best balance of speed and stability (batch size 32-64)
- Default optimizer: Adam

6. Key Formulas

$$\text{Cost Function:} \quad J(w) = \frac{1}{m} \sum_{i=1}^{m} L^{(i)}(w)$$

$$\text{Cross-Entropy:} \quad L = -\sum_{j=1}^{M} y_j \log(\hat{y}_j)$$

$$\text{Softmax:} \quad \hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

■ **Course:** CSCI E-89B: Introduction to Natural Language Processing

■ **Lecture:** Lecture 02 – TF-IDF and Recurrent Neural Networks

■ **Instructor:** Dmitry Kurochkin

■ **Objective:** Master TF-IDF for text quantification, understand RNN architecture for sequential data, and learn about LSTM/GRU for overcoming vanishing gradient problems

Contents

1 Introduction: From Text to Numbers

Lecture Overview

The fundamental challenge in Natural Language Processing is converting human language into numerical representations that machines can process. This lecture covers two major topics:

Part 1: TF-IDF

- How to quantify the importance of words in documents
- Using statistical tests (t-test) for feature selection
- Application: Patent classification

Part 2: Recurrent Neural Networks

- Processing sequential data (time series, text)
- Understanding weight sharing in RNNs
- The vanishing gradient problem and its solutions
- LSTM and GRU architectures
- Bidirectional RNNs

2 TF-IDF: Term Frequency-Inverse Document Frequency

Definition:

TF-IDF **TF-IDF** (Term Frequency-Inverse Document Frequency) is a numerical statistic that reflects how important a word is to a document within a collection (corpus). It combines two metrics:

- **TF (Term Frequency)**: How often a word appears in a specific document
- **IDF (Inverse Document Frequency)**: How rare or common a word is across all documents

2.1 The Intuition Behind TF-IDF

Key Summary

Core Idea: A word is important for a document if:

1. It appears frequently *within* that document (high TF)
2. It appears rarely *across* all documents (high IDF)

Words like "the," "is," "and" have high TF but low IDF (they appear everywhere), so their TF-IDF is low. Domain-specific terms like "transformer" in ML papers have high TF-IDF.

2.2 Computing TF-IDF Step by Step

Example:

TF-IDF Calculation Consider three documents:

1. "the cat sat on the mat" (6 words)
2. "the cat did something again" (5 words)
3. "some sentence but no cat" (5 words)

Step 1: Calculate TF for "cat" in Document 1

$$\text{TF}(\text{“cat”}, \text{Doc 1}) = \frac{\text{Count of “cat” in Doc 1}}{\text{Total words in Doc 1}} = \frac{1}{6} \tag{1}$$

Step 2: Calculate IDF for "cat"

$$\text{IDF}(\text{“cat”}) = \log\left(\frac{\text{Total documents}}{\text{Documents containing “cat”} + 1}\right) = \log\left(\frac{3}{3+1}\right) = \log(0.75) \tag{2}$$

Note: The +1 in the denominator is **smoothing** to prevent division by zero.

Step 3: Calculate TF-IDF

$$\text{TF-IDF}(\text{“cat”}, \text{Doc 1}) = \text{TF} \times \text{IDF} = \frac{1}{6} \times \log(0.75) \tag{3}$$

2.3 IDF Variants

Different libraries use slightly different IDF formulas:

Variant	Formula
Standard	$\log\left(\frac{N}{df_t}\right)$
Smoothed	$\log\left(\frac{N}{df_t+1}\right)+1$
Probabilistic	$\log\left(\frac{N-df_t}{df_t}\right)$

Table 1: *IDF Formula Variants (N = total docs, df_t = docs containing term t)*

2.4 Normalization

Warning

Always Normalize!

After computing TF-IDF vectors, normalize them to unit length:

$$\text{normalized}(\vec{v}) = \frac{\vec{v}}{||\vec{v}||} \tag{4}$$

This ensures:

- Document length doesn't bias the representation
- Cosine similarity calculations are meaningful
- Values are bounded and comparable

2.5 Train/Test Split Considerations

Important:

Critical: IDF Uses Only Training Data! When applying TF-IDF to test data:

- **TF**: Computed fresh for each test document
- **IDF**: Transferred from training data (never recomputed on test!)

Why? Computing IDF on test data would leak information from future data into your model—a form of *data leakage*.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Fit on training data ONLY
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(train_documents)

# Transform test data using trained IDF values
X_test = vectorizer.transform(test_documents)  # NOT fit_transform!
```

3 Feature Selection with t-Tests

3.1 The Dimensionality Problem

When using TF-IDF, you create one feature per unique word in your vocabulary. This can result in:

- Tens of thousands of features
- Computational expense
- The **curse of dimensionality**
- Overfitting risk

Solution: Select only the most discriminative words using statistical tests.

3.2 t-Test for Feature Selection

Definition:

Two-Sample t-Test The **t-test** determines if there's a statistically significant difference between the means of two groups.

For a word w:

- Group 1: TF-IDF scores in documents with label 1
- Group 2: TF-IDF scores in documents with label 0

Test Statistic:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5}$$

Where $\bar{x}_i$ = sample mean, s_i^2 = sample variance, n_i = sample size.

3.3 Interpreting p-Values

p-value	Meaning	Action
Very small (e.g., 0.001)	Strong evidence that the word distinguishes classes	Keep this feature
Small (e.g., 0.05)	Moderate evidence of difference	Likely keep
Large (e.g., 0.5)	No significant difference between classes	Remove this feature

Table 2: *p-Value Interpretation for Feature Selection*

3.4 Feature Selection Pipeline

```
from scipy import stats
import numpy as np

def select_features_by_ttest(X_train, y_train, feature_names, top_k=300):
```

```
    """
    Select top_k features with smallest p-values (most discriminative)
    """
    p_values = []

    for i in range(X_train.shape[1]):
        # Split by class
        class_0 = X_train[y_train == 0, i]
        class_1 = X_train[y_train == 1, i]

        # Compute t-test
        _, p_value = stats.ttest_ind(class_0, class_1)
        p_values.append(p_value)

    # Select features with smallest p-values
    p_values = np.array(p_values)
    selected_indices = np.argsort(p_values)[:top_k]

    return selected_indices, feature_names[selected_indices]
```

Listing 1: t-Test Feature Selection

Example:

Patent Classification with t-Test Feature Selection **Task:** Classify patents as automobile-related or not (1895-1935 US patents)

Process:

1. Extract TF-IDF features from patent titles/descriptions
2. Run t-test for each word comparing:
   - Group 1: Patents from known auto companies (label = 1)
   - Group 2: Random sample of patents (label = 0)
3. Keep top 300 words with smallest p-values
4. Train classifier (logistic regression, neural network)

Result: Words like "engine," "wheel," "chassis" have tiny p-values and are selected. Words like "and," "the," "of" have large p-values and are removed.

4 Recurrent Neural Networks (RNN)

4.1 Why RNNs for Sequential Data?

Key Information

The Problem with Feedforward Networks for Sequences

A standard feedforward network:

- Treats each input as independent
- Has a fixed input size
- Cannot capture temporal/sequential dependencies

For language: "The cat sat on the mat" requires understanding word *order* and *context*.

The RNN Solution:

- Maintains a "hidden state" that acts as memory
- Processes sequences one element at a time
- Shares weights across time steps

4.2 RNN Architecture

Definition:

Recurrent Neuron A **recurrent neuron** computes its output based on:

1. The current input x_t
2. The previous hidden state h_{t-1}

Formula:

$$h_t = f(W_x x_t + W_h h_{t-1} + b) \tag{6}$$

Where:

- h_t: Hidden state at time t
- x_t: Input at time t
- W_x: Input-to-hidden weights
- W_h: Hidden-to-hidden weights (recurrent weights)
- b: Bias
- f: Activation function (typically tanh)

4.3 Unrolling the RNN

When we "unroll" an RNN across time steps, we see that it's equivalent to a very deep network where each layer shares the same weights:

```
h_0=0 --> [RNN Cell] --> h_1 --> [RNN Cell] --> h_2 --> ... --> h_T
             ^                      ^                       ^
             |                      |                       |
            x_1                    x_2                     x_T
```

Important:

Weight Sharing is Key The same weights (W_x, W_h, b) are used at every time step!

Benefits:

- Dramatically reduces parameters compared to a fully connected network
- Allows processing sequences of any length
- Encodes the assumption that the same transformation applies at each step

4.4 Computing RNN Parameters

Example:

Parameter Count Example For a simple RNN layer with:

- Input dimension: $n_{in} = 2$
- Hidden state dimension (neurons): $n_h = 3$

Parameters:

$$\text{Input weights } W_x : n_{in} \times n_h = 2 \times 3 = 6 \tag{7}$$

$$\text{Recurrent weights } W_h : n_h \times n_h = 3 \times 3 = 9 \tag{8}$$

$$\text{Biases } b : n_h = 3 \tag{9}$$

$$\textbf{Total} : 6 + 9 + 3 = \boxed{18} \tag{10}$$

General Formula:

$$\text{Parameters} = (n_{in} + n_h + 1) \times n_h \tag{11}$$

4.5 Keras Implementation

```python
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense

# Input: sequences of length 200, with 2 features per time step
model = Sequential([
    SimpleRNN(3, activation='relu', input_shape=(200, 2)),
    Dense(1)  # Output layer for prediction
])

model.summary()
# SimpleRNN layer: (2 + 3 + 1) * 3 = 18 parameters
# Dense layer: 3 * 1 + 1 = 4 parameters
# Total: 22 parameters
```

Listing 2: Simple RNN in Keras

5 The Vanishing Gradient Problem

5.1 Why RNNs Struggle with Long Sequences

Warning

The Vanishing Gradient Problem

When training RNNs via backpropagation through time (BPTT), gradients must flow backward through many time steps. Consider the derivative of the loss with respect to early weights:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_T} \cdot \underbrace{\prod_{k=t}^{T-1} \frac{\partial h_{k+1}}{\partial h_k}}_{\text{product of many terms}} \cdot \frac{\partial h_t}{\partial W} \tag{12}$$

Each term $\frac{\partial h_{k+1}}{\partial h_k}$ involves the derivative of the activation function:

$$\frac{\partial h_{k+1}}{\partial h_k} = W_h^T \cdot \text{diag}(f'(z_k)) \tag{13}$$

Problem: For sigmoid/tanh, $|f'(z)| \leq 0.25$ or $|f'(z)| \leq 1$
When multiplied T times: $0.25^{100} \approx 10^{-60}$ — effectively zero!

5.2 Consequences

Problem	Cause	Effect
Vanishing Gradient	$\|f'\| < 1$ multiplied many times	Early inputs have no effect on learning
Exploding Gradient	$\|f'\| > 1$ multiplied many times	Training diverges, loss becomes NaN

Table 3: *Gradient Problems in RNNs*

Key Information

Gradient Clipping for Exploding Gradients

A simple fix for exploding gradients:

```
if gradient_norm > threshold:
    gradient = gradient * (threshold / gradient_norm)
```

This caps the gradient magnitude while preserving direction.

6 LSTM: Long Short-Term Memory

Definition:

LSTM **LSTM** (Long Short-Term Memory) is a specialized RNN architecture designed to learn long-term dependencies by using **gates** to control information flow.

Key Innovation: A separate "cell state" C_t that acts as a highway for information, allowing gradients to flow unchanged across many time steps.

6.1 The Highway Analogy

Key Summary

Think of LSTM as having an "information highway" (the cell state C_t):

- Information can travel down this highway with minimal change
- **Forget Gate**: Controls what to remove from the highway
- **Input Gate**: Controls what to add to the highway
- **Output Gate**: Controls what to output from the highway

Unlike vanilla RNN where everything gets squashed through tanh at each step, the highway allows information to persist.

6.2 LSTM Gates

Gate	Function	Computation
Forget Gate f_t	What to forget from C_{t-1}	$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$
Input Gate i_t	What to add to cell state	$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$
Cell Candidate $\tilde{C}_t$	New candidate values	$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$
Output Gate o_t	What to output	$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$

Table 4: *LSTM Gate Functions*

6.3 LSTM State Updates

$$\text{Cell State: } C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{14}$$

$$\text{Hidden State: } h_t = o_t \odot \tanh(C_t) \tag{15}$$

Where $\odot$ denotes element-wise multiplication.

6.4 Why LSTM Solves Vanishing Gradients

Important:

The Gradient Highway In the cell state update:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{16}$$

When computing $\frac{\partial C_t}{\partial C_{t-1}}$, we get f_t (the forget gate).
If $f_t \approx 1$, gradients flow through unchanged! The network learns when to remember ($f_t \to 1$) and when to forget ($f_t \to 0$).

6.5 LSTM Parameter Count

Example:

LSTM Parameters LSTM has 4 sub-networks (one for each gate), so:

$$\text{LSTM Parameters} = 4 \times (n_{in} + n_h + 1) \times n_h \tag{17}$$

For $n_{in} = 2$, $n_h = 16$:

$$4 \times (2 + 16 + 1) \times 16 = 4 \times 19 \times 16 = 1216 \text{ parameters} \tag{18}$$

7 GRU: Gated Recurrent Unit

Definition:

GRU **GRU** (Gated Recurrent Unit) is a simplified version of LSTM that:

- Merges cell state and hidden state into one
- Uses only 2 gates instead of 3
- Has fewer parameters while achieving similar performance

7.1 GRU vs LSTM

Aspect	LSTM	GRU
States	Cell state C_t + Hidden state h_t	Only hidden state h_t
Gates	3 (forget, input, output)	2 (reset, update)
Parameters	$4 \times (n_{in} + n_h + 1) \times n_h$	$3 \times (n_{in} + n_h + 1) \times n_h$
Performance	Slightly better on very long sequences	Often comparable, faster to train

Table 5: *LSTM vs GRU Comparison*

7.2 GRU Gates

$$\text{Update Gate: } z_t = \sigma(W_z[h_{t-1}, x_t]) \tag{19}$$

$$\text{Reset Gate: } r_t = \sigma(W_r[h_{t-1}, x_t]) \tag{20}$$

$$\text{Candidate: } \tilde{h}_t = \tanh(W[r_t \odot h_{t-1}, x_t]) \tag{21}$$

$$\text{Output: } h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{22}$$

8 Bidirectional RNNs

Definition:

Bidirectional RNN A **Bidirectional RNN** processes the sequence in both directions:

- **Forward**: $x_1 \to x_2 \to \ldots \to x_T$
- **Backward**: $x_T \to x_{T-1} \to \ldots \to x_1$

The outputs are concatenated: $h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$

8.1 Why Bidirectional?

Example:

Context from Both Directions Consider: "I love [MASK] because they are so fluffy."

- **Forward context**: "I love" → Could be anything
- **Backward context**: "fluffy" → Suggests animals (cats, dogs, etc.)

For tasks like sentiment analysis or translation, understanding the entire context (past AND future) often helps.

8.2 Keras Implementation

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

model = Sequential([
    Embedding(input_dim=10000, output_dim=32),
    Bidirectional(LSTM(32)),  # Output: 64 dimensions (32 forward + 32
        backward)
    Dense(1, activation='sigmoid')
])

model.summary()
# Bidirectional LSTM: 2x the parameters of regular LSTM
```

Listing 3: Bidirectional LSTM

Warning

When NOT to Use Bidirectional:

- **Real-time prediction**: You can't use future information that hasn't arrived
- **Autoregressive generation**: Generating text word-by-word
- **Time series forecasting**: Future values are unknown

Use bidirectional for tasks where the entire sequence is available at once (classification, translation, named entity recognition).

9 Training RNNs: Practical Considerations

9.1 Learning Rate and Data Scaling

Warning

Why Default Learning Rates May Fail

Default learning rates (e.g., $\alpha = 0.01$) assume your data is scaled!

If one feature ranges from 0-1 and another from 0-1,000,000:

- The loss surface becomes elongated
- Gradient descent zigzags inefficiently
- May diverge or converge very slowly

Solution: Always scale your inputs before training.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Use same scaler!
```

9.2 Dropout for Regularization

```
from keras.layers import LSTM, Dropout

model = Sequential([
    LSTM(32, return_sequences=True, input_shape=(timesteps, features)),
    Dropout(0.2),  # Drop 20% of connections
    LSTM(16),
    Dropout(0.2),
    Dense(1)
])
```

Listing 4: RNN with Dropout

9.3 Return Sequences

Setting	Output Shape	Use Case
`return_sequences=False`	`(batch, units)`	Classification, final prediction
`return_sequences=True`	`(batch, timesteps, units)`	Stacking RNN layers, seq-to-seq

Table 6: *Return Sequences Options*

Glossary

Term	Definition
TF-IDF	Term Frequency-Inverse Document Frequency; measures word importance
t-Test	Statistical test comparing means of two groups
p-Value	Probability of observing data if null hypothesis is true
RNN	Recurrent Neural Network; processes sequences with memory
Hidden State	Internal memory vector passed between time steps
Weight Sharing	Using same weights at all time steps
Vanishing Gradient	Gradients becoming too small to update early weights
LSTM	Long Short-Term Memory; RNN with gates to preserve gradients
GRU	Gated Recurrent Unit; simplified LSTM
Cell State	LSTM's long-term memory pathway
Gate	Learned mechanism to control information flow
Bidirectional	Processing sequences in both forward and backward directions
Epoch	One complete pass through the training dataset
Time Steps	Number of sequential elements in an input

One-Page Summary

CSCI E-89B Lecture 02: TF-IDF and Recurrent Neural Networks

1. TF-IDF

- TF-IDF = TF(word, doc) × IDF(word)
- High TF-IDF = frequent in document, rare across corpus
- IDF computed ONLY on training data

2. Feature Selection with t-Test

- Compare TF-IDF distributions between classes
- Small p-value → discriminative feature (keep)
- Large p-value → non-discriminative (remove)

3. RNN Architecture

- Hidden state: $h_t = f(W_x x_t + W_h h_{t-1} + b)$
- Same weights at every time step (weight sharing)
- Parameters: $(n_{in} + n_h + 1) \times n_h$

4. Vanishing Gradient Problem

- Gradients shrink exponentially over long sequences
- Result: RNNs can't learn long-term dependencies
- Solution: LSTM, GRU with gating mechanisms

5. LSTM

- Cell state C_t: information highway with minimal transformation
- 3 gates: Forget, Input, Output
- Parameters: $4 \times (n_{in} + n_h + 1) \times n_h$

6. GRU

- Simplified LSTM: merges cell/hidden state
- 2 gates: Reset, Update
- Fewer parameters, often similar performance

7. Bidirectional RNN

- Process sequence forward AND backward
- Concatenate outputs: $h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$
- Use when full sequence available at inference

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

```
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

```
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

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)

CSCI E-89B: Introduction to Natural Language Processing
Lecture 05: TF-IDF and Word Embeddings

Harvard Extension School

Fall 2024

■ **Course:** CSCI E-89B: Introduction to Natural Language Processing

■ **Week:** Lecture 05

■ **Instructor:** Dmitry Kurochkin

■ **Objective:** Master TF-IDF representation for text analysis and understand word embedding techniques including Word2Vec and GloVe

Contents

1 Quiz Review: Weight Sharing in Neural Networks

Lecture Overview

This lecture covers TF-IDF (Term Frequency-Inverse Document Frequency) as an improvement over bag-of-words, introduces the concept of word embeddings, and explores Word2Vec and GloVe as advanced embedding techniques that capture semantic relationships.

1.1 Where Does Weight Sharing Occur?

Key Summary

Weight sharing is a key technique to reduce parameters in neural networks. It occurs in:

- **Recurrent Neural Networks (RNNs)**: Same weights across all time steps
- **Convolutional Neural Networks (CNNs)**: Same filter weights at all spatial positions
- **NOT in Fully Connected Networks**: Each connection has unique weights

1.1.1 Fully Connected Networks

In a fully connected (dense) network:

- Every neuron is connected to every neuron in adjacent layers
- Each connection has its **own unique weight** w_{ij}
- No weight sharing—maximum flexibility, maximum parameters

1.1.2 Recurrent Neural Networks

When unrolled through time, RNNs use the same weights:

- Time step 1: Uses W
- Time step 2: Uses **same** W
- Time step T: Uses **same** W

Why Weight Sharing in RNNs?

We assume that the relationship between signals is **independent of time**. The way we process word 1 should be the same as how we process word 100. This reduces parameters dramatically and enables processing of variable-length sequences.

1.1.3 Convolutional Neural Networks

CNNs share weights across **spatial positions**:

- Filter applied at position (0,0): Uses weights W
- Filter applied at position (1,1): Uses **same** W
- Filter "slides" across the image with identical weights

Example: Analogy: Using the Same Eyes

When you look at the upper-left corner of an image, you use the same eyes as when you look at the lower-right corner. Similarly, CNNs use the same filter (same "eyes") at every position.

1.2 N-gram Counting

Critical: N-grams are Overlapping

When computing bigrams, tokens **overlap**:

- Text: "port assembly line reduced costs"
- Bigrams: (port, assembly), (assembly, line), (line, reduced), (reduced, costs)
- Count: $n - 1$ bigrams for n words

Bigram Vocabulary Explosion

In practice, the number of bigrams is **much larger** than unigrams:

- "Computer" appears once as a unigram
- But "computer is," "computer was," "computer does," etc. are all different bigrams
- Vocabulary grows dramatically, increasing sparsity and computation

1.3 CNN Padding and Strides

Definition: Preserving Dimensions

To preserve input dimensions in a CNN:

- **Padding**: Must be `same` (add zeros around input)
- **Strides**: Must be 1 (move one position at a time)

If strides > 1, dimensions will be reduced even with `same` padding.

2 TF-IDF: Term Frequency-Inverse Document Frequency

2.1 Motivation: Beyond Bag of Words

Bag of words treats all words equally by just counting occurrences. But consider:

- In Boston local news, "Boston" appears everywhere—it's not informative
- A rare word like "hurricane" only appears in certain documents—very informative

Key Summary

TF-IDF addresses this by:

1. Counting how often a term appears in a document (TF)
2. Discounting terms that appear in many documents (IDF)
3. Multiplying: TF $\times$ IDF

2.2 Term Frequency (TF)

Definition: Term Frequency

$$\text{TF}(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Note: This is a *frequency* (ratio), not just a count. It normalizes by document length.

Example: TF Calculation

Document 1: "cat cat dog" (3 terms)

TF(cat, doc1) $= 2/3 = 0.67$

TF(dog, doc1) $= 1/3 = 0.33$

TF(mouse, doc1) $= 0/3 = 0.00$

2.3 Inverse Document Frequency (IDF)

Definition: Inverse Document Frequency

$$\text{IDF}(t) = \ln\left(\frac{\text{Total number of documents}}{\text{Number of documents containing term } t}\right)$$

Key insight: IDF is computed from the **training corpus** and remains fixed, even when processing test documents.

2.3.1 IDF Properties

- Term in **every** document: IDF $= \ln(N/N) = \ln(1) = 0$
- Term in **half** documents: IDF $= \ln(N/(N/2)) = \ln(2) \approx 0.69$
- Term in **one** document: IDF $= \ln(N/1) = \ln(N)$ (large!)

Example: IDF Calculation

Corpus with 4 documents:

- Cat appears in 2 documents: IDF $= \ln(4/2) = 0.69$
- Dog appears in 4 documents: IDF $= \ln(4/4) = 0$ (useless!)
- Mouse appears in 3 documents: IDF $= \ln(4/3) = 0.29$

Interpretation: Cat is a good discriminator (occurs sometimes), dog is useless (occurs everywhere).

2.4 TF-IDF Computation

Definition: TF-IDF

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

High TF-IDF means: term appears frequently in this document but rarely across the corpus.

Critical: Train vs Test Data

When computing TF-IDF for test documents:

- **TF**: Computed from the test document itself
- **IDF**: Always uses the training corpus values (pre-computed)

Rationale: IDF measures how discriminative a term is across the corpus. We can't use test data to compute this—that would be data leakage!

2.5 TF-IDF Variations

2.5.1 Smoothed IDF (sklearn default)

$$\text{IDF}_{\text{smooth}}(t) = \ln\left(\frac{N+1}{df(t)+1}\right) + 1$$

- +1 in denominator: Prevents division by zero
- +1 added to result: Ensures even common words have **some** weight

2.5.2 Why Add +1 to the Result?

Without the +1, words appearing everywhere get IDF = 0, making their TF-IDF = 0. Adding 1 ensures:

- Common words still contribute (just less than rare words)
- No term is completely ignored

2.6 L2 Normalization

Definition: L2 Normalization

After computing TF-IDF values, normalize the vector to unit length:

$$\tilde{v} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{\sum_i v_i^2}}$$

This places all document vectors on a **unit hypersphere**.

2.6.1 Why Normalize?

1. **Cosine similarity becomes dot product**: For unit vectors, $\cos(\theta) = a \cdot b$
2. **Scale invariance**: Long documents don't dominate short ones
3. **Numerical stability**: All values in similar range

```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Default: lowercase=True, L2 normalization
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)

# Check: each row has L2 norm of 1
import numpy as np
print(np.linalg.norm(X[0].toarray()))  # Should be ~1.0
```

2.7 Document Similarity with Cosine

Definition: Cosine Similarity

$$\text{similarity}(d_1, d_2) = \cos(\theta) = \frac{d_1 \cdot d_2}{\|d_1\|\|d_2\|}$$

For L2-normalized vectors: similarity $= d_1 \cdot d_2$ (just dot product!)

Interpreting Cosine Similarity

$\cos(\theta) = 1$	Identical direction (very similar)
$\cos(\theta) = 0$	Orthogonal (no common terms)
$\cos(\theta) = -1$	Opposite direction (rare in TF-IDF since values ≥ 0)

3 Word Embeddings: Dense Representations

3.1 Limitations of One-Hot Encoding

One-hot encoding represents each word as a sparse vector:

- "cat" = [0, 0, 1, 0, 0, ..., 0] (10,000 dimensions)
- "dog" = [0, 0, 0, 1, 0, ..., 0] (10,000 dimensions)

Problems:

1. **High dimensionality**: Vector size = vocabulary size
2. **Sparse**: Mostly zeros, computationally wasteful
3. **No semantics**: "cat" and "dog" are as different as "cat" and "table"

3.2 What is an Embedding?

Definition: Word Embedding

A learned mapping from discrete words to dense, continuous vectors in low-dimensional space:

$$\text{embed} : \{1, 2, \ldots, V\} \to \mathbb{R}^d$$

where V = vocabulary size, d = embedding dimension (typically 50–300).

3.3 Embedding as a Neural Network Layer

Example: Embedding Layer Mechanics

Consider vocabulary size 3, embedding dimension 2:
Embedding matrix W (learnable parameters):

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

One-hot input for word 2: $x = [0, 1, 0]$
Embedding output: $x \cdot W = [w_{21}, w_{22}]$ (just row 2!)
Efficient implementation: Instead of matrix multiplication, just look up row by index.

Embedding Layer

The embedding layer is mathematically a linear layer without bias. But because input is one-hot:

- Matrix multiplication with one-hot = selecting one row
- Implementation: Direct lookup by index (much faster)
- Parameters: $V \times d$ (vocabulary size × embedding dimension)

3.4 Training Embeddings

Two approaches:

1. **Task-specific**: Train embedding layer as part of your model (classification, etc.)
2. **Pre-trained**: Use Word2Vec, GloVe, etc. trained on large corpora

Task-Specific vs Pre-trained Tradeoffs

Task-specific embeddings:

- Optimized for your specific task
- May not capture general semantics
- Synonyms might not be similar if task doesn't require it

Pre-trained embeddings (Word2Vec, GloVe):

- Capture general semantic relationships
- Transfer learning: works even with limited data
- May include biases from training corpus

4 Word2Vec: Learning from Context

4.1 The Key Idea

Key Summary

"You shall know a word by the company it keeps." —J.R. Firth (1957)
Word2Vec learns embeddings by predicting words from their context (or vice versa).

4.2 Two Architectures

4.2.1 Skip-gram: Predict Context from Word

- **Input**: Current word
- **Output**: Surrounding context words
- **Example**: Given "cat," predict ["the", "sat", "on", "mat"]

4.2.2 CBOW (Continuous Bag of Words): Predict Word from Context

- **Input**: Surrounding context words
- **Output**: Current word
- **Example**: Given ["the", "sat", "on", "mat"], predict "cat"

Example: Window Size

With window size 5:

- Consider 5 words before and 5 words after
- Total context: up to 10 words (plus target)
- Larger window = broader context, slower training

4.3 Why Word2Vec Captures Semantics

Because embeddings are trained to predict context:

- Words appearing in similar contexts get similar embeddings
- "king" and "queen" appear in similar contexts $\Rightarrow$ similar vectors
- Synonyms naturally cluster together

4.4 Word Analogies

Definition: Vector Arithmetic

Word2Vec embeddings enable semantic arithmetic:

$$\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$$

The vector $\vec{\text{man}} - \vec{\text{woman}}$ encodes the concept of "gender."

```python
from gensim.models import Word2Vec

# Train model
model = Word2Vec(sentences, vector_size=100, window=5)

# Word analogy
result = model.wv.most_similar(
    positive=['king', 'woman'],
    negative=['man'],
    topn=1
)
print(result)  # [('queen', 0.85)]
```

4.5 Finding Similar Words

```python
# Find words most similar to "destruction"
similar = model.wv.most_similar('destruction', topn=5)
# Might return: [('flood', 0.91), ('damage', 0.87), ...]

# Similarity is cosine of angle between vectors
similarity = model.wv.similarity('cat', 'dog')
# Returns value between -1 and 1
```

5 GloVe: Global Vectors for Word Representation

5.1 Motivation

Word2Vec only looks at **local** context windows. GloVe uses **global** co-occurrence statistics from the entire corpus.

Definition: Co-occurrence Matrix

Build a matrix X where X_{ij} = number of times word i appears near word j across the entire corpus.

5.2 The GloVe Objective

GloVe learns embeddings such that:

$$\vec{w}_i \cdot \vec{w}_j + b_i + b_j = \log(X_{ij})$$

Interpretation: The dot product of word vectors should approximate the log of their co-occurrence count.

Why Log Co-occurrence?

- Raw counts vary wildly (some pairs co-occur millions of times)
- Log compresses the range
- Dot product naturally represents similarity

5.3 Word2Vec vs GloVe

Aspect	Word2Vec	GloVe
Training	Local context windows	Global co-occurrence matrix
Approach	Predictive (neural network)	Count-based (matrix factorization)
Memory	Streams through corpus	Needs entire co-occurrence matrix
Speed	Slower per epoch	Faster convergence
Results	Very similar quality	Very similar quality

5.4 Using Pre-trained Embeddings

```
# Load pre-trained GloVe
import gensim.downloader as api
glove = api.load('glove-wiki-gigaword-100')

# Use like Word2Vec
similar = glove.most_similar('computer', topn=5)
```

```
vector = glove['computer']  # 100-dimensional vector
```

6 Practical Considerations

6.1 Embedding Dimension

Choosing Embedding Dimension

- **Too small (e.g., 10)**: Can't capture rich semantics
- **Too large (e.g., 1000)**: Overfitting, slow training
- **Common choices**: 50, 100, 200, 300
- **Rule of thumb**: Try 100–300 for most applications

6.2 Out-of-Vocabulary (OOV) Words

OOV Problem

Pre-trained embeddings only cover words in their training vocabulary:

- New words ("ChatGPT") won't have embeddings
- Misspellings won't be recognized
- Rare technical terms may be missing

Solutions:

- Use subword embeddings (FastText, BPE)
- Map unknown words to special "UNK" token
- Train domain-specific embeddings

6.3 Bias in Embeddings

Societal Biases

Embeddings learn from text, including biases:

- "doctor" may be closer to "man" than "woman"
- Historical texts contain outdated stereotypes
- These biases affect downstream applications

Mitigation: Debiasing techniques, careful data curation, bias auditing.

6.4 Aggregating Word Embeddings

For classification without sequence models:

```python
def document_embedding(doc, model):
    """Average word embeddings for document."""
    vectors = [model[word] for word in doc if word in model]
    if vectors:
        return np.mean(vectors, axis=0)
    return np.zeros(model.vector_size)
```

Averaging Embeddings

Simple averaging:

- Loses word order (like bag of words)
- Works surprisingly well for classification
- Fast and simple baseline

Better approaches: TF-IDF weighted averaging, use RNN/Transformer instead.

7 TF-IDF Implementation Guide

7.1 Basic TF-IDF with sklearn

```python
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "The cat sat on the mat",
    "The dog sat on the log",
    "Cats and dogs are friends"
]

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

# View vocabulary
print(vectorizer.get_feature_names_out())
# ['and', 'are', 'cat', 'cats', 'dog', 'dogs', 'friends', ...]

# View TF-IDF matrix
print(X.toarray())
```

7.2 Customizing TfidfVectorizer

```python
vectorizer = TfidfVectorizer(
    lowercase=True,           # Convert to lowercase
    stop_words='english',     # Remove English stop words
    ngram_range=(1, 2),       # Unigrams and bigrams
    max_features=10000,       # Top 10k features only
    min_df=2,                 # Ignore terms in < 2 docs
    max_df=0.95,              # Ignore terms in > 95% docs
    norm='l2'                 # L2 normalization (default)
)
```

7.3 Using with Preprocessing

```python
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

stemmer = PorterStemmer()

def preprocess(text):
    tokens = word_tokenize(text.lower())
    stems = [stemmer.stem(t) for t in tokens]
    return ' '.join(stems)

# Preprocess documents
```

```
processed = [preprocess(doc) for doc in corpus]

# Then apply TF-IDF
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(processed)
```

8 One-Page Summary

TF-IDF

Term Frequency: $\text{TF}(t, d) = \frac{\text{count}(t,d)}{\text{len}(d)}$
Inverse Document Frequency: $\text{IDF}(t) = \ln\left(\frac{N}{df(t)}\right)$
TF-IDF: $\text{TF}(t, d) \times \text{IDF}(t)$
Key insight: High TF-IDF = frequent in document, rare in corpus

Word Embeddings

One-hot: Sparse, high-dimensional, no semantics
Embedding: Dense, low-dimensional, captures meaning
Parameters: vocabulary_size × embedding_dim

Word2Vec

Skip-gram: Predict context from word
CBOW: Predict word from context
Key property: $\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$

GloVe

Approach: Learn from global co-occurrence matrix
Objective: $\vec{w_i} \cdot \vec{w_j} \approx \log(\text{co-occurrence})$
Result: Similar quality to Word2Vec

Cosine Similarity

$\cos(\theta) = \frac{a \cdot b}{\|a\|\|b\|}$
For unit vectors: $\cos(\theta) = a \cdot b$
Used to measure word/document similarity

9 Glossary

Term	**Definition**
CBOW	Continuous Bag of Words: Word2Vec variant predicting word from context
Co-occurrence Matrix	Matrix counting how often word pairs appear together
Cosine Similarity	Similarity measure based on angle between vectors
Document Frequency	Number of documents containing a term
Embedding	Dense vector representation of discrete items
GloVe	Global Vectors: embedding method using co-occurrence statistics
IDF	Inverse Document Frequency: log(total docs / docs with term)
L2 Normalization	Scaling vector to unit length using Euclidean norm
One-hot Encoding	Sparse vector with single 1 indicating category
Skip-gram	Word2Vec variant predicting context from word
TF	Term Frequency: count of term / document length
TF-IDF	Term Frequency-Inverse Document Frequency
Weight Sharing	Using same weights at different positions (RNN, CNN)
Window Size	Number of context words considered around target
Word2Vec	Neural embedding method learning from local context

CSCI E-89B: Introduction to Natural Language Processing
Lecture 06: Character Embeddings and Autoencoders

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-89B: Introduction to Natural Language Processing
■ **Week:**	Lecture 06
■ **Instructor:**	Dmitry Kurochkin
■ **Objective:**	Understand character-level embeddings, autoencoder architectures for dimensionality reduction, and sparse/variational autoencoders

Contents

1 Quiz Review: TF-IDF and Embeddings

Lecture Overview

This lecture explores character-level embeddings as an alternative to word embeddings, then introduces autoencoders as a powerful technique for unsupervised representation learning. We cover standard autoencoders, stacked (deep) autoencoders, sparse autoencoders, and variational autoencoders.

1.1 TF-IDF Computation Review

Example: TF-IDF Calculation

Documents:

- Doc 1: "apple banana apple" (3 terms)
- Doc 2: "banana cherry" (2 terms)
- Doc 3: "apple cherry" (2 terms)

TF for "apple" in Doc 1:

$$\text{TF}(\text{apple}, D_1) = \frac{2}{3} \approx 0.667$$

IDF for "apple" (appears in 2 of 3 docs):

$$\text{IDF}(\text{apple}) = \ln\left(\frac{3}{2}\right) \approx 0.405$$

TF-IDF:

$$\text{TF-IDF} = 0.667 \times 0.405 \approx 0.270$$

Logarithm Base Doesn't Matter

Whether using ln (natural log) or $\log_{10}$:

$$\log_{10}(x) = \frac{\ln(x)}{\ln(10)} \approx 0.434 \times \ln(x)$$

It's just a constant multiplier. After L2 normalization, results are identical!

1.2 Static Embedding Issues

Semantic Drift

Static word embeddings face challenges when language evolves:

- Words acquire new meanings over time
- Slang and technical terms emerge
- Cultural contexts shift

Example: "viral" meant only disease-related before social media.

1.3 Cosine Similarity Review

Definition: Cosine Similarity

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|} = \frac{\sum_i a_i b_i}{\sqrt{\sum_i a_i^2}\sqrt{\sum_i b_i^2}}$$

Interpretation:

- $\cos(\theta) = 1$: Identical direction (most similar)
- $\cos(\theta) = 0$: Orthogonal (no similarity)
- Higher cosine = smaller angle = more similar

Example: Finding Most Similar Word

Given embeddings:

- cat = [2, 3]
- dog = [5, 7]
- mouse = [1, 2]

cat-dog similarity: $\frac{2\times5+3\times7}{\sqrt{13}\times\sqrt{74}} \approx 0.9948$
cat-mouse similarity: $\frac{2\times1+3\times2}{\sqrt{13}\times\sqrt{5}} \approx 0.9922$
Dog is closer to cat (despite being farther in Euclidean distance!)

2 Character-Level Embeddings

2.1 Motivation

Word embeddings face limitations:

- **Out-of-vocabulary (OOV) words**: Misspellings, new words
- **Morphologically rich languages**: Turkish, Finnish (15+ word forms)
- **Spell checking**: Need to recognize character-level patterns

Key Summary

Character embeddings represent text at the character level:

- Each character is a token
- Vocabulary is tiny (26 letters + punctuation $\approx$ 40)
- Naturally handles OOV, misspellings, any language

2.2 Character vs Word Embeddings

Aspect	Word Embeddings	Character Embeddings
Vocabulary	Large (10k–100k)	Small (~40)
OOV handling	Problematic	Natural
Embedding dim	100–300	8–20 (sufficient)
Sequence length	# words	# characters (much longer)
Training data	Moderate	More needed

2.3 When to Use Character Embeddings

Best Applications

- **Spell checking/correction**: Character-level patterns matter
- **Named Entity Recognition**: Recognize unseen names
- **Morphologically rich languages**: Turkish, Finnish, Arabic
- **Social media text**: Slang, misspellings, creative spelling

2.4 Implementation

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

texts = ["Hello world", "machine learning", "deep learning"]
labels = [1, 0, 0]

# Character-level tokenization
```

```
tokenizer = Tokenizer(char_level=True)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Pad sequences (add zeros to make equal length)
padded = pad_sequences(sequences, padding='post')

# Vocabulary size (characters + padding token)
vocab_size = len(tokenizer.word_index) + 1  # ~20 characters

# Build model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

model = Sequential([
    Embedding(vocab_size, 8),  # Small embedding dim for characters
    LSTM(32),
    Dense(1, activation='sigmoid')
])
```

Embedding Dimension for Characters

Since vocabulary is small (∼40 characters), embedding dimension can be small too:

- Word embeddings: 100–300 dimensions
- Character embeddings: 8–20 dimensions

2.5 Hybrid Approaches

Combine character and word embeddings:

1. Process characters through RNN → word representation
2. Concatenate with standard word embedding
3. Use combined representation for downstream task

Benefits of Hybrid

- Word embedding captures semantic meaning
- Character embedding handles morphology and OOV
- Best of both worlds!

3 Autoencoders: Learning Efficient Representations

3.1 The Compression Intuition

Example: Memory and Patterns

Which is easier to remember?

Sequence A: 7, 3, 9, 1, 5, 8, 2, 6, 4, 0

Sequence B: 70, 68, 66, 64, 62, 60, 58, 56, 54, 52

Sequence B has a **pattern** (subtract 2 each time). We can encode it as: "start at 70, subtract 2"—much more efficient!

3.2 What is an Autoencoder?

Definition: Autoencoder

A neural network trained to reconstruct its input through a **bottleneck**:

- **Encoder**: Compresses input to lower-dimensional representation
- **Bottleneck**: The compressed representation (encodings/codings)
- **Decoder**: Reconstructs input from compressed representation

Loss function: Reconstruction loss $= \|x - \hat{x}\|^2$

3.3 Architecture

Autoencoder Structure

Input (784) $\rightarrow$ Encoder $\rightarrow$ $\boxed{\text{Bottleneck (30)}}$ $\rightarrow$ Decoder $\rightarrow$ Output (784)

```python
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

# Encoder
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(30, activation='relu')(encoded)  # Bottleneck

# Decoder
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mse')
```

3.4 Key Concepts

Critical: Undercomplete Autoencoder

When bottleneck dimension $<$ input dimension:

- Network is forced to learn efficient representations
- Similar to PCA (Principal Component Analysis) for linear activations
- Captures the most important features

3.4.1 Encoder and Decoder

- **Encoder**: Maps input x to encoding c
- **Decoder**: Maps encoding c to reconstruction $\hat{x}$
- **Encodings**: The compressed representation (bottleneck values)

3.4.2 After Training

1. Train full autoencoder on unlabeled data
2. **Throw away decoder**
3. Use encoder to create compressed representations
4. Feed compressed representations to classifier (much fewer parameters!)

3.5 Why Autoencoders Work

Dimensionality Reduction without Labels

Autoencoders learn to:

- Keep important information (needed to reconstruct)
- Discard noise and irrelevant details
- Create clustered representations (similar inputs $\rightarrow$ similar encodings)

Key benefit: No labels needed! Train on millions of unlabeled images.

4 Autoencoder Applications

4.1 Dimensionality Reduction for Classification

Key Summary

Problem: Limited labeled data, high-dimensional input

Solution:

1. Train autoencoder on large unlabeled dataset
2. Use encoder to compress inputs (e.g., 784 → 30)
3. Train small classifier on compressed representations
4. Much fewer parameters = needs much less labeled data!

Example: Fashion MNIST

- Input: 28×28 = 784 pixels
- Bottleneck: 30 encodings
- Result: T-shirts cluster together, shoes cluster together, etc.
- After t-SNE visualization: Clear separation of classes!

4.2 Denoising Autoencoders

Definition: Denoising Autoencoder

Train to reconstruct **clean** input from **corrupted** input:

- Input: Noisy/corrupted signal
- Target output: Original clean signal
- Network learns to ignore/remove noise

Applications:

- **Image restoration**: Remove scratches, artifacts
- **Audio denoising**: Clean up recordings
- **Text correction**: Fix spelling/grammar errors

```
# Add noise to training data
noise_factor = 0.3
x_train_noisy = x_train + noise_factor * np.random.normal(
    size=x_train.shape
)

# Train: noisy input -> clean output
autoencoder.fit(x_train_noisy, x_train, epochs=10)
```

4.3 t-SNE for Visualization

Definition: t-SNE

t-Distributed Stochastic Neighbor Embedding: Visualization technique that maps high-dimensional data to 2D while preserving local structure.

NOT for dimensionality reduction in pipelines—only for visualization!

t-SNE Properties

- Non-deterministic (different runs give different results)
- Preserves local neighborhoods
- Distances between clusters may not be meaningful
- Excellent for visualizing autoencoder encodings

5 Stacked (Deep) Autoencoders

5.1 The Deep Network Problem

Deep autoencoders have many layers:

$$784 \to 200 \to 100 \to 30 \to 100 \to 200 \to 784$$

Problem: Deep networks are hard to train:

- Vanishing gradients
- Different layers have vastly different gradient scales
- Optimization landscape has stretched contours

5.2 Layer-wise Pretraining

Key Summary

Train one layer at a time:
Phase 1: Train shallow autoencoder $784 \to 200 \to 784$
Phase 2: Freeze Phase 1 weights. Train $200 \to 100 \to 200$
Phase 3: Freeze Phases 1-2. Train $100 \to 30 \to 100$
Result: Never train deep network from scratch!

```
# Phase 1: Train first layer
encoder1 = Dense(200, activation='relu')
decoder1 = Dense(784, activation='sigmoid')
ae1 = Model(input, decoder1(encoder1(input)))
ae1.fit(x_train, x_train)

# Phase 2: Freeze encoder1, train second layer
encoder1.trainable = False
encoder2 = Dense(100, activation='relu')
decoder2 = Dense(200, activation='relu')
# ... continue stacking
```

Modern Alternative

Layer-wise pretraining was essential before modern techniques:

- Adam optimizer handles scale differences better
- Batch normalization stabilizes training
- Skip connections (ResNet) enable very deep networks

Today: Often train deep autoencoders end-to-end.

6 Sparse Autoencoders

6.1 The Heterogeneous Data Problem

When Bottleneck Isn't Enough

For diverse datasets (digits + animals + cars):

- 30 encodings might not capture all variation
- Need more encodings (e.g., 300)
- But with 300 encodings, **all** neurons activate for every input
- No specialization: digits use same neurons as animals

6.2 Sparsity Constraint

Definition: Sparse Autoencoder

Add regularization to encourage only a **few** encodings to be active:

$$\mathcal{L} = \|x - \hat{x}\|^2 + \lambda \sum_i |c_i|$$

Effect: Most encodings are zero; only relevant ones activate.

Example: Intuition

With 300 encodings and 10% sparsity target:

- Digits activate encodings 1–30
- Animals activate encodings 31–60
- Cars activate encodings 61–90
- Each category has its own "experts"

6.3 L1 Activity Regularization

```
from tensorflow.keras.regularizers import l1

# L1 regularization encourages zeros
encoding_layer = Dense(
    300,
    activation='relu',
    activity_regularizer=l1(1e-3)  # lambda = 0.001
)
```

6.4 KL Divergence Sparsity

More sophisticated approach: Match average activation to target sparsity.

Definition: KL Divergence for Sparsity

$$D_{KL}(p\|\hat{p}) = p \ln \frac{p}{\hat{p}} + (1-p) \ln \frac{1-p}{1-\hat{p}}$$

Where:

- p = target sparsity (e.g., 0.1 = 10% neurons active)
- $\hat{p}$ = actual average activation

```python
import tensorflow.keras.backend as K

def kl_divergence(target, actual):
    return target * K.log(target / actual) + \
           (1 - target) * K.log((1 - target) / (1 - actual))

class KLDivergenceRegularizer:
    def __init__(self, target=0.1, weight=0.05):
        self.target = target
        self.weight = weight

    def __call__(self, activations):
        mean_activations = K.mean(activations, axis=0)
        return self.weight * K.sum(
            kl_divergence(self.target, mean_activations)
        )
```

Why KL Divergence?

L1 makes encodings small but doesn't guarantee sparsity.

KL divergence:

- Explicitly targets specific sparsity level
- Steeper gradient for faster convergence
- More control over sparsity percentage

7 Variational Autoencoders (VAE)

7.1 The Structured Space Problem

Standard Autoencoder Limitation

In standard autoencoders, the encoding space is **unstructured**:

- Small movements in encoding space may produce garbage
- Gaps between encodings don't correspond to valid inputs
- Can't smoothly interpolate between images
- Can't generate new, meaningful samples

7.2 VAE Key Idea

Definition: Variational Autoencoder

During training, add **random noise** to encodings:

1. Encoder outputs μ (mean) and σ (std dev)
2. Sample encoding: $c = \mu + \sigma \cdot \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$
3. Decoder reconstructs from noisy encoding

Result: Encodings become probability distributions, not points!

7.3 Why Noise Helps

Structured Latent Space

Adding noise during training:

- Forces decoder to handle nearby points
- Fills gaps in encoding space with valid reconstructions
- Creates smooth transitions between classes
- Enables meaningful interpolation and generation

7.4 The Sigma Problem

If σ is trainable, network will set $\sigma \to 0$ to minimize reconstruction loss!

Solution: Add KL divergence to encourage $\sigma \approx 1$:

$$\mathcal{L} = \|x - \hat{x}\|^2 + D_{KL}(\mathcal{N}(\mu, \sigma) \| \mathcal{N}(0, 1))$$

The KL term simplifies to:

$$D_{KL} = -\frac{1}{2} \sum_j \left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2\right)$$

7.5 VAE Applications

- **Image generation**: Sample from latent space
- **Image editing**: Move in latent space (e.g., add smile)
- **Interpolation**: Smooth transitions between images
- **Anomaly detection**: Unusual inputs have high reconstruction loss

8 One-Page Summary

Character Embeddings

When to use: OOV words, spell checking, morphologically rich languages
Advantages: Small vocabulary, handles any text
Embedding dim: 8–20 (vs 100–300 for words)

Autoencoder Architecture

Input $\xrightarrow{\text{Encoder}}$ Bottleneck (Encodings) $\xrightarrow{\text{Decoder}}$ Reconstruction
Loss: $\|x - \hat{x}\|^2$ (reconstruction loss)
Key idea: Force network to compress through bottleneck

Autoencoder Types

Standard	Bottleneck forces compression
Denoising	Corrupt input, reconstruct clean
Sparse	L1 or KL divergence for sparsity
Variational	Add noise, structured latent space

Sparse Autoencoder

Problem: Diverse data needs many encodings
Solution: Regularize to activate only few encodings
L1: $\mathcal{L} = \text{recon} + \lambda \sum |c_i|$
KL: Match average activation to target sparsity

VAE Key Points

Standard AE problem: Unstructured latent space
VAE solution: Encode as distribution (μ, σ), sample with noise
KL term: Prevents $\sigma \to 0$, encourages standard normal
Result: Smooth, structured latent space for generation

9 Glossary

Term	**Definition**
Activity Regularization	Penalizing large activation values to encourage sparsity
Autoencoder	Neural network trained to reconstruct input through bottleneck
Bottleneck	Layer with fewer neurons than input, forcing compression
Character Embedding	Dense vector representation for individual characters
Denoising Autoencoder	AE trained to reconstruct clean input from corrupted input
Encoder	Part of autoencoder that compresses input to encoding
Decoder	Part of autoencoder that reconstructs from encoding
Encodings/Codings	The compressed representation at the bottleneck
KL Divergence	Measure of difference between two probability distributions
Latent Space	The space of encodings/compressed representations
Reconstruction Loss	$\|x - \hat{x}\|^2$, measures how well input is reconstructed
Semantic Drift	Change in word meanings over time
Sparse Autoencoder	AE with regularization encouraging few active encodings
Stacked Autoencoder	Deep autoencoder with multiple hidden layers
t-SNE	Visualization technique for high-dimensional data
Undercomplete	Autoencoder where bottleneck dim $<$ input dim
Variational AE	AE that encodes inputs as distributions, enabling generation

Lecture Information

Course: CSCI E-89B: Natural Language Processing
Lecture: Lecture 7
Topic: Latent Dirichlet Allocation and Topic Modeling
Date: Fall 2024

Contents

1 Quiz Review: Autoencoders Revisited

Overview

This lecture begins with a review of autoencoder concepts before introducing topic modeling—a powerful unsupervised learning technique for discovering hidden themes in document collections.

1.1 One-Hot Encoding Disadvantages

One-Hot Encoding Problems

One-hot encoding has two major disadvantages:

1. **Sparsity**: The representation contains many zeros
2. **High dimensionality**: As a direct result of sparsity

Why sparsity is problematic:

- When computing linear combinations, you perform many operations with zeros
- $0 \times$ something contributes nothing but still requires computation
- Information is represented inefficiently
- This is precisely why we use embeddings instead

1.2 Undercomplete Autoencoders

Undercomplete Autoencoder

An autoencoder is called **undercomplete** when the dimensionality of the representation (encoding) is **lower** than the dimensionality of the input. This creates a bottleneck that forces compression.

Understanding "Undercomplete"

The terminology makes intuitive sense:

- **Complete**: 4 dimensions in $\rightarrow$ 4 dimensions in middle $\rightarrow$ 4 dimensions out
- **Undercomplete**: 4 dimensions in $\rightarrow$ 2 dimensions in middle $\rightarrow$ 4 dimensions out
- **Overcomplete**: 3 dimensions in $\rightarrow$ 7 dimensions in middle $\rightarrow$ 3 dimensions out

If you have a 3-dimensional dataset and map it to 2 dimensions, your representation is "undercomplete" because you cannot fully represent 3D data in 2D without some loss. You're taking projections rather than keeping complete information.

1.3 Stacked Autoencoders

Stacked Autoencoder

A **stacked autoencoder** (also called a deep autoencoder) has multiple hidden layers. "Stacked" by definition implies depth—at least 2 hidden layers.

- **Shallow network**: 1 hidden layer only
- **Deep network**: 2 or more hidden layers

1.3.1 Layer-wise Training

Important

Layers of stacked autoencoders do not need to be trained together. You can train them one at a time using a "sandwich" approach:

Phase 1: Train only input $\rightarrow$ hidden$_1$ $\rightarrow$ output (no middle layers)

Phase 2: Freeze coefficients, add hidden$_2$ between hidden$_1$ and output

Phase 3: Train middle part while keeping frozen layers fixed

Phase 4: Continue stacking more layers

1.3.2 Why Layer-wise Training?

The motivation relates to gradient descent optimization:

The Scale Problem

Weights W for different layers can be on different scales:

- W_1 (near input) and W_{200} (near output) may differ vastly in magnitude
- Cost function level curves become stretched (elliptical rather than circular)
- The gradient $-\alpha \nabla J$ points away from the true minimum
- Training may diverge or take forever

Modern solutions to the scale problem:

1. **Adam optimizer**: Adjusts gradient direction based on local curvature
2. **Batch normalization**: Normalizes signals locally, mitigating weight scale issues
3. **Shortcut connections**: Connect layers to later layers, allowing signals to bypass problematic areas

Modern Practice

Due to these techniques, the scale problem is largely solved today. People often train deep autoencoders with all layers at once. However, the phase-wise approach remains elegant and worth knowing.

1.4 Autoencoders for Sequences

Important

Autoencoders **can** be used for sequences. A sequence of words (sentence) can be compressed into a single vector (the bottleneck), which has no time component.

$$\text{Sequence } \underbrace{[x_1, x_2, \ldots, x_T]}_{\text{vectors}} \xrightarrow{\text{Encoder}} \underbrace{z}_{\text{single vector}} \xrightarrow{\text{Decoder}} \text{Reconstructed sequence}$$

1.4.1 How Small Should the Bottleneck Be?

This is a fundamental question with no universal answer:

Goal Determines Architecture

If your goal is perfect reconstruction:

- Don't squeeze at all!
- Why would you compress if you need to recover everything?
- Just use input = output directly

If your goal is feature extraction for downstream tasks:

- Use the encoder output as input to a classifier
- The optimal bottleneck size depends on classification performance
- Iterate: try 2D, 5D, 10D representations and evaluate which works best

Practical Consideration

If you have millions of input features, feeding them directly to a classifier creates millions of parameters. You may want to compress first to:

- Reduce parameter count
- Avoid getting stuck in local minima
- Make optimization tractable

2 Introduction to Topic Modeling

Overview

Topic modeling is an **unsupervised learning** approach for discovering hidden themes in document collections. Unlike clustering, documents can belong to **multiple topics** with different proportions.

2.1 Motivation: Why Not Clustering?

Traditional Clustering

Standard clustering algorithms (K-means, hierarchical) assign each data point to **exactly one cluster**:

- Students belong to "students" cluster
- Retired people belong to "retired" cluster
- Points don't overlap between clusters

The problem with text documents:

Documents Are Different

In text, topics naturally **overlap**:

- A news article may start discussing economics, shift to politics, and mention artificial intelligence
- The same document contains multiple themes
- Hard assignment to one cluster loses important information

Document as Topic Mixture

Consider Document 2 in a corpus:

- Topic 1 (Economics): 60%
- Topic 2 (Politics): 30%
- Topic 3 (AI): 10%

This soft assignment captures the document's multi-topical nature far better than forcing it into one category.

2.2 Topic Modeling Methods

Three main approaches for topic modeling:

Method	Type	Characteristics
LDA	Probabilistic	Assumes stochastic text generation
NMF	Deterministic	Matrix factorization approach
STM	Probabilistic	LDA + covariates (metadata)

Why Social Scientists Love Topic Models

Topic modeling is popular in political science, government departments, and social sciences:

- Extract information from large text corpora computationally
- Discover biases and patterns

- Correlate topics with covariates (gender, source, etc.)

2.3 Latent Topics

Latent Topics

Topics are called "latent" because:

- We don't define topics upfront (e.g., "economics")
- The model discovers topics from data
- We only assign labels **after** processing
- Labels come from examining top-weighted documents/words per topic

Important

The number of topics is a **hyperparameter** you must specify beforehand. If you say 3 topics, the model will find exactly 3 topics—it won't tell you the "true" number.

3 Maximum Likelihood Estimation

Overview

Before diving into LDA, we need to understand **maximum likelihood estimation (MLE)**—the framework used to recover model parameters from observed data.

3.1 The MLE Framework

Maximum Likelihood Estimation

MLE finds parameters that maximize the probability of observing the data we actually observed:

$$\hat{\theta} = \arg\max_{\theta} P(\text{data}|\theta)$$

Simple Example: Normal Distribution

Suppose we observe data and create a histogram. We assume data comes from a normal distribution with unknown μ and σ^2.
Step 1: Try parameters (Model A): μ far left of data

- Probability of observing our data given Model A is very small
- Likelihood ≈ 0

Step 2: Try parameters (Model B): μ closer to data center

- Probability is higher
- Likelihood increases

Step 3: Try parameters (Model C): μ at data mean, appropriate σ^2

- Probability is highest
- This is our MLE solution!

Step 4: Try parameters (Model D): Overshoot μ

- Probability decreases again

Conceptual diagram: Scanning through parameter space to find maximum likelihood

Important

MLE is incredibly powerful:

- Works with many parameters (dozens or more)
- Used in time series, LDA, STM, and countless applications
- Very natural approach: find parameters that make observed data most probable

4 Latent Dirichlet Allocation (LDA)

Overview

LDA models each document as a **mixture of topics**, where each topic is a distribution over words. Unlike clustering, LDA provides soft (probabilistic) topic assignments.

4.1 The Generative Model

LDA assumes text is generated by the following process:

LDA Text Generation Process

For document m:

Step 1: Choose number of words $N \sim \text{Poisson}(\xi)$

Step 2: Choose topic proportions $\theta_m \sim \text{Dirichlet}(\alpha)$

Step 3: For each word $n = 1, \ldots, N$:

(a) Choose topic $z_n \sim \text{Multinomial}(\theta_m)$

(b) Choose word $w_n \sim \text{Multinomial}(\beta_{z_n})$

4.2 Understanding Each Component

4.2.1 Poisson Distribution for Document Length

Poisson Distribution

The Poisson distribution models count data:

$$P(N = k) = \frac{\xi^k e^{-\xi}}{k!}$$

where ξ is both the mean and variance. If documents average 25 words, $\xi \approx 25$.

4.2.2 Dirichlet Distribution for Topic Proportions

Dirichlet Distribution

The Dirichlet distribution produces vectors of probabilities that sum to 1. For K topics:

$$\theta_m = (\theta_{m,1}, \theta_{m,2}, \ldots, \theta_{m,K}) \quad \text{where} \quad \sum_k \theta_{m,k} = 1$$

Example for 3 topics: $\theta_m = (0.6, 0.3, 0.1)$ means:

- 60% Topic 1 (Economics)
- 30% Topic 2 (Politics)
- 10% Topic 3 (AI)

Dirichlet as Generalized Beta

- For 2 topics: Dirichlet reduces to the Beta distribution
- For $K > 2$ topics: Dirichlet is the natural generalization
- Parameter α (vector) controls the shape of the distribution

4.2.3 Multinomial Distribution for Topic and Word Selection

Multinomial Selection

Given probabilities, multinomial sampling selects one category:

- $z_n \sim \text{Multinomial}(\theta_m)$: Select topic for word n
- $w_n \sim \text{Multinomial}(\beta_{z_n})$: Select word from chosen topic's vocabulary distribution

Concrete Word Generation

Given $\theta_m = (0.1, 0.2, 0.6)$ for three topics:

1. Draw topic: Most likely Topic 3 (60%), but could be Topic 2 (20% chance)
2. Suppose Topic 2 is selected
3. Draw word from β_2: Each topic has its own word distribution

4. Word "maximization" is selected from Topic 2's distribution

This is repeated independently for each word position.

4.3 Key Assumptions of LDA

Bag of Words Assumption

LDA assumes **no word order**:

- Words are generated independently
- "maximization" doesn't depend on what came before
- Shuffling words in a document doesn't change its LDA representation
- Generated text wouldn't be grammatical—that's not the point!

Important

LDA is not for generating readable text. It's for:

- Discovering what topics a document discusses
- Finding topic proportions for each document
- Identifying words associated with each topic

4.4 The EM Algorithm

Expectation-Maximization (EM)

LDA uses the EM algorithm because topic assignments z_n are **latent variables** (not observed):

1. **E-step**: Estimate expected values of latent variables z_n given current parameters
2. **M-step**: Maximize likelihood given these expected values
3. Iterate until convergence

Why EM?

We observe documents (sequences of words) but not:

- Which topic each word came from (z_n)
- The true topic proportions (θ_m)
- The topic-word distributions (β_k)

EM handles this missing information elegantly.

5 LDA Implementation in Python

5.1 Using scikit-learn

```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# Example documents
documents = [
    "Cats are wonderful pets",
    "Cats and dogs are popular animals",
    "Dogs enjoy long walks",
    "Walks in the park are relaxing",
    # ... more documents
]

# Create document-term matrix
vectorizer = CountVectorizer(stop_words='english')
X = vectorizer.fit_transform(documents)

# Fit LDA
lda = LatentDirichletAllocation(
    n_components=2,       # Number of topics
    random_state=42       # For reproducibility
)
lda.fit(X)

# Get topic-document distribution
doc_topic_dist = lda.transform(X)
print(doc_topic_dist)
# Output: [[0.05, 0.95], [0.06, 0.94], [0.93, 0.07], ...]
```

Listing 1: LDA with scikit-learn

Interpreting Results

Output $[0.05, 0.95]$ means:

- 5% contribution from Topic 1
- 95% contribution from Topic 2

Since this document is mostly Topic 2, and it's about cats, Topic 2 is likely the "cats" topic.

5.2 Extracting Top Words per Topic

```python
def display_topics(model, feature_names, num_top_words=5):
    for topic_idx, topic in enumerate(model.components_):
        top_words_idx = topic.argsort()[:-num_top_words-1:-1]
        top_words = [feature_names[i] for i in top_words_idx]
        print(f"Topic {topic_idx}: {', '.join(top_words)}")

feature_names = vectorizer.get_feature_names_out()
display_topics(lda, feature_names)

# Output:
# Topic 0: dogs, enjoy, long, walks, exploring
# Topic 1: cats, purr, love, climb, trees
```

Listing 2: Display top words for each topic

5.3 Document-Based Topic Interpretation

Important

A better way to understand topics: look at documents with highest topic prevalence, not just top words.

```
# For each topic, find documents with highest prevalence
for topic_idx in range(n_topics):
    # Sort documents by topic prevalence
    top_docs = doc_topic_dist[:, topic_idx].argsort()[::-1][:2]

    print(f"\nTopic {topic_idx} - Top documents:")
    for doc_idx in top_docs:
        prevalence = doc_topic_dist[doc_idx, topic_idx]
        print(f"  [{prevalence:.2%}] {documents[doc_idx]}")
```

Listing 3: Find most representative documents

Why Document-Based Interpretation?

Looking at actual documents is more reliable than top words because:

- Top words may be ambiguous or share meanings
- Documents provide context
- 95% prevalence means the document is almost entirely about that topic
- Reading the document tells you definitively what the topic represents

6 Non-negative Matrix Factorization (NMF)

Overview

NMF is a **deterministic** alternative to LDA. It uses linear algebra rather than probabilistic modeling to decompose documents into topics.

6.1 The Matrix Factorization Idea

NMF Decomposition

Given document-term matrix V (documents $\times$ vocabulary):

$$V \approx W \cdot H$$

where:

- W: Document-topic matrix (documents $\times$ topics)
- H: Topic-word matrix (topics $\times$ vocabulary)
- All entries in W and H are ≥ 0 (non-negative)

6.2 Matrix Multiplication Review

Matrix Multiplication Example

$$W = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad H = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

$$V = W \cdot H = \begin{pmatrix} 1 \cdot 1 + 2 \cdot 0 & 1 \cdot 1 + 2 \cdot 0 & 1 \cdot 0 + 2 \cdot 2 \\ 3 \cdot 1 + 4 \cdot 0 & 3 \cdot 1 + 4 \cdot 0 & 3 \cdot 0 + 4 \cdot 2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 4 \\ 3 & 3 & 8 \end{pmatrix}$$

Rules:

- $(m \times n) \cdot (n \times p) = (m \times p)$
- Element (i, j) = dot product of row i from first matrix and column j from second

6.3 NMF for Topic Modeling

Concrete NMF Example

Three documents with vocabulary [cats, dogs, bark, purr, growl]:

- Doc 1: "cats meow" → [1, 0, 0, 1, 0]
- Doc 2: "dogs bark" → [0, 1, 1, 0, 0]
- Doc 3: "cats purr dogs growl" → [1, 1, 0, 1, 1]

$$V = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

NMF finds:

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0.5 & 0.5 \end{pmatrix}, \quad H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Interpretation:

- Topic 1: cats-related (cats, purr)
- Topic 2: dogs-related (dogs, bark, growl)
- Doc 3: 50% Topic 1, 50% Topic 2

6.4 NMF vs LDA

Aspect	LDA	NMF
Approach	Probabilistic	Deterministic
Algorithm	EM (iterative)	Matrix factorization
Reproducibility	Stochastic (varies)	Deterministic (same result)
Interpretability	Often better	Harder to interpret W values
Speed	Slower	Faster
Constraints	Probabilities sum to 1	Only non-negativity

NMF Interpretation Challenge

NMF's W matrix entries are just non-negative numbers, not probabilities:

- Values like 2.0 and 5.0 are valid
- Harder to say "60% Topic 1"
- Requires normalization for probability interpretation

6.5 NMF Implementation

```python
from sklearn.decomposition import NMF
from sklearn.feature_extraction.text import CountVectorizer

# Same document-term matrix X from before
nmf = NMF(n_components=2, random_state=42)
W = nmf.fit_transform(X)  # Document-topic matrix
H = nmf.components_         # Topic-word matrix

# Display top words per topic
for idx, topic in enumerate(H):
    top_words = [feature_names[i] for i in topic.argsort()[-5:]]
    print(f"Topic {idx}: {top_words}")
```

Listing 4: NMF in Python

7 Choosing the Number of Topics

Overview

Selecting the optimal number of topics is crucial. Two key metrics help: **coherence** (within-topic word relatedness) and **exclusivity** (between-topic word distinctiveness).

7.1 Coherence

Topic Coherence

Coherence measures how related the top words within a topic are to each other. Higher coherence means the topic's top words frequently co-occur in documents.

Intuition: If Topic 1's top words are [dogs, walks, fetch, leash, park], these words should appear together in documents more often than random word pairs.

Computing Coherence (Simplified)

For top words in a topic, compute co-occurrence:

$$\text{Coherence} \propto \sum_{i<j} \log \frac{P(w_i, w_j) + \epsilon}{P(w_i) \cdot P(w_j)}$$

This is essentially a log-transformed correlation measure.

7.2 Exclusivity

Topic Exclusivity

Exclusivity measures how distinct topics are from each other. High exclusivity means top words in one topic don't appear as top words in other topics.

Exclusivity Calculation (Simplified)

For word w in topic k:

$$\text{Exclusivity}(w, k) = \frac{P(w|\text{topic } k)}{\sum_{k'} P(w|\text{topic } k')}$$

If "dogs" has high probability only in Topic 1 and low in others, it has high exclusivity for Topic 1.

7.3 The Coherence-Exclusivity Trade-off

Trade-off

Coherence and exclusivity often compete:

- More topics → Higher exclusivity but potentially lower coherence
- Fewer topics → Higher coherence but topics may overlap

Important

Selection strategy:

1. Compute coherence and exclusivity for different numbers of topics
2. Standardize both metrics (z-scores)
3. Plot: x-axis = exclusivity, y-axis = coherence
4. Choose model closest to top-right corner (high both)

```python
from gensim.models.coherencemodel import CoherenceModel

# After training LDA model
coherence_model = CoherenceModel(
    model=lda_gensim,
    texts=tokenized_docs,
    dictionary=dictionary,
    coherence='c_v',      # Coherence type
    topn=20               # Top 20 words per topic
)
coherence_score = coherence_model.get_coherence()
print(f"Coherence: {coherence_score:.4f}")
```

Listing 5: Coherence calculation with gensim

7.4 Multiple Runs

Stochastic Nature of LDA

LDA results vary between runs due to:

- Random initialization
- EM algorithm finding different local optima
- Topic labeling is arbitrary (Topic 1 in run A might be Topic 2 in run B)

Best practice: Run multiple times with different seeds, evaluate coherence/exclusivity, select best run.

8 LDA Implementation in R

Overview

R has excellent support for topic modeling, especially for Structural Topic Modeling (STM). Learning basic R is worthwhile for NLP research.

8.1 Setting Up R

1. Download R from `https://cran.r-project.org/`
2. Download RStudio from `https://posit.co/`
3. Install R first, then RStudio (so RStudio finds R automatically)

8.2 Basic R Syntax for LDA

```r
# Install and load packages
install.packages("topicmodels")
library(topicmodels)

# Create documents
documents <- c(
  "cats are wonderful pets",
  "cats and dogs are popular",
  "dogs enjoy long walks",
  # ... more documents
)

# Preprocessing
corpus <- Corpus(VectorSource(documents))
corpus <- tm_map(corpus, tolower)
corpus <- tm_map(corpus, removePunctuation)
corpus <- tm_map(corpus, removeWords, stopwords("english"))

# Create Document-Term Matrix
dtm <- DocumentTermMatrix(corpus)

# Fit LDA
lda_model <- LDA(dtm, k = 2, control = list(seed = 42))

# Get top terms per topic
```

```r
terms(lda_model, 5)
```

Listing 6: LDA in R

8.3 R Markdown for Reports

RMarkdown Files (.Rmd)

Similar to Jupyter notebooks:

- Mix code and text
- Code in "chunks" (like cells)
- `Ctrl+Enter` runs current line
- `Knit` creates HTML/PDF report
- Use `#` for sections, `##` for subsections

9 Practical Applications

9.1 What Can You Do with Topic Models?

1. **Document Classification**: Assign documents to dominant topics
2. **Search**: Find documents about specific topics
3. **Trend Analysis**: Track topic prevalence over time
4. **Bias Detection**: Correlate topics with metadata (gender, source)
5. **Summarization**: Understand what a corpus is "about"

9.2 Real-World Example: Student Evaluations

Analyzing Student Evaluations

Study with 1 million student evaluations:

- Extracted 11 topics from evaluation text
- Topics: "caring instructor", "interesting lectures", "good feedback", etc.
- Correlated with instructor gender

Findings:

- Female instructors: more mentions of "caring", "facilitates discussion", "nice feedback"
- Male instructors: more mentions of "humor", "interesting", "relevant"
- This pattern persisted after controlling for department and course type
- Suggests systematic bias in how students perceive instructors

9.3 Comparing Across Departments

Topic Variation by Division

Same student evaluation study:

- **Sciences**: "explains complex concepts effectively"
- **Humanities**: "facilitates effective discussions"
- **Freshman seminars**: "positive timely feedback"

These differences reflect genuine pedagogical differences across disciplines.

10 Structural Topic Modeling (STM) Preview

Overview

STM extends LDA by incorporating **covariates**—document-level metadata that can affect topic prevalence and content.

LDA vs STM

- **LDA**: Documents have topic proportions, but ignores metadata
- **STM**: Topic proportions can depend on covariates (author gender, publication source, date, etc.)

When STM Helps

Analyzing news articles with known sources:

- LDA: Discovers topics, then you manually correlate with sources
- STM: Directly models how source affects topic distribution
- STM produces more accurate topics by using all available information

Software Note

STM is primarily implemented in R (`stm` package). Python implementations exist but are unofficial. For serious STM work, learn R.

11 One-Page Summary

Summary

Topic Modeling discovers hidden themes in document collections.
Why Not Clustering?

- Documents contain multiple topics (soft assignment)
- Clustering forces hard assignment to one cluster

LDA (Latent Dirichlet Allocation):

- Probabilistic model: documents are mixtures of topics
- Topics are distributions over words
- Generative process: choose topic proportions, then for each word, choose topic then word
- Uses EM algorithm for parameter estimation
- Number of topics is a hyperparameter

NMF (Non-negative Matrix Factorization):

- Deterministic approach: $V \approx W \cdot H$
- Faster but harder to interpret
- No probabilistic interpretation

Choosing Number of Topics:

- **Coherence**: Are top words related? (higher = better)
- **Exclusivity**: Are topics distinct? (higher = better)
- Balance both; maximize average of standardized scores

Interpreting Topics:

- Look at top words per topic
- Better: examine documents with highest topic prevalence
- Assign human-readable labels after analysis

Key Formulas:

$$\theta_m \sim \text{Dirichlet}(\alpha) \quad \text{(topic proportions)}$$
$$z_n \sim \text{Multinomial}(\theta_m) \quad \text{(topic selection)}$$
$$w_n \sim \text{Multinomial}(\beta_{z_n}) \quad \text{(word selection)}$$
$$V \approx W \cdot H \quad \text{(NMF decomposition)}$$

12 Glossary

Key Terms

- **LDA**: Latent Dirichlet Allocation—probabilistic topic model
- **NMF**: Non-negative Matrix Factorization—deterministic topic model
- **STM**: Structural Topic Modeling—LDA with covariates
- **Dirichlet distribution**: Produces probability vectors summing to 1
- **Latent topic**: Hidden theme discovered from data (not predefined)
- **Topic proportions**: How much each topic contributes to a document

- **Coherence**: Metric measuring word co-occurrence within topics
- **Exclusivity**: Metric measuring distinctiveness between topics
- **EM algorithm**: Expectation-Maximization for latent variable models
- **MLE**: Maximum Likelihood Estimation—find parameters maximizing data probability
- **Undercomplete**: Bottleneck dimension $<$ input dimension
- **Stacked autoencoder**: Deep autoencoder with multiple hidden layers
- **Bag of words**: Document representation ignoring word order
- **Document-term matrix**: Matrix of word counts (documents $\times$ vocabulary)

Lecture Information

Course: CSCI E-89B: Natural Language Processing
Lecture: Lecture 8
Topic: Structural Topic Modeling (STM)
Date: Fall 2024

Contents

1 Quiz Review: LDA and Topic Modeling

Overview

This lecture extends LDA to Structural Topic Modeling (STM), which incorporates document-level metadata (covariates) into topic modeling. We begin with a review of LDA concepts.

1.1 LDA Quiz Questions

What LDA Does

Question: Which statement best describes what LDA does?
Correct Answer: LDA assumes each document is a **mixture of topics**.
Why other options are wrong:

- "Assigns a single topic to each document" — No, LDA assigns **probability distributions** over topics
- "Supervised algorithm requiring labels" — No, LDA is **unsupervised**
- "Uses K-means" — No, LDA uses probabilistic inference, not K-means

1.2 Determining Optimal Number of Topics

Key Challenge in LDA

Determining the optimal number of topics is a significant challenge:

- **Too many topics**: Different topics become similar (redundancy)
- **Too few topics**: Unrelated concepts get combined
- Number of topics is a **hyperparameter** chosen before training

Methods for choosing K:

1. Maximize coherence and exclusivity (balance both)
2. Maximize held-out likelihood (test set likelihood)
3. Domain knowledge and interpretability

1.3 Role of the Dirichlet Distribution

Dirichlet Distribution in LDA

The Dirichlet distribution generates **topic proportions** (prevalence) for each document:

$$\theta_d \sim \text{Dirichlet}(\alpha)$$

θ_d is a vector of probabilities summing to 1, representing how much each topic contributes to document d.

1.4 LDA vs NMF

Important

Key Difference:

- **LDA**: Probabilistic model using maximum likelihood estimation
- **NMF**: Deterministic matrix algebra ($V \approx W \cdot H$)

NMF does **not** "break down documents into additive parts"—it decomposes a matrix into a **product** (multiplication) of two non-negative matrices.

2 Challenges with Sequence Autoencoders

Overview

Before diving into STM, we address a common challenge students face: building autoencoders for text sequences. This is significantly harder than image autoencoders.

2.1 Why Sequence Autoencoders Are Difficult

The Bottleneck Problem

When building a sequence autoencoder:

- You compress an entire sequence (many vectors) into a **single vector**
- This bottleneck loses the **time component**
- Reconstruction becomes extremely difficult without sufficient data

Comparison: Images vs Text

Image Autoencoders:

- Easier because spatial relationships are preserved through convolutions
- Even with compression, local structure remains

Text Autoencoders:

- Entire sentence compressed to single vector
- All word order and sequence information must be encoded
- Requires **enormous** amounts of training data

2.2 Practical Solutions

Strategies for Better Results

1. **Increase bottleneck dimension**: If reconstruction fails, try larger latent representations
2. **Data augmentation**: Create artificial training samples
   - Replace words with synonyms
   - Drop or shuffle words
   - Vary sentence structure
3. **Alternative architecture**: Skip the bottleneck entirely
   - Use sequence-to-sequence without compression
   - Train embeddings without the "encoding" constraint

Historical Note

Early machine translation systems tried this bottleneck approach—compress source sentence to a vector, then decode to target language. This was state-of-the-art briefly, but was abandoned because of the exact difficulties described above. Modern systems (Transformers) avoid hard bottlenecks.

3 Maximum Likelihood Estimation Revisited

Overview

Understanding MLE is crucial for topic modeling. We use an intuitive analogy before applying it to STM.

3.1 The Wet Cat Analogy

Intuitive MLE Example

Your cat comes home wet. What happened?
Possible explanations:

- It's raining outside $\Rightarrow P(\text{cat wet}|\text{rain}) \approx 1$
- Someone deliberately sprayed the cat $\Rightarrow P(\text{cat wet}|\text{sprayed}) < 1$

MLE conclusion: Most likely it was raining, because that explanation maximizes the probability of observing a wet cat.
Caveat: MLE finds the most likely explanation given the model, but isn't always "correct"—the model could be wrong!

3.2 Non-uniqueness in Topic Models

LDA Results Vary Between Runs

LDA/STM may produce different results each time because:

1. **Label switching**: Topic 1 and Topic 2 could swap
2. **Different local optima**: Multiple valid topic configurations exist
3. **Random initialization**: Starting point affects final solution

Multiple Valid Topic Configurations

Given documents:

- Doc 1: "excellent but difficult"
- Doc 2: "interesting but fast"
- Doc 3: "difficult but interesting"

Option A: Topic 1 = {excellent, difficult}, Topic 2 = {interesting, fast}

- Doc 1: 100% Topic 1
- Doc 2: 100% Topic 2
- Doc 3: 50% each

Option B: Topic 1 = {interesting, fast}, Topic 2 = {difficult, interesting}

- Doc 1: 100% Topic 2
- Doc 2: 100% Topic 1
- Doc 3: 50% each (different composition!)

Both are valid MLE solutions! That's why we run multiple times and select the best.

4 Limitations of LDA

Overview

LDA is powerful but has limitations that motivate Structural Topic Modeling (STM).

4.1 The Metadata Problem

Metadata (Covariates)

Metadata is information **about** documents, not the text itself:

- Author name, gender, age
- Publication date
- Source (New York Times, Financial Times, etc.)

- Department or category
- Any other document-level attributes

Metadata Structure

Document	Gender	Author	Year
"Cats sat..."	Female	Amanda Smith	2024
"Dog ran away..."	Male	Douglas Parker	2025

This metadata could help predict topic distributions but LDA ignores it.

Why Ignoring Metadata is Wasteful

If you know:

- An author's typical writing topics
- A publication's editorial focus
- Time periods when certain topics were trending

Ignoring this information makes topic assignment less accurate!

5 Structural Topic Modeling (STM)

Overview

STM extends LDA by incorporating document-level metadata (covariates) directly into the model, improving topic assignment accuracy and enabling hypothesis testing about how covariates affect topic prevalence.

5.1 The STM Model

STM vs LDA

- **LDA**: $\theta_d \sim \text{Dirichlet}(\alpha)$ (same for all documents)
- **STM**: $\theta_d \sim \text{Logistic-Normal}(\mu_d, \Sigma)$ where μ_d depends on covariates

5.2 The Logistic Normal Distribution

Logistic Normal for Topic Proportions

In STM, topic proportions are generated as:

$$\theta_d | X_d \sim \text{Logistic-Normal}(X_d\gamma, \Sigma)$$

where:

- X_d: Covariate vector for document d (metadata)
- γ: Coefficient matrix (to be estimated)

- Σ: Covariance matrix (to be estimated)

How it works:

1. Compute $\mu_d = X_d\gamma = \gamma_0 + \gamma_1 x_{d,1} + \gamma_2 x_{d,2} + \ldots$
2. Draw $\eta_d \sim \mathcal{N}(\mu_d, \Sigma)$ (multivariate normal)
3. Apply softmax: $\theta_d = \text{softmax}(\eta_d)$

Concrete Example

If metadata is `Gender` (0 = Male, 1 = Female):

$$\mu_d = \gamma_0 + \gamma_1 \cdot \text{Gender}_d$$

For a female author ($\text{Gender}_d = 1$):

$$\mu_d = \gamma_0 + \gamma_1$$

The coefficient γ_1 captures how gender affects expected topic proportions!

5.3 Categorical Variables as Covariates

One-Hot Encoding for Covariates

When covariates are categorical (like author name), they become multiple coefficients:

$$X_d\gamma = \gamma_0 + \underbrace{\gamma_1 \cdot \mathbf{1}[\text{Amanda}] + \gamma_2 \cdot \mathbf{1}[\text{Douglas}] + \ldots}_{\text{One-hot encoded author}}$$

Each category gets its own coefficient in γ.

5.4 The Covariance Matrix

Topic Correlations

The covariance matrix Σ captures correlations between topics:

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12} & \cdots \\ \sigma_{12} & \sigma_2^2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

- Diagonal: Variance of each topic's prevalence
- Off-diagonal: Correlations between topics

These are **estimated from data**, not hyperparameters.

6 STM Estimation: The EM Algorithm

Overview

STM uses the Expectation-Maximization (EM) algorithm because topic assignments are latent (unobserved) variables.

6.1 Why EM is Needed

Important

We observe documents but **not**:

- Which topic generated each word (z_n)
- True topic proportions (θ_d)
- Topic-word distributions (β_k)

Since z_n is unobserved, we can't directly compute the likelihood.

6.2 EM Algorithm Steps

EM Algorithm

E-step (Expectation):

- Given current parameters, compute expected values of latent variables
- Calculate $E[\log P(\text{data}, z|\theta)]$

M-step (Maximization):

- Maximize expected log-likelihood with respect to parameters
- Update γ, Σ, β

Iterate until convergence.

Non-uniqueness and Multiple Runs

EM may converge to different local optima. **Best practice**:

1. Run STM multiple times with different initializations
2. Compare coherence and exclusivity for each run
3. Select the best model

The STM package does this automatically by default.

7 STM Implementation in R

Overview

STM is primarily implemented in R. The `stm` package is highly reliable and widely used in social science research.

7.1 Basic Workflow

```r
# Install and load
install.packages("stm")
library(stm)

# Create documents and metadata
documents <- c(
  "cats are wonderful pets",
  "cats enjoy climbing trees",
  # ... more documents by author 1
  "dogs love running outside",
  "dogs are loyal companions"
  # ... more documents by author 2
)

meta <- data.frame(
  author = c(rep("author1", 8), rep("author2", 8))
)

# Preprocess text
processed <- textProcessor(
  documents,
  metadata = meta,
  lowercase = TRUE,
  removestopwords = TRUE,
  removenumbers = TRUE,
  removepunctuation = TRUE,
  stem = TRUE
)

# Prepare documents
out <- prepDocuments(
  processed$documents,
  processed$vocab,
  processed$meta
)

# Fit STM
stm_model <- stm(
  documents = out$documents,
  vocab = out$vocab,
  K = 5,                          # Number of topics
  prevalence = ~ author,          # Covariates
  data = out$meta,
  max.em.its = 100,
  init.type = "Spectral"
)
```

Listing 1: STM Workflow in R

7.2 The Formula Interface

R Formula Notation

The `prevalence` formula specifies covariates:

- ∼ `author`: Intercept + author effect
- ∼ `author + date`: Multiple covariates
- ∼ `author * date`: Interaction effects

The tilde (∼) notation is standard R regression syntax.

7.3 Preprocessing Details

Document Removal During Preprocessing

`textProcessor` may remove:

- Stop words
- Numbers
- Punctuation
- Infrequent terms

If a document becomes **empty** after preprocessing, it's removed along with its metadata row. That's why we use `out$meta` instead of the original metadata!

8 Analyzing STM Results

8.1 Viewing Topic Summaries

```
# Summary of topics
summary(stm_model)

# Plot expected topic proportions
plot(stm_model, type = "summary", n = 5)
```

Listing 2: Summarize and plot topics

8.2 Interpreting Topics

Important

Best Practice: Don't rely solely on top words to interpret topics. Instead, examine **documents with highest topic prevalence**.
Top words may be:

- Common across many topics (e.g., "Australia" in Australian news)
- Stemmed and hard to interpret
- Ambiguous out of context

```r
# Find documents most associated with each topic
findThoughts(stm_model, texts = documents, n = 3, topics = 1:5)
```

Listing 3: Find representative documents

8.3 Estimating Covariate Effects

estimate Effect Function

To understand how covariates affect topic prevalence, use `estimateEffect`:

1. Sample from posterior distribution of θ
2. Regress sampled prevalences on covariates
3. Compute confidence intervals

```r
# Estimate effects for all topics
effects <- estimateEffect(
  1:5 ~ author,                    # Topics ~ covariates
  stmobj = stm_model,
  metadata = out$meta,
  uncertainty = "Global"
)

# Plot difference between authors
plot(effects,
  covariate = "author",
  topics = 1:5,
  model = stm_model,
  method = "difference",
  cov.value1 = "author2",          # On the right
  cov.value2 = "author1",          # Subtracted
  main = "Effect of Author on Topic Prevalence"
)
```

Listing 4: Estimate and plot effects

8.4 Time Series of Topic Prevalence

```r
# If date is a covariate
effects_time <- estimateEffect(
  1:5 ~ date,
  stmobj = stm_model,
  metadata = out$meta
)

plot(effects_time,
  covariate = "date",
  topics = 1:5,
  model = stm_model,
  method = "continuous",
  xlab = "Date",
  main = "Topic Prevalence Over Time"
)
```

Listing 5: Topic prevalence over time

Interpreting Time Plots

The plot shows:

- Expected topic prevalence (line)
- Confidence intervals (shaded region)
- Upward trend: topic becoming more prevalent
- Downward trend: topic becoming less prevalent

This assumes a **linear** trend. For non-linear patterns, use date as categorical or add polynomial terms.

9 Selecting the Number of Topics

Overview

Choosing K (number of topics) requires running STM multiple times and comparing performance metrics.

9.1 Using searchK

```
# Search across different numbers of topics
k_search <- searchK(
  documents = out$documents,
  vocab = out$vocab,
  K = 2:10,                          # Range of K to try
  prevalence = ~ author,
  data = out$meta,
  init.type = "Spectral"
)

# Plot diagnostics
plot(k_search)
```

Listing 6: Search for optimal K

9.2 Coherence vs Exclusivity Trade-off

Model Selection Criteria

Semantic Coherence: Are top words within a topic co-occurring in documents?
Exclusivity: Are top words unique to each topic?
These often trade off:

- More topics ⇒ Higher exclusivity, lower coherence
- Fewer topics ⇒ Higher coherence, lower exclusivity

```
# Extract metrics
coherence <- k_search$results$semcoh
exclusivity <- k_search$results$exclus

# Create comparison data frame
```

```r
metrics <- data.frame(
  K = 2:10,
  coherence = coherence,
  exclusivity = exclusivity
)

# Plot
library(ggplot2)
ggplot(metrics, aes(x = exclusivity, y = coherence, label = K)) +
  geom_point() +
  geom_text(nudge_x = 0.01) +
  labs(x = "Exclusivity", y = "Semantic Coherence",
       title = "Coherence vs Exclusivity by Number of Topics") +
  theme_minimal()
```

Listing 7: Extract and plot coherence/exclusivity

Important

Selection Strategy:

1. Rescale both metrics to $[0, 1]$
2. Compute average of rescaled metrics
3. Choose K that maximizes the average
4. Alternatively: visual inspection—pick the point closest to the upper-right corner

10 Real-World Application: Student Evaluations

Overview

A published study analyzed 11 years of student evaluations at Harvard using STM, discovering how topic prevalence varies with instructor gender and department.

10.1 Study Design

Student Evaluation Study

Data: 1 million student evaluations
Covariates:

- Instructor gender
- Instructor age
- Academic division
- Course type

Number of topics: 11 (selected via coherence/exclusivity)

10.2 Key Findings

Important

Gender Differences in Topic Prevalence:
When students discuss **female** instructors, they more often mention:

- "Caring, enthusiastic instructor"
- "Facilitates effective discussion"
- "Nice feedback"

When students discuss **male** instructors, they more often mention:

- "Lectures are interesting and relevant"
- "Uses humor effectively"

These patterns persisted even after controlling for department and course type—suggesting potential **student bias**.

10.3 Division-Level Patterns

Topic Variation by Academic Division

Sciences: "Explains complex concepts effectively" (high prevalence)
Humanities: "Facilitates effective discussions" (high prevalence)
Freshman Seminars: "Positive timely feedback" (high prevalence)
These differences reflect genuine pedagogical differences across disciplines.

10.4 Practical Implications

Why This Matters

If student evaluations show systematic biases:

- Promotion decisions may be affected
- Tenure reviews could be biased
- Adjustments might be needed when interpreting evaluations

STM allows researchers to **quantify** these effects and test their significance.

11 Advanced STM Features

11.1 Topic Correlations

```
# Plot topic correlations
topicCorr(stm_model, method = "simple")
```

Listing 8: Visualize topic correlations

This shows which topics tend to co-occur within documents.

11.2 Selecting Among Multiple Runs

```r
# searchK already runs multiple times internally
# To manually select the best model:
best_model <- selectModel(
  documents = out$documents,
  vocab = out$vocab,
  K = 5,
  prevalence = ~ author,
  data = out$meta,
  runs = 20                          # Number of runs
)

# Select based on exclusivity/coherence
plotModels(best_model)
```

Listing 9: Select best model from multiple runs

12 One-Page Summary

Summary

Structural Topic Modeling (STM) extends LDA by incorporating document metadata.

Key Differences from LDA:

- LDA: $\theta_d \sim \text{Dirichlet}(\alpha)$
- STM: $\theta_d \sim \text{Logistic-Normal}(X_d\gamma, \Sigma)$

STM Generation Process:

1. Compute $\mu_d = X_d\gamma$ (linear function of covariates)
2. Draw $\eta_d \sim \mathcal{N}(\mu_d, \Sigma)$
3. Apply softmax: $\theta_d = \text{softmax}(\eta_d)$
4. Generate words as in LDA

Why Use STM?:

- Incorporates metadata (author, date, source)
- More accurate topic assignments
- Test hypotheses about covariate effects
- Track topic prevalence over time

R Implementation:

1. `textProcessor()`: Preprocess documents
2. `prepDocuments()`: Prepare for modeling
3. `stm()`: Fit the model
4. `estimateEffect()`: Analyze covariate effects

5. `searchK()`: Find optimal number of topics

Model Selection:

- Balance **coherence** (words co-occur) and **exclusivity** (topics distinct)
- Run multiple times, select best via metrics
- Rescale metrics to $[0, 1]$, maximize average

Best Practices:

- Interpret topics via representative documents, not just top words
- Use `out$meta` after preprocessing (some rows removed)
- Multiple runs are essential due to non-uniqueness

13 Glossary

Key Terms

- **STM**: Structural Topic Modeling—LDA extension with covariates
- **Metadata/Covariates**: Document-level information (author, date, source)
- **Logistic Normal**: Distribution for generating topic proportions in STM
- **Prevalence**: Expected proportion of a topic in documents
- **EM Algorithm**: Expectation-Maximization for latent variable models
- **E-step**: Compute expected log-likelihood given current parameters
- **M-step**: Maximize expected log-likelihood to update parameters
- **Posterior distribution**: Distribution of parameters after observing data
- **estimateEffect**: STM function to analyze covariate effects
- **searchK**: STM function to find optimal number of topics
- **selectModel**: STM function to choose best run
- **findThoughts**: STM function to find representative documents
- **Coherence**: Metric for within-topic word co-occurrence
- **Exclusivity**: Metric for between-topic word distinctiveness
- **textProcessor**: STM preprocessing function
- **prepDocuments**: STM document preparation function

Lecture Information

Course: CSCI E-89B: Natural Language Processing
Lecture: Lecture 9
Topic: Classical Machine Learning for NLP
Date: Fall 2024

Contents

1 Introduction to Classical ML for NLP

Overview

This lecture covers classical machine learning techniques applied to natural language processing. While neural networks dominate modern NLP, classical methods remain valuable for smaller datasets and provide interpretable baselines.

1.1 Why Classical Methods?

When to Use Classical Methods

Classical methods are particularly useful when:

- You have a **small dataset** (neural networks need lots of data)
- You need **interpretability** (understand why predictions are made)
- You need **fast training** (no GPU required)
- You want a **baseline** before trying complex models

1.2 Text Representation for Classical Methods

Important

Classical methods require **numerical vectors** as input. We cannot feed raw text directly. Common representations:

- **TF-IDF vectors**: Term frequency-inverse document frequency
- **Bag of Words**: Simple word counts
- **N-gram features**: Sequences of consecutive words

2 Naive Bayes Classifier

Overview

Naive Bayes is a probabilistic classifier based on Bayes' theorem with a "naive" assumption of feature independence. Despite its simplicity, it works surprisingly well for text classification.

2.1 Bayes' Theorem

Bayes' Theorem

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

For classification:

$$P(C_k|X) = \frac{P(X|C_k) \cdot P(C_k)}{P(X)}$$

where:

- C_k: Class k (e.g., spam or not spam)
- X: Feature vector (e.g., TF-IDF representation)
- $P(C_k)$: Prior probability of class k
- $P(X|C_k)$: Likelihood of features given class
- $P(C_k|X)$: Posterior probability (what we want)

2.2 The Naive Assumption

Conditional Independence

Naive Bayes assumes features are **conditionally independent** given the class:

$$P(X|C_k) = P(x_1|C_k) \cdot P(x_2|C_k) \cdot \ldots \cdot P(x_n|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$$

Why "Naive"?

The independence assumption is rarely true in practice:

- If "cat" appears, "dog" is less likely in the same document
- Words are correlated through context

However, TF-IDF already loses much contextual information, so the assumption is less problematic than it seems. The classifier still performs well in practice!

2.3 Classification Decision

Classification Rule

Choose the class with highest posterior probability:

$$\hat{y} = \arg\max_{C_k} P(C_k|X) = \arg\max_{C_k} P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

In practice, use log probabilities for numerical stability:

$$\hat{y} = \arg\max_{C_k} \left[\log P(C_k) + \sum_{i=1}^{n} \log P(x_i|C_k) \right]$$

2.4 Applications

Common Applications

- **Spam detection**: Classic application
- **Sentiment analysis**: Positive/negative classification
- **Document categorization**: News topic classification

2.5 Implementation

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split

# Sample documents
documents = [
    "Cats are wonderful pets",
    "Dogs enjoy long walks",
    # ... more documents
]
labels = [0, 1, ...]  # 0=cats, 1=dogs

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    documents, labels, test_size=0.25
)

# Create TF-IDF vectors
vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)  # Don't refit!

# Train and predict
nb = MultinomialNB()
nb.fit(X_train_tfidf, y_train)
predictions = nb.predict(X_test_tfidf)
```

Listing 1: Naive Bayes for Text Classification

Data Leakage Warning

Always fit the TF-IDF vectorizer on **training data only**, then transform test data. Never fit on test data—this causes data leakage!

3 K-Nearest Neighbors (KNN)

Overview

KNN is a simple, non-parametric algorithm that classifies based on the labels of the K closest training examples. It requires no training phase—it memorizes the entire dataset.

3.1 How KNN Works

KNN Algorithm

For a new data point:

1. Compute distances to all training points
2. Find the K nearest neighbors
3. **Classification**: Take majority vote among K neighbors
4. **Regression**: Take average of K neighbors' values

1D Example

Data points: $\{(1,0),(2,0),(3,0),(5,1),(6,1),(7,1)\}$
To classify point $x = 4$ with $K = 3$:

1. Nearest neighbors: $x = 3, 5, 6$ (or $3, 5, 2$)
2. Labels: $\{0, 1, 1\}$
3. Majority vote: 1 wins
4. Prediction: Class 1

3.2 Distance Metrics

Common Distance Metrics

Euclidean Distance (default):

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Manhattan Distance:

$$d(x,y) = \sum_{i=1}^{n}|x_i - y_i|$$

Cosine Similarity (often better for text):

$$\text{similarity}(x,y) = \frac{x \cdot y}{\|x\|\|y\|}$$

3.3 Feature Scaling

Scale Your Features!

If features are on different scales, larger-scale features dominate:

- Age: 20–80 years (range: 60)
- Income: \$20,000–\$200,000 (range: 180,000)

Income will completely dominate the distance calculation! Always normalize features to similar scales.

3.4 Choosing K

Important

Use odd K for binary classification to avoid ties.
Choosing K:

- Too small K: Overfitting (sensitive to noise)
- Too large K: Underfitting (ignores local structure)
- Optimize K using validation set or cross-validation

3.5 Is KNN Deterministic?

KNN Can Be Stochastic!

KNN is **not always deterministic**:

- **Even K**: May have ties in majority vote
- **Equal distances**: Multiple points at same distance—which to include?

Ties must be broken randomly, so results may vary between runs. Default implementations often run multiple times and average.

3.6 Implementation

```python
from sklearn.neighbors import KNeighborsClassifier

# After TF-IDF vectorization...
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_tfidf, y_train)
predictions = knn.predict(X_test_tfidf)

# Optimize K
from sklearn.model_selection import GridSearchCV

param_grid = {'n_neighbors': range(1, 21)}
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_tfidf, y_train)
print(f"Best K: {grid_search.best_params_}")
```

Listing 2: KNN for Text Classification

4 Logistic Regression

Overview

Logistic regression is a linear model for binary classification. It's equivalent to a single-layer neural network with sigmoid activation.

4.1 The Model

Logistic Regression

$$P(y = 1|X) = \sigma(W \cdot X + b) = \frac{1}{1 + e^{-(W \cdot X + b)}}$$

where:

- σ: Sigmoid function
- W: Weight vector (learned)
- b: Bias term (learned)
- X: Feature vector

4.2 Decision Boundary

Important

The decision boundary is where $P(y = 1|X) = 0.5$, which means $W \cdot X + b = 0$.
In 2D, this is a **straight line**. In higher dimensions, it's a **hyperplane**.
Limitation: Logistic regression can only separate classes that are linearly separable.

4.3 Connection to Neural Networks

Logistic Regression

Logistic regression is exactly a neural network with:

- No hidden layers
- Sigmoid activation on output
- Binary cross-entropy loss

Training via maximum likelihood is mathematically equivalent to minimizing cross-entropy loss.

4.4 Non-linear Decision Boundaries

Feature Engineering for Non-linearity

If data isn't linearly separable, add polynomial features:
Original features: x_1, x_2
Extended features: $x_1, x_2, x_1^2, x_2^2, x_1 x_2$
This allows curved decision boundaries, but requires manual feature design.

4.5 Implementation

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(max_iter=1000)
lr.fit(X_train_tfidf, y_train)
predictions = lr.predict(X_test_tfidf)
```

```
6
7 # Get probabilities
8 probabilities = lr.predict_proba(X_test_tfidf)
```

Listing 3: Logistic Regression for Text

5 Support Vector Machines (SVM)

Overview

SVMs were the dominant machine learning method before deep learning (pre-2012). They find the hyperplane that maximizes the margin between classes.

5.1 The Maximum Margin Idea

Maximum Margin Classifier

Given two linearly separable classes, find the hyperplane that:

- Correctly separates all points
- Maximizes the **margin**—the distance to the nearest points (support vectors)

Visualizing the Margin

Imagine two parallel lines (in 2D) separating two classes. The gap between these lines is the margin. SVM finds the position that makes this gap as wide as possible.
Points touching the margin boundaries are **support vectors**—only these points determine the decision boundary position.

5.2 Mathematical Formulation

SVM Optimization

The hyperplane is defined by $W \cdot X + b = 0$.
Hard Margin SVM (linearly separable):

$$\min_{W,b} \frac{1}{2}\|W\|^2 \quad \text{subject to} \quad y_i(W \cdot X_i + b) \geq 1 \quad \forall i$$

This maximizes the margin $\frac{2}{\|W\|}$.

5.3 The Overfitting Problem

Hard Margin Overfitting

With hard margin SVM, only support vectors affect the decision boundary. Moving one support vector changes everything!
This is **overfitting**—the model is too sensitive to individual points.

5.4 Soft Margin SVM

Soft Margin SVM

Allow some points to be inside the margin or misclassified:

$$\min_{W,b,\xi} \frac{1}{2}\|W\|^2 + C\sum_i \xi_i \quad \text{subject to} \quad y_i(W \cdot X_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

where:

- ξ_i: Slack variable (how far point i is from correct side)
- C: Regularization parameter (trade-off between margin and violations)

Important

Parameter C:

- Large C: Few violations allowed, narrower margin (risk overfitting)
- Small C: More violations allowed, wider margin (risk underfitting)

Tune C using cross-validation.

5.5 Non-linear Decision Boundaries: Kernels

What if data isn't linearly separable?

Consider points arranged in a circle (class 0) surrounded by points outside (class 1). No line can separate them!

The Kernel Trick

Add a new dimension: $z = x_1^2 + x_2^2$ (distance from origin).
In this 3D space, points are linearly separable by a plane!
Key insight: We don't actually compute the transformation. Instead, we replace the dot product $X_i \cdot X_j$ with a **kernel function** $K(X_i, X_j)$.

5.6 Common Kernels

Kernel	Formula
Linear	$K(x, y) = x \cdot y$
Polynomial	$K(x, y) = (x \cdot y + c)^d$
RBF (Gaussian)	$K(x, y) = \exp(-\gamma\|x - y\|^2)$

RBF Kernel

The RBF (Radial Basis Function) kernel can approximate any decision boundary. It's equivalent to projecting to infinite dimensions!
The γ parameter controls how "local" the influence of each point is.

5.7 Implementation

```python
from sklearn.svm import SVC

# Linear kernel
svm_linear = SVC(kernel='linear', C=1.0)
svm_linear.fit(X_train_tfidf, y_train)

# RBF kernel
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale')
svm_rbf.fit(X_train_tfidf, y_train)

# Grid search for optimal parameters
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf', 'poly']
}
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train_tfidf, y_train)
```

Listing 4: SVM for Text Classification

6 Decision Trees and Random Forests

Overview

Decision trees recursively split data based on feature values. Random forests combine many trees to reduce overfitting and improve accuracy.

6.1 Decision Tree Algorithm

Building a Decision Tree

At each node:

1. Consider all features and all possible split points
2. Choose the split that best separates classes
3. Recurse on each resulting subset
4. Stop when a criterion is met (max depth, min samples, pure node)

6.2 Splitting Criteria

Gini Index (Classification)

For a node with proportion p of class 1:

$$\text{Gini} = 1 - (p^2 + (1-p)^2) = 2p(1-p)$$

- Pure node ($p = 0$ or $p = 1$): Gini $= 0$
- Maximum impurity ($p = 0.5$): Gini $= 0.5$

Choose splits that minimize total Gini index of child nodes.

Entropy (Alternative)

$$\text{Entropy} = -p\log_2(p) - (1-p)\log_2(1-p)$$

Information gain = parent entropy minus weighted child entropy.

6.3 Overfitting in Decision Trees

Trees Easily Overfit

Without restrictions, a decision tree will:

- Split until every leaf contains one sample
- Perfectly classify training data
- Perform poorly on test data

Solutions: Limit max depth, require minimum samples per leaf, or pruning.

6.4 Random Forests

Random Forest Algorithm

1. Create B bootstrap samples (sample with replacement)
2. For each sample, build a decision tree:
   - At each split, consider only $\sqrt{n}$ random features
   - Grow tree fully (no pruning needed)
3. Final prediction: Majority vote (classification) or average (regression)

6.5 Why Random Forests Work

Important

Two sources of randomness:

1. **Bootstrap sampling**: Each tree sees different data
2. **Random feature selection**: Each split considers different features

This creates **diverse** trees. When averaged, individual tree errors cancel out, leaving only the "signal."
Random forests are nearly **impossible to overfit**!

6.6 Bootstrap Sampling

Bootstrap Sample

Sample n points **with replacement** from dataset of size n:

- Some points appear multiple times
- Some points don't appear at all (37%)
- Each bootstrap sample is slightly different

6.7 Interpreting Black Box Models

Interpretability Challenge

Random forests are "black boxes"—hard to understand why predictions are made.
Solutions:

- **Partial Dependence Plots (PDP)**: Show how each feature affects predictions
- **SHAP values**: Explain individual predictions
- **Feature importance**: Which features matter most

6.8 Implementation

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(
    n_estimators=100,        # Number of trees
    max_features='sqrt',     # Features per split
    criterion='gini',        # Splitting criterion
    random_state=42
)
rf.fit(X_train_tfidf, y_train)
predictions = rf.predict(X_test_tfidf)

# Feature importance
importance = rf.feature_importances_
```

Listing 5: Random Forest for Text

7 K-Fold Cross-Validation

Overview

K-fold cross-validation is essential for classical methods where datasets are small and we need reliable performance estimates.

7.1 Why Cross-Validation?

Important

For neural networks with large datasets:

- Split into train/validation/test
- Plenty of data for each split

For classical methods with small datasets:

- Can't afford to reserve much for testing
- Single split may not be representative
- Need K-fold cross-validation

7.2 K-Fold Procedure

K-Fold Cross-Validation

1. Split data into K equal parts (folds)
2. For $i = 1, \ldots, K$:
   - Use fold i as test set
   - Use remaining $K - 1$ folds as training set
   - Train model and evaluate on fold i
3. Average performance across all K folds

5-Fold Cross-Validation

Data split: [Fold 1] [Fold 2] [Fold 3] [Fold 4] [Fold 5]

- Run 1: Train on 2,3,4,5, test on 1
- Run 2: Train on 1,3,4,5, test on 2
- Run 3: Train on 1,2,4,5, test on 3
- Run 4: Train on 1,2,3,5, test on 4
- Run 5: Train on 1,2,3,4, test on 5

Every data point is tested exactly once!

7.3 Using Pipelines

TF-IDF Must Be Refit Each Fold!

When using K-fold, the TF-IDF vectorizer must be fit on training folds only, not test fold. Use sklearn Pipeline to handle this automatically.

```
1 from sklearn.pipeline import Pipeline
```

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold

# Create pipeline
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer(stop_words='english')),
    ('clf', MultinomialNB())
])

# K-fold cross-validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipeline, documents, labels, cv=cv, scoring='accuracy')

print(f"Mean accuracy: {scores.mean():.4f}")
print(f"Std: {scores.std():.4f}")
```

Listing 6: Pipeline with Cross-Validation

8 Evaluation Metrics for Classification

8.1 Confusion Matrix

Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

8.2 Key Metrics

Classification Metrics

Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Sensitivity (Recall, True Positive Rate):

$$\text{Sensitivity} = \frac{TP}{TP + FN} = \frac{\text{Correctly predicted positives}}{\text{All actual positives}}$$

Specificity (True Negative Rate):

$$\text{Specificity} = \frac{TN}{TN + FP} = \frac{\text{Correctly predicted negatives}}{\text{All actual negatives}}$$

Precision:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{\text{True positives}}{\text{All predicted positives}}$$

F1 Score:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

8.3 Imbalanced Datasets

Accuracy is Misleading for Imbalanced Data

If 95% of emails are not spam:

- Always predicting "not spam" gives 95% accuracy!
- But sensitivity for spam is 0%

Solutions:

- Use precision/recall/F1 instead of accuracy
- Adjust classification threshold (from 0.5)
- Use class weights or focal loss
- Resample data (SMOTE, undersampling)

9 Model Comparison: Text Classification Example

BBC News Classification Results

Task: Classify news articles as "tech" or "not tech"

Model	Accuracy	Sensitivity	Specificity
Naive Bayes	0.90	0.43	1.00
KNN (K=5)	0.97	0.94	0.98
KNN (K=19, optimized)	0.98	0.94	0.99
Logistic Regression	0.92	0.71	1.00
SVM (linear)	0.92	0.73	1.00
SVM (optimized)	0.99	0.97	1.00
Random Forest	0.98	0.89	1.00

Winner: SVM with optimized parameters (C=10, linear kernel)

Observations

- Models predicting mostly "not tech" have high specificity but low sensitivity
- This happens because "not tech" is the majority class
- SVM (pre-neural network era champion) performs best overall
- KNN performs surprisingly well with optimized K

10 One-Page Summary

Summary

Classical ML for NLP: Still valuable for small datasets and interpretability.
Naive Bayes:

- Assumes feature independence: $P(X|C) = \prod_i P(x_i|C)$
- Fast, simple, works well for text
- Use: Spam detection, sentiment analysis

K-Nearest Neighbors:

- Classify by majority vote of K nearest points
- No training—stores entire dataset
- Scale features! Choose odd K
- Tune K via cross-validation

Logistic Regression:

- $P(y = 1|X) = \sigma(W \cdot X + b)$
- Linear decision boundary (hyperplane)
- Equivalent to single-layer neural network

Support Vector Machines:

- Maximize margin between classes
- Soft margin allows violations (parameter C)
- Kernels enable non-linear boundaries (RBF, polynomial)
- Dominant method pre-2012

Random Forests:

- Ensemble of decision trees
- Bootstrap + random feature selection
- Nearly impossible to overfit
- Interpret via SHAP, PDP

K-Fold Cross-Validation:

- Essential for small datasets
- Each point tested exactly once
- Use Pipeline to refit TF-IDF each fold

Metrics for Imbalanced Data:

- Accuracy misleading when classes unbalanced
- Use precision, recall, F1, AUC-PR

11 Glossary

Key Terms

- **Naive Bayes**: Probabilistic classifier assuming feature independence
- **KNN**: K-Nearest Neighbors—classify by neighbor majority
- **Logistic Regression**: Linear model with sigmoid output
- **SVM**: Support Vector Machine—maximize margin classifier
- **Kernel**: Function replacing dot product to enable non-linear boundaries
- **Support Vector**: Point touching the margin boundary
- **Soft Margin**: SVM allowing margin violations
- **Decision Tree**: Recursive feature splitting
- **Random Forest**: Ensemble of trees with bootstrap + random features
- **Bagging**: Bootstrap AGGregatING—averaging bootstrap models
- **Gini Index**: Impurity measure for splitting
- **Cross-Validation**: Rotating train/test splits
- **Sensitivity (Recall)**: TP / (TP + FN)
- **Specificity**: TN / (TN + FP)
- **Precision**: TP / (TP + FP)
- **F1 Score**: Harmonic mean of precision and recall
- **Data Leakage**: Using test information during training
- **Bootstrap Sample**: Sample with replacement
- **PDP**: Partial Dependence Plot for interpretability
- **SHAP**: Shapley Additive Explanations for feature importance

Lecture Information

Course: CSCI E-89B: Natural Language Processing
Lecture: Lecture 10
Topic: Named Entity Recognition (NER)
Date: Fall 2024

Contents

1 Introduction to Named Entity Recognition

Overview

Named Entity Recognition (NER) is the task of identifying and classifying named entities in text into predefined categories such as person names, organizations, locations, dates, and more. It's a fundamental NLP task with applications in information extraction, question answering, and machine translation.

1.1 What are Named Entities?

Named Entity Categories

Common named entity types include:

- **PERSON**: Names of people (Donald Trump, Marie Curie)
- **ORG**: Organizations, companies, institutions (Tesla, Harvard University)
- **GPE**: Geopolitical entities—countries, cities, states (America, Paris)
- **LOC**: Non-GPE locations (Mount Everest, Pacific Ocean)
- **DATE**: Dates and time periods (November 4, 2024, this year)
- **TIME**: Times (3:00 PM, midnight)
- **MONEY**: Monetary values ($1 trillion, 50 euros)
- **PERCENT**: Percentages (40%, two-thirds)
- **CARDINAL**: Numbers not fitting other categories (50, three)
- **ORDINAL**: Ordinal numbers (first, 2nd)
- **NORP**: Nationalities, religious/political groups (Chinese, Republican)

1.2 Why is NER Important?

Important

NER is crucial for many downstream tasks:

- **Machine Translation**: Knowing "Tesla" is an organization helps translate correctly
- **Information Extraction**: Extract structured data from unstructured text
- **Question Answering**: Identify entities mentioned in questions
- **Search**: Improve semantic search by understanding entity types
- **Sentiment Analysis**: Attribute sentiment to specific entities

NER in Action

Input text: "Donald Trump won more than 50 electoral votes this year. Tesla's stock rose 2.4%."
NER output:

Entity	Type	Position
Donald Trump	PERSON	0–11
more than 50	CARDINAL	17–29
this year	DATE	47–56
Tesla	ORG	58–63
2.4%	PERCENT	78–82

1.3 NER for Feature Enhancement

Enhancing Classification with NER

NER can improve text classification by:

1. **Adding entity counts**: Concatenate counts of persons, organizations, etc.
2. **Entity-based features**: Create binary indicators for entity presence
3. **Structured metadata**: Extract entities as document metadata
4. **Relationship extraction**: Find connections between entities

2 Two Approaches to NER

Overview

NER can be performed using two main approaches: rule-based methods using pattern matching, and statistical/neural methods using machine learning.

2.1 Approach Comparison

Aspect	Rule-Based	Statistical/Neural
Training Data	Not required	Required (labeled)
Context Awareness	Limited	High
Maintenance	High burden	Lower
Adaptability	Poor	Good
Interpretability	High	Lower
Scalability	Poor	Good
Accuracy	Depends on rules	Generally higher

3 Rule-Based NER

Overview

Rule-based NER uses handcrafted patterns (regular expressions, dictionaries, linguistic rules) to identify entities. While limited, it's interpretable and requires no training data.

3.1 Regular Expressions for Entity Detection

Common Pattern Components

- `\b` : Word boundary
- `\d{n}` : Exactly n digits
- `\d{1,2}` : 1 or 2 digits
- `\s+` : One or more whitespace characters
- `[A-Z]` : One uppercase letter
- `[a-z]+` : One or more lowercase letters
- `(?:...)` : Non-capturing group
- `?` : Makes preceding element optional
- `|` : OR operator

3.2 Date Pattern Example

```python
import re

# Pattern for dates like "11/04/2024" or "November 4, 2024"
date_pattern = r'''
    \b                              # Word boundary
    (?:
        \d{1,2}/\d{1,2}/\d{4}       # MM/DD/YYYY format
        |
        (?:January|February|March|April|May|June|
           July|August|September|October|November|December)
        \s+                          # Required space
        \d{1,2}                      # Day (1-31)
        (?:,\s*)?                    # Optional comma and space
        \d{4}                        # Year
    )
    \b
'''

text = "The event is on November 4, 2024 or 11/04/2024."
dates = re.findall(date_pattern, text, re.VERBOSE)
print(dates)  # ['November 4, 2024', '11/04/2024']
```

Listing 1: Regular Expression for Dates

3.3 Person Name Pattern

```python
# Pattern for names with titles
```

```
person_pattern = r'''
    \b
    (?:Mr\.|Mrs\.|Ms\.|Dr\.|Professor)  # Title
    \s+                                  # Space
    [A-Z][a-z]+                          # First name (capitalized)
    (?:\s+[A-Z][a-z]+)?                  # Optional last name
    \b
'''

text = "Mr. Trump met with Dr. Smith yesterday."
persons = re.findall(person_pattern, text, re.VERBOSE)
print(persons)  # ['Mr. Trump', 'Dr. Smith']
```

Listing 2: Pattern for Names with Titles

3.4 Organization Pattern

```
# Pattern for company names
org_pattern = r'''
    \b
    [A-Z][a-zA-Z\s]+                    # Company name
    (?:Inc\.|Ltd\.|Corporation|Corp\.)  # Corporate suffix
    \b
'''

text = "Apple Inc. announced new products."
orgs = re.findall(org_pattern, text, re.VERBOSE)
print(orgs)  # ['Apple Inc.']
```

Listing 3: Pattern for Organizations

3.5 Complete Rule-Based NER System

```
import re

def rule_based_ner(text):
    entities = []

    # Date patterns
    date_patterns = [
        r'\b\d{1,2}/\d{1,2}/\d{4}\b',
        r'\b(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[a-z]*\s+\d{1,2},?\s*\d{4}\b'
    ]

    # Email pattern
    email_pattern = r'\b[\w.]+@[\w.]+\.\w+\b'

    # Time pattern
    time_pattern = r'\b\d{1,2}:\d{2}\s*(?:AM|PM|am|pm)?\b'

    # Person pattern (with titles)
    person_pattern = r'\b(?:Mr\.|Mrs\.|Ms\.|Dr\.)\s+[A-Z][a-z]+(?:\s+[A-Z][a-z]+)?\b'

    # Organization pattern
    org_pattern = r'\b[A-Z][a-zA-Z\s]+(?:Inc\.|Ltd\.|Corp\.)\b'

    # Apply patterns
    for pattern in date_patterns:
        for match in re.finditer(pattern, text):
```

```
            entities.append(('DATE', match.group(), match.span()))

    for match in re.finditer(email_pattern, text):
        entities.append(('EMAIL', match.group(), match.span()))

    for match in re.finditer(time_pattern, text):
        entities.append(('TIME', match.group(), match.span()))

    for match in re.finditer(person_pattern, text):
        entities.append(('PERSON', match.group(), match.span()))

    for match in re.finditer(org_pattern, text):
        entities.append(('ORG', match.group(), match.span()))

    return entities

# Test
text = """
Meeting with Dr. Smith at 3:00 PM on November 18, 2024.
Contact: john.doe@company.com. Apple Inc. will attend.
"""
entities = rule_based_ner(text)
for entity_type, entity, span in entities:
    print(f"{entity_type}: {entity} at {span}")
```

Listing 4: Simple Rule-Based NER System

3.6 Limitations of Rule-Based NER

Rule-Based Limitations

- **No context awareness**: "Tesla" could be a person (Nikola Tesla) or company
- **Missing entities**: "Donald Trump" without title won't be recognized
- **Language-specific**: Rules must be rewritten for each language
- **Date format variations**: US vs European formats differ
- **Maintenance burden**: Rules must be constantly updated
- **Name variations**: "La Place" vs "Laplace" requires special handling

4 Statistical and Neural NER

Overview

Modern NER systems use machine learning to learn patterns from labeled data. Neural networks, particularly CNNs and transformers, achieve state-of-the-art performance.

4.1 NER as Sequence Labeling

BIO Tagging Scheme

NER is typically framed as a sequence labeling problem using BIO tags:

- **B-TYPE**: Beginning of an entity of TYPE
- **I-TYPE**: Inside/continuation of an entity

- **O**: Outside any entity

BIO Tagging Example

Sentence: "Donald Trump visited Tesla headquarters."

Token	Tag
Donald	B-PERSON
Trump	I-PERSON
visited	O
Tesla	B-ORG
headquarters	O
.	O

4.2 Statistical Methods

Traditional ML for NER

Hidden Markov Models (HMM):

- Model sequence of tags as Markov chain
- Assume current tag depends only on previous tag
- Limited feature representation

Conditional Random Fields (CRF):

- Discriminative model (directly models $P(\text{tags}|\text{words})$)
- Can use arbitrary features
- Global normalization (considers entire sequence)

4.3 Neural Network Architecture for NER

Important

SpaCy uses a **Convolutional Neural Network** (CNN) for NER. Here's why:

1. Word embeddings capture semantic meaning
2. CNN filters capture local context (neighboring words)
3. Sliding window naturally handles variable-length text
4. Efficient parallel computation

4.4 How Neural NER Works

Neural NER Pipeline

1. **Input**: Document text
2. **Tokenization**: Split into tokens
3. **Embedding**: Convert tokens to vectors (e.g., 4D, 100D)
4. **CNN**: Apply filters over embedding sequences
5. **Output Layer**: Softmax over entity types for each token

NER as CNN Classification

Input: "The cat sat on mat"
Step 1: Embed each token:

Token	Dim 1	Dim 2	Dim 3	Dim 4
The	0.8	0.1	0.3	0.7
cat	0.5	0.7	0.2	0.9
sat	0.3	0.4	0.6	0.1
on	0.2	0.8	0.1	0.5
mat	0.4	0.3	0.7	0.2

Step 2: CNN filters slide over embeddings (captures context)
Step 3: For each token, output probabilities:

Token	PERSON	ORG	GPE	DATE	O
The	0.01	0.01	0.01	0.02	0.95
cat	0.10	0.02	0.01	0.02	0.85
...					

5 Using SpaCy for NER

Overview

SpaCy provides pre-trained NER models that are easy to use and highly accurate for common entity types.

5.1 Basic SpaCy NER

```
import spacy
from spacy import displacy

# Load pre-trained model
nlp = spacy.load("en_core_web_sm")

# Process text
text = """Donald Trump won more than 50 electoral votes this year.
Tesla's stock rose 2.4% after the Federal Reserve announcement."""

doc = nlp(text)

```

```python
# Extract entities
for ent in doc.ents:
    print(f"{ent.text:20} {ent.label_:10} {ent.start_char}-{ent.end_char}")
```

Listing 5: SpaCy NER Basics

Output:

```
Donald Trump         PERSON     0-12
more than 50         CARDINAL   17-29
this year            DATE       47-56
Tesla                ORG        58-63
2.4%                 PERCENT    78-82
Federal Reserve      ORG        94-109
```

5.2 Visualizing Entities

```python
# Render in Jupyter notebook
displacy.render(doc, style="ent", jupyter=True)

# Or save to HTML file
html = displacy.render(doc, style="ent", page=True)
with open("ner_visualization.html", "w") as f:
    f.write(html)
```

Listing 6: Visualize NER with displacy

5.3 SpaCy NER Label Reference

Label	Description
PERSON	People, including fictional
NORP	Nationalities, religious, political groups
FAC	Facilities (buildings, airports, highways)
ORG	Companies, agencies, institutions
GPE	Countries, cities, states
LOC	Non-GPE locations
PRODUCT	Objects, vehicles, foods
EVENT	Named hurricanes, battles, wars
WORK_OF_ART	Titles of books, songs
LAW	Named documents made into laws
DATE	Absolute or relative dates
TIME	Times smaller than a day
PERCENT	Percentage
MONEY	Monetary values
QUANTITY	Measurements
ORDINAL	“first”, “second”, etc.
CARDINAL	Numerals not falling into other categories

6 Fine-Tuning SpaCy NER

Overview

Pre-trained NER models may not recognize domain-specific entities. SpaCy allows fine-tuning on custom data to add new entity types or improve accuracy.

6.1 When to Fine-Tune

Important

Consider fine-tuning when:

- Domain-specific entities (drug names, product codes)
- New entity categories not in default model
- Improving accuracy for your specific text type
- Handling industry jargon or technical terms

6.2 Training Data Format

```
# Training data format: (text, {"entities": [(start, end, label)]})
train_data = [
    ("Cars in China are selling well",
     {"entities": [(0, 4, "VEHICLE")]}),

    ("Tesla has a lot on the line as an electric vehicle maker",
     {"entities": [(0, 5, "ORG"), (40, 56, "VEHICLE")]}),

    ("My family loves our Honda Civic",
     {"entities": [(23, 34, "VEHICLE")]}),

    ("This car is the best",
     {"entities": [(5, 8, "VEHICLE")]})
]
```

Listing 7: SpaCy Training Data Format

6.3 Fine-Tuning Process

```
import spacy
from spacy.training import Example
import random

# Load existing model
nlp = spacy.load("en_core_web_sm")

# Get the NER component
ner = nlp.get_pipe("ner")

# Add new entity label
ner.add_label("VEHICLE")

# Disable other pipes during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]

# Training loop
```

```python
with nlp.disable_pipes(*other_pipes):
    optimizer = nlp.resume_training()

    for iteration in range(20):
        random.shuffle(train_data)
        losses = {}

        for text, annotations in train_data:
            doc = nlp.make_doc(text)
            example = Example.from_dict(doc, annotations)
            nlp.update([example], drop=0.5, losses=losses)

        print(f"Iteration {iteration}, Losses: {losses}")

# Test the model
doc = nlp("I bought a Toyota Camry yesterday")
for ent in doc.ents:
    print(f"{ent.text}: {ent.label_}")
```

Listing 8: Fine-Tuning SpaCy NER

Fine-Tuning Pitfalls

- **Catastrophic forgetting**: Model may "forget" original entities
- **Insufficient data**: Need many examples per entity type
- **Label consistency**: Annotations must be consistent
- Always include some original data in training to prevent forgetting

7 Part-of-Speech Tagging

Overview

Part-of-Speech (POS) tagging identifies grammatical categories (noun, verb, adjective, etc.) for each word. It's closely related to NER and often used as a preprocessing step.

7.1 Common POS Tags

Tag	Description	Example
NN	Noun, singular	cat, dog, house
NNS	Noun, plural	cats, dogs, houses
NNP	Proper noun, singular	John, London
VB	Verb, base form	run, eat, be
VBD	Verb, past tense	ran, ate, was
VBG	Verb, gerund	running, eating
JJ	Adjective	big, red, beautiful
RB	Adverb	quickly, very, well
IN	Preposition	in, on, at, by
DT	Determiner	the, a, an
CC	Coordinating conjunction	and, but, or
TO	"to"	to (as in "to run")

7.2 POS Tagging with NLTK

```python
import nltk
from nltk import word_tokenize, pos_tag

# Download required data
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('tagsets')

# View tag descriptions
nltk.help.upenn_tagset('NN')   # Noun
nltk.help.upenn_tagset('VB')   # Verb

# POS tagging
text = "The quick brown fox jumps over the lazy dog"
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)

print(pos_tags)
# [('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'),
#  ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'),
#  ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]
```

Listing 9: POS Tagging with NLTK

7.3 POS Tagging with SpaCy

```python
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("The quick brown fox jumps over the lazy dog")

for token in doc:
    print(f"{token.text:10} {token.pos_:6} {token.tag_}")
```

Listing 10: POS Tagging with SpaCy

8 Enhancing Classification with NER

Overview

NER features can improve text classification by providing structured information about document content.

8.1 Feature Engineering with NER

NER-Based Features

Count-based features:

- Number of persons mentioned
- Number of organizations
- Number of locations
- Number of dates/times

Binary indicators:

- Contains person name? (0/1)
- Contains organization? (0/1)
- Contains specific entity (e.g., “Tesla”)? (0/1)

8.2 Implementation Example

```python
import spacy
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
import numpy as np

nlp = spacy.load("en_core_web_sm")

def extract_ner_features(text):
    """Extract NER-based features from text"""
    doc = nlp(text)

    features = {
        'n_persons': 0,
        'n_orgs': 0,
        'n_gpes': 0,
        'n_dates': 0,
        'n_money': 0,
        'n_percent': 0
    }

    for ent in doc.ents:
        if ent.label_ == 'PERSON':
            features['n_persons'] += 1
        elif ent.label_ == 'ORG':
            features['n_orgs'] += 1
        elif ent.label_ == 'GPE':
            features['n_gpes'] += 1
        elif ent.label_ == 'DATE':
            features['n_dates'] += 1
        elif ent.label_ == 'MONEY':
            features['n_money'] += 1
        elif ent.label_ == 'PERCENT':
            features['n_percent'] += 1

    return features

# Extract NER features for all documents
ner_features = [extract_ner_features(text) for text in documents]
ner_df = pd.DataFrame(ner_features)

# Combine TF-IDF and NER features
tfidf = TfidfVectorizer(max_features=20)
X_tfidf = tfidf.fit_transform(documents).toarray()
X_combined = np.hstack([X_tfidf, ner_df.values])

# Train classifier
X_train, X_test, y_train, y_test = train_test_split(
    X_combined, labels, test_size=0.2, random_state=42
)

clf = MLPClassifier(hidden_layer_sizes=(50,), max_iter=500)
```

```
clf.fit(X_train, y_train)
accuracy = clf.score(X_test, y_test)
print(f"Accuracy with NER features: {accuracy:.4f}")
```

Listing 11: NER Feature Enhancement

8.3 Entity-Based Binary Features

```
def get_entity_dummies(documents):
    """Create binary indicators for each unique entity"""
    all_entities = set()

    # First pass: collect all unique entities
    for text in documents:
        doc = nlp(text)
        for ent in doc.ents:
            all_entities.add((ent.text, ent.label_))

    # Second pass: create binary features
    feature_matrix = []
    for text in documents:
        doc = nlp(text)
        doc_entities = set((ent.text, ent.label_) for ent in doc.ents)

        row = [1 if entity in doc_entities else 0
               for entity in all_entities]
        feature_matrix.append(row)

    columns = [f"{text}_{label}" for text, label in all_entities]
    return pd.DataFrame(feature_matrix, columns=columns)

entity_features = get_entity_dummies(documents)
```

Listing 12: Binary Entity Indicators

9 One-Page Summary

Summary

Named Entity Recognition (NER) identifies and classifies named entities in text.
Common Entity Types:

- PERSON, ORG, GPE, LOC, DATE, TIME, MONEY, PERCENT, CARDINAL

Two Approaches:
Rule-Based:

- Uses regular expressions and dictionaries
- No training data needed
- Limited by predefined patterns
- No context awareness

Statistical/Neural:

- Learns from labeled data
- Uses context for disambiguation

- CNNs capture local context via filters
- SpaCy: `nlp = spacy.load("en_core_web_sm")`

NER as Sequence Labeling:

- BIO scheme: B-TYPE (begin), I-TYPE (inside), O (outside)
- Each token gets a tag
- Output: softmax probabilities over entity types

SpaCy Usage:

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Donald Trump visited Tesla.")
for ent in doc.ents:
    print(ent.text, ent.label_)
```

Fine-Tuning: Add custom entity types with labeled examples. Watch for catastrophic forgetting.
Feature Enhancement:

- Add entity counts to feature vectors
- Create binary entity indicators
- Combine with TF-IDF for classification

Applications: Translation, information extraction, question answering, sentiment analysis, search.

10 Glossary

Key Terms

- **NER**: Named Entity Recognition—identifying entities in text
- **Named Entity**: Real-world object with a name (person, organization, place)
- **BIO Tagging**: Begin-Inside-Outside scheme for sequence labeling
- **POS Tagging**: Part-of-Speech tagging—grammatical categories
- **Regular Expression**: Pattern matching syntax for text
- **Rule-Based NER**: Pattern-matching approach to entity extraction
- **Statistical NER**: Machine learning approach to entity extraction
- **HMM**: Hidden Markov Model—probabilistic sequence model
- **CRF**: Conditional Random Fields—discriminative sequence model
- **SpaCy**: Industrial-strength NLP library for Python
- **Fine-Tuning**: Continuing training on domain-specific data

- **Catastrophic Forgetting**: Model losing original knowledge during fine-tuning
- **GPE**: Geopolitical Entity (countries, cities, states)
- **NORP**: Nationalities, religious, or political groups
- **displacy**: SpaCy's visualization module
- **NLTK**: Natural Language Toolkit—Python NLP library
- **Context**: Surrounding words that help disambiguate meaning
- **Word Boundary**: `\b` in regex—edges of words

Lecture Information

Course: CSCI E-89B: Natural Language Processing
Lecture: Lecture 11
Topic: Sequence Models: HMMs, CRFs, and Generative Models
Date: Fall 2024

Contents

1 Introduction to Sequence Models

Overview

This lecture covers probabilistic sequence models used in NLP: Markov Chains, Hidden Markov Models (HMMs), and Conditional Random Fields (CRFs). We also explore how to combine these with neural networks (BiLSTM-CRF) and introduce generative models like VAEs and GANs.

2 Markov Chains

Overview

Markov chains model sequences where the probability of each element depends only on the previous element. This "memoryless" property, called the Markov property, simplifies modeling but limits expressiveness.

2.1 Definition and Structure

Markov Chain

A **Markov chain** is a sequence of random variables $X_1, X_2, \ldots, X_T$ satisfying the **Markov property**:

$$P(X_{t+1}|X_t, X_{t-1}, \ldots, X_1) = P(X_{t+1}|X_t)$$

The future depends only on the present, not on the past.

Weather as Markov Chain

States: {Sunny, Rainy, Cloudy}
Transition probabilities:

From/To	Sunny	Rainy	Cloudy
Sunny	0.7	0.1	0.2
Rainy	0.2	0.5	0.3
Cloudy	0.3	0.3	0.4

Each row sums to 1 (must transition somewhere).

2.2 Transition Probabilities

Transition Matrix

For states $\{1, 2, \ldots, N\}$, the **transition probability** from state i to state j is:

$$P_{ij} = P(X_{t+1} = j|X_t = i)$$

Constraints:

- $P_{ij} \geq 0$ (non-negative)
- $\sum_{j=1}^{N} P_{ij} = 1$ (rows sum to 1)

2.3 NLP Application: Language Modeling

Bigram Language Model

Treat words as states. Transition probabilities model word sequences:

$$P(\text{“sat”}|\text{“cat”}) = 0.15$$

A sentence's probability:

$$P(\text{“the cat sat”}) = P(\text{the}) \cdot P(\text{cat}|\text{the}) \cdot P(\text{sat}|\text{cat})$$

2.4 Limitations of Markov Chains

The Markov Assumption is Limiting

- Only previous word matters—ignores long-range dependencies
- “The cat that I saw yesterday sat” vs “The cats that I saw yesterday sat”
- Solution: Use n-grams (state = last n-1 words) or neural models

3 Hidden Markov Models (HMMs)

Overview

HMMs extend Markov chains by introducing **hidden states** that generate **observations**. We observe the outputs but not the underlying state sequence.

3.1 Motivation

Why Hidden States?

In language, we observe words but not their underlying structure:

- **Observed**: “The cat sat on the mat”
- **Hidden**: DET NOUN VERB PREP DET NOUN (part-of-speech tags)

The hidden states (POS tags) follow Markov dynamics, and each state “emits” an observed word.

3.2 HMM Structure

Hidden Markov Model Components

An HMM consists of:

1. **Hidden states**: $Y_1, Y_2, \ldots, Y_T$ (e.g., POS tags)
2. **Observations**: $X_1, X_2, \ldots, X_T$ (e.g., words)
3. **Transition probabilities**: $P(Y_{t+1} = j | Y_t = i) = A_{ij}$
4. **Emission probabilities**: $P(X_t = x | Y_t = j) = B_{jx}$

5. **Initial state distribution**: $\pi_i = P(Y_1 = i)$

POS Tagging as HMM

Hidden states: {NOUN, VERB, DET, ADJ, PREP, ...}
Observations: {the, cat, sat, on, mat, ...}
Transitions (e.g.):

- $P(\text{VERB}|\text{NOUN}) = 0.35$ (nouns often followed by verbs)
- $P(\text{NOUN}|\text{DET}) = 0.60$ (determiners often followed by nouns)

Emissions (e.g.):

- $P(\text{“cat”}|\text{NOUN}) = 0.002$
- $P(\text{“sat”}|\text{VERB}) = 0.001$

3.3 HMM Parameters

HMM Parameter Summary

$$\lambda = (A, B, \pi)$$

where:

- A: Transition matrix ($N \times N$ for N hidden states)
- B: Emission matrix ($N \times V$ for vocabulary size V)
- π: Initial state distribution (length N)

3.4 Training HMMs

Important

If hidden states are observed (supervised training):

- Count transitions and emissions directly
- Estimate probabilities via maximum likelihood

If hidden states are NOT observed (unsupervised):

- Use **Baum-Welch algorithm** (a form of EM)
- E-step: Estimate expected counts of transitions/emissions
- M-step: Update parameters from expected counts

3.5 Decoding: Finding Hidden States

Viterbi Algorithm

Given observations $X_1, \ldots, X_T$, find the most likely hidden state sequence:

$$\hat{Y}_{1:T} = \arg\max_{Y_{1:T}} P(Y_{1:T}|X_{1:T})$$

The **Viterbi algorithm** uses dynamic programming to find this efficiently in $O(T \cdot N^2)$ time.

3.6 HMM Limitations

HMM Limitations

1. **Markov assumption**: Hidden state depends only on previous state
2. **Independence assumption**: Observation depends only on current hidden state
3. **No future context**: Can't use future words to help label current word

4 Conditional Random Fields (CRFs)

Overview

CRFs address HMM limitations by modeling $P(Y|X)$ directly (discriminative) rather than the joint $P(X, Y)$ (generative). They allow arbitrary features and bidirectional dependencies.

4.1 From HMMs to CRFs

Key Differences: HMM vs CRF

Aspect	HMM	CRF
Model type	Generative	Discriminative
Models	$P(X, Y)$	$P(Y\|X)$
Direction	Forward only	Bidirectional
Features	Emissions only	Arbitrary features
Independence	Strong assumptions	Flexible

4.2 CRF Model

Linear-Chain CRF

$$P(Y|X) = \frac{1}{Z(X)} \exp\left(\sum_{t=1}^{T} \sum_{k} \lambda_k f_k(y_{t-1}, y_t, X, t)\right)$$

where:

- f_k: Feature functions (manually designed)
- λ_k: Feature weights (learned)

- $Z(X)$: Normalization constant (partition function)

4.3 Feature Functions

CRF Feature Functions

Feature functions $f_k(y_{t-1}, y_t, X, t)$ encode patterns. They typically return 0 or 1:
Example features for NER:

- $f_1 = 1$ if y_t = PERSON and x_{t-1} = "Mr."
- $f_2 = 1$ if y_t = PERSON and x_t starts with capital letter
- $f_3 = 1$ if y_t = ORG and x_t ends with "Inc."
- $f_4 = 1$ if y_{t-1} = B-PERSON and y_t = I-PERSON

Concrete Feature Example

For the input "Mr. Smith works at Apple Inc.":
Feature: "If previous word is 'Mr.' and current word is capitalized, likely PERSON"

$$f_1(y_{t-1}, y_t, X, t) = \mathbf{1}[x_{t-1} = \text{"Mr."} \wedge x_t[0] \in \text{A-Z} \wedge y_t = \text{PERSON}]$$

The weight λ_1 is learned from data—higher weight means this pattern is more predictive.

4.4 Advantages of CRFs

Important

CRF Advantages:

1. **Arbitrary features**: Include any information about entire input
2. **Global normalization**: Avoids label bias problem
3. **Bidirectional context**: Feature can look at future words
4. **No independence assumptions**: Features can overlap

4.5 Disadvantages of CRFs

CRF Disadvantages

1. **Manual feature engineering**: Must design features by hand
2. **Computational cost**: Training can be expensive
3. **Feature explosion**: Many features needed for good performance

5 BiLSTM-CRF

Overview

BiLSTM-CRF combines the automatic feature learning of neural networks with the sequence modeling of CRFs. The BiLSTM replaces hand-crafted features; the CRF layer captures label dependencies.

5.1 Architecture

BiLSTM-CRF Architecture

1. **Embedding Layer**: Words → dense vectors
2. **BiLSTM Layer**: Captures bidirectional context
3. **Linear Layer**: Maps hidden states to "emission scores"
4. **CRF Layer**: Models label transitions, outputs final labels

5.2 How It Works

Important

Step 1: Embeddings
Each word x_t is mapped to an embedding vector e_t.
Step 2: BiLSTM
Forward and backward LSTMs process the sequence:

$$\overrightarrow{h_t} = \text{LSTM}_{\rightarrow}(e_t, \overrightarrow{h_{t-1}})$$

$$\overleftarrow{h_t} = \text{LSTM}_{\leftarrow}(e_t, \overleftarrow{h_{t+1}})$$

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

Step 3: Emission Scores
Linear transformation produces scores for each label:

$$E_t = W \cdot h_t + b$$

E_t has dimension equal to number of labels (e.g., B-PER, I-PER, O, ...).
Step 4: CRF Layer
CRF uses emission scores E and learns transition matrix T (label-to-label scores). Final prediction maximizes:

$$\text{score}(X, Y) = \sum_{t=1}^{T} E_{t,y_t} + \sum_{t=1}^{T-1} T_{y_t, y_{t+1}}$$

5.3 Why CRF on Top of BiLSTM?

CRF Layer Benefits

Without CRF, BiLSTM predicts each label independently. This can produce invalid sequences like:

- I-PER following O (invalid—can't continue without begin)
- B-LOC immediately after B-PER (missing I-PER)

The CRF layer learns that certain transitions are unlikely (e.g., $T_{\text{O,I-PER}} \ll 0$), enforcing valid sequences.

5.4 Implementation Sketch

```python
import torch
import torch.nn as nn
from torchcrf import CRF

class BiLSTM_CRF(nn.Module):
    def __init__(self, vocab_size, tag_size, embed_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim // 2,
                            bidirectional=True, batch_first=True)
        self.linear = nn.Linear(hidden_dim, tag_size)
        self.crf = CRF(tag_size, batch_first=True)

    def forward(self, x):
        embeds = self.embedding(x)
        lstm_out, _ = self.lstm(embeds)
        emissions = self.linear(lstm_out)
        return emissions

    def loss(self, x, tags):
        emissions = self.forward(x)
        return -self.crf(emissions, tags)  # Negative log-likelihood

    def predict(self, x):
        emissions = self.forward(x)
        return self.crf.decode(emissions)  # Viterbi decoding
```

Listing 1: BiLSTM-CRF in PyTorch (Simplified)

6 Variational Autoencoders (VAEs)

Overview

VAEs are generative models that learn a structured latent space. Unlike regular autoencoders, VAEs can generate new, realistic samples by sampling from the latent space.

6.1 Regular Autoencoder Problem

Unstructured Latent Space

In a regular autoencoder:

- Encoder compresses input to latent code z
- Decoder reconstructs input from z
- Problem: Latent space is **unstructured**

If you move slightly away from a learned encoding, the decoder produces garbage—no smooth interpolation between points.

6.2 VAE Solution

VAE Key Idea

Instead of encoding to a point, encode to a **distribution**:

1. Encoder outputs μ (mean) and σ (standard deviation)
2. Sample $z \sim \mathcal{N}(\mu, \sigma^2)$
3. Decoder reconstructs from sampled z

This forces nearby points in latent space to also decode to realistic outputs.

6.3 VAE Loss Function

VAE Loss

$$\mathcal{L} = \underbrace{\|x - \hat{x}\|^2}_{\text{Reconstruction loss}} + \underbrace{D_{KL}(\mathcal{N}(\mu, \sigma^2) \| \mathcal{N}(0, 1))}_{\text{KL divergence (regularization)}}$$

The KL term forces distributions to stay close to standard normal $\mathcal{N}(0, 1)$.

6.4 Why KL Divergence?

Important

Without KL regularization:

- Network minimizes reconstruction by shrinking $\sigma \to 0$
- Returns to point encoding—loses structure

With KL regularization:

- Forces $\mu \to 0$ and $\sigma \to 1$
- Distributions overlap, creating smooth latent space
- Can interpolate between encodings

7 Generative Adversarial Networks (GANs)

Overview

GANs learn to generate realistic data through adversarial training: a generator tries to fool a discriminator, while the discriminator tries to distinguish real from fake data.

7.1 The Chess Analogy

Learning Without a Teacher

Problem: Train children to play chess without an expert teacher.
Solution: Have them play against each other! Both improve through competition.
GANs work similarly: two networks compete, both improving without labeled data.

7.2 GAN Architecture

GAN Components

Generator G:

- Input: Random noise $z \sim \mathcal{N}(0, 1)$
- Output: Fake sample $G(z)$
- Goal: Generate samples indistinguishable from real data

Discriminator D:

- Input: Real sample x or fake sample $G(z)$
- Output: Probability that input is real
- Goal: Correctly classify real vs fake

7.3 GAN Training

GAN Objective (Minimax Game)

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

- D maximizes: classify real as real, fake as fake
- G minimizes: fool D into thinking fake is real

7.4 Training Procedure

Important

Alternating optimization:

1. **Train D**: Fix G, update D to better distinguish real/fake
2. **Train G**: Fix D, update G to better fool D

3. Repeat until equilibrium

At equilibrium: D outputs 0.5 for everything (can't tell real from fake), G generates perfect samples.

7.5 Mode Collapse Problem

Mode Collapse

GANs can suffer from **mode collapse**:

- Generator finds one output that fools discriminator
- Keeps producing only that output (e.g., only shoes)
- Discriminator adapts, generator switches to another mode
- Cycle continues without learning diversity

Solutions: Experience replay, progressive growing, StyleGAN architecture

8 One-Page Summary

Summary

Markov Chains: Sequence model where $P(X_{t+1}|X_t, \ldots) = P(X_{t+1}|X_t)$. Simple but limited—no long-range dependencies.
Hidden Markov Models (HMMs):

- Hidden states Y generate observations X
- Parameters: transitions A, emissions B, initial π
- Training: Baum-Welch (EM) if hidden states unknown
- Decoding: Viterbi algorithm
- Limitation: Only uses past context

Conditional Random Fields (CRFs):

- Discriminative: models $P(Y|X)$ directly
- Feature functions $f_k(y_{t-1}, y_t, X, t)$ encode patterns
- Bidirectional context, no independence assumptions
- Disadvantage: Manual feature engineering

BiLSTM-CRF:

- BiLSTM provides automatic feature learning
- CRF layer captures label dependencies
- State-of-the-art for NER, POS tagging before transformers

- No manual features needed

Variational Autoencoders (VAEs):

- Encode to distribution (μ, σ), sample, decode
- KL divergence regularization prevents collapse
- Creates structured, interpolatable latent space

GANs:

- Generator vs Discriminator adversarial game
- No labeled data needed—self-supervised
- Mode collapse is common challenge
- Can generate highly realistic images/text

9 Glossary

Key Terms

- **Markov Property**: Future depends only on present, not past
- **HMM**: Hidden Markov Model—hidden states emit observations
- **Transition Probabilities**: $P(Y_{t+1}|Y_t)$
- **Emission Probabilities**: $P(X_t|Y_t)$
- **Viterbi Algorithm**: Dynamic programming for most likely path
- **Baum-Welch**: EM algorithm for HMM parameter estimation
- **CRF**: Conditional Random Field—discriminative sequence model
- **Feature Function**: Pattern indicator in CRF
- **Partition Function**: Normalization constant $Z(X)$
- **BiLSTM**: Bidirectional LSTM—forward and backward context
- **Emission Scores**: BiLSTM output before CRF layer
- **VAE**: Variational Autoencoder—generative with structured latent space
- **KL Divergence**: Measures distribution similarity
- **Latent Space**: Compressed representation space
- **GAN**: Generative Adversarial Network
- **Generator**: Creates fake samples from noise
- **Discriminator**: Classifies real vs fake
- **Mode Collapse**: GAN failure mode—limited diversity

- **Discriminative Model**: Models $P(Y|X)$
- **Generative Model**: Models $P(X, Y)$ or $P(X)$

Attention Mechanism and Transformers

CSCI E-89B: Introduction to Natural Language Processing

Lecture 12

Lecture Information

Course: CSCI E-89B: Introduction to Natural Language Processing
Lecture: 12 – Attention Mechanism and Transformers
Institution: Harvard Extension School
Topics: Attention Mechanism, Query-Key-Value, Multi-Head Attention, Transformers, BERT, GPT

Contents

1 Introduction and Quiz Review

Overview

This lecture introduces two revolutionary concepts in modern NLP: the **attention mechanism** and the **transformer architecture**. These form the foundation of virtually all state-of-the-art language models including BERT, GPT, and ChatGPT.

1.1 Review of Previous Concepts

Before diving into attention mechanisms, let's review key concepts from previous lectures:

Important

Conditional Random Fields vs. Hidden Markov Models:

- CRFs can use more complex dependencies—they can look at joint distributions with **both past and future tokens**
- HMMs assume current state depends **only on the previous state**
- This is not about discrete vs. continuous variables; it's about the directionality of dependencies

Key Quiz Points:

1. **CRF Advantage:** Ability to capture dependencies from both past and future context
2. **LSTM + CRF Benefit:** Enhanced ability to capture both past and future context
3. **Regular Autoencoder Issue:** The encoding space is *not structured*—even slight shifts in encodings can completely destroy the output
4. **Variational Autoencoder:** The mean (μ) is a deterministic function of the input; adding noise makes the output non-deterministic

2 Limitations of Recurrent Neural Networks

2.1 The Problem with Sequential Processing

Definition

Recurrent Neural Network (RNN) Structure:
In a standard RNN, we have:

- Input sequence: $x_1, x_2, \ldots, x_T$
- Hidden states: $h_1, h_2, \ldots, h_T$
- Each hidden state depends on the previous: $h_t = f(h_{t-1}, x_t)$

This sequential nature creates several fundamental problems:

2.1.1 Problem 1: Vanishing Gradients

Warning

As sequences get longer, gradients that flow backward through time become increasingly small. This makes it extremely difficult to learn long-range dependencies.
Analogy: Imagine passing a message through a long chain of people, where each person slightly garbles the message. By the end, the original message is nearly unrecognizable.

Even with LSTM's long-term memory mechanism, the vanishing gradient problem is only *mitigated*, not eliminated.

2.1.2 Problem 2: No Parallel Computation

Warning

Because each hidden state h_t depends on h_{t-1}, we **cannot** compute states in parallel. This makes training extremely slow for long sequences.
Impact: A 1000-word document requires 1000 sequential computations, regardless of how many GPUs you have.

2.1.3 Problem 3: Fixed-Size Bottleneck

In encoder-decoder architectures (like sequence-to-sequence models for translation):

- The entire input sequence is compressed into a **single fixed-dimension vector**
- Whether your sentence is 5 words or 50 words, it must fit into the same size representation (e.g., 128 dimensions)
- This creates an information bottleneck

Example

The Compression Problem:
Consider trying to store different length sentences in a 128-dimensional vector:

- "Hello" $\rightarrow$ 128-dim vector (easy)
- "The quick brown fox jumps over the lazy dog" $\rightarrow$ same 128-dim vector
- A 500-word paragraph $\rightarrow$ still the same 128-dim vector!

Mathematically, there IS enough space (128 dimensions is huge), but practically, learning this mapping is extremely difficult.

2.1.4 Problem 4: Difficulty with Future Context

In a standard RNN:

$$y_2 = f(y_1, x_2) = f(f(y_0, x_1), x_2) \tag{1}$$

The prediction at position 2 depends only on positions 0, 1, and 2. But what if position 2's meaning depends on position 4?

Example

When Future Words Matter:
"I went to the bank to deposit my check."
When translating "bank," you need to see "deposit" (which comes later) to know it's a financial institution, not a river bank.

3 The Attention Mechanism

3.1 Core Idea

Definition

Attention: Instead of compressing the entire input into a single vector, attention allows the model to "look at" all input positions and dynamically decide which positions are most relevant for each output position.
Key Innovation: Create a **context vector** C_t for each time step that is a weighted combination of *all* hidden states.

3.2 Historical Context

The attention mechanism was introduced in the landmark paper:

Info

"Neural Machine Translation by Jointly Learning to Align and Translate"
Bahdanau, Cho, and Bengio (2015)
This paper showed that attention-based models significantly outperform traditional encoder-decoder models for machine translation.

3.3 How Attention Works

3.3.1 Step 1: Compute Hidden States

First, we run a bidirectional RNN (or LSTM) to get hidden states that capture both past and future context:

$$\overrightarrow{h_t} = \text{forward RNN}(x_1, \ldots, x_t) \tag{2}$$

$$\overleftarrow{h_t} = \text{backward RNN}(x_T, \ldots, x_t) \tag{3}$$

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}] \quad \text{(concatenation)} \tag{4}$$

3.3.2 Step 2: Compute Alignment Scores

For each output position t, we compute how much it should "attend" to each input position j:

$$e_{tj} = a(s_{t-1}, h_j) \tag{5}$$

where:

- s_{t-1} is the previous decoder state
- h_j is the encoder hidden state at position j
- a is an **alignment function** (a small neural network)

The alignment function is typically:

$$a(s_{t-1}, h_j) = \tanh(W_s \cdot s_{t-1} + W_h \cdot h_j) \tag{6}$$

3.3.3 Step 3: Convert to Probabilities (Attention Weights)

Apply softmax to get normalized attention weights:

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})} \tag{7}$$

Important

Key Insight: The α values are **not learnable parameters**—they are computed dynamically based on the input! This allows the model to adapt to inputs of any length.

3.3.4 Step 4: Compute Context Vector

The context vector is a weighted sum of all hidden states:

$$\boxed{C_t = \sum_{j=1}^{T} \alpha_{tj} \cdot h_j} \tag{8}$$

3.3.5 Step 5: Generate Output

The decoder uses both the previous state and the context vector:

$$s_t = f(s_{t-1}, y_{t-1}, C_t) \tag{9}$$

3.4 Visualizing Attention

Example

Translation Example: English to French
"The agreement on the European Economic Area was signed in August 1992."
↓ translates to ↓
"L'accord sur la zone économique européenne a été signé en août 1992."
The attention mechanism learns that:

- "zone" (French, position 5) attends strongly to "Area" (English, position 7)
- "1992" attends to "1992" (diagonal alignment)
- "signé" attends to "signed"

This is visualized as a matrix where bright squares indicate high attention weights.

3.5 Benefits of Attention

Summary

Advantages of Attention Mechanism:

1. **Dynamic Context:** No more rigid, fixed-size bottleneck
2. **Long-Range Dependencies:** Can directly connect distant positions
3. **Variable-Length Inputs:** Same architecture works for any sequence length
4. **Interpretability:** Attention weights show what the model focuses on
5. **Better Translation:** Handles word reordering across languages

4 From Attention to Transformers

4.1 The Key Question

With attention, we've solved the bottleneck problem. But we still have recurrent connections. A researcher might ask:

Important

"If attention is so powerful that we're computing relationships between ALL positions anyway, do we even need the recurrent network?"
Answer: **NO!** This insight led to the Transformer architecture.

4.2 The Transformer Paper

Info

"Attention Is All You Need"
Vaswani et al., Google Brain (2017)
This paper introduced the Transformer architecture, which completely removes recurrence and relies solely on attention. It revolutionized NLP and became the foundation for BERT, GPT, and virtually all modern language models.

4.3 Limitations of RNNs with Attention

Even with attention, RNN-based models still have issues:

1. **Sequential Computation:** Still can't parallelize the RNN part
2. **Vanishing Gradients:** Still present in the recurrent connections
3. **Knowledge Transfer:** Difficult to transfer learned representations to new tasks

4.4 Training Cost Perspective

Warning

The Cost of Training:
Training a large Transformer model (like GPT-3) is estimated to cost around **$4.6 million** in compute costs alone!
This is why transfer learning is so important—we want to train once and reuse the knowledge for many downstream tasks.

5 The Transformer Architecture

5.1 High-Level Overview

The Transformer consists of two main components:

1. **Encoder:** Processes the input sequence
2. **Decoder:** Generates the output sequence

Definition

Transformer Architecture Components:
Encoder:

- Input Embeddings
- Positional Encoding
- Multi-Head Self-Attention
- Feed-Forward Network

- Residual Connections + Layer Normalization

Decoder:

- Output Embeddings (shifted right)
- Positional Encoding
- Masked Multi-Head Self-Attention
- Encoder-Decoder Attention
- Feed-Forward Network
- Linear + Softmax Output

5.2 Query, Key, and Value

Important

The heart of the Transformer is the **Query-Key-Value (QKV)** attention mechanism. This is fundamentally different from the earlier attention mechanism.

5.2.1 The Motivation: Context-Dependent Embeddings

Example

The Problem with Static Embeddings:
Consider the word "bank":

- "I went to the bank to deposit my check." → Financial institution
- "I sat by the river bank." → Edge of a river

With traditional embeddings (Word2Vec, GloVe), "bank" has the **same vector** in both cases!
We need **context-dependent representations**.

5.2.2 Defining Q, K, V

Given input embeddings H (a matrix where each row is a token embedding):

$$Q = H \cdot W_Q \quad \text{(Query)} \tag{10}$$
$$K = H \cdot W_K \quad \text{(Key)} \tag{11}$$
$$V = H \cdot W_V \quad \text{(Value)} \tag{12}$$

where W_Q, W_K, W_V are learnable weight matrices.

5.2.3 Intuition for Q, K, V

Definition

Understanding Query, Key, Value:
Think of it like a database lookup:

- **Query (Q):** "What am I looking for?" — Represents the current token asking for information
- **Key (K):** "What do I have to offer?" — Represents each token's identity for matching
- **Value (V):** "What information do I carry?" — The actual content to be retrieved

Analogy: Searching a library

- Query: Your search terms
- Key: Book titles/keywords (used for matching)
- Value: Actual book content (what you get back)

5.2.4 Why Three Different Representations?

Info

Flexibility Through Different Roles:
The same token plays different roles in attention:

1. When it's the **token being updated**: Use Q
2. When it's **helping to update another token**: Use K (for matching) and V (for content)

Having separate transformations gives the model more flexibility to learn different aspects for each role.

5.3 Scaled Dot-Product Attention

The attention scores are computed as:

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V} \tag{13}$$

5.3.1 Step-by-Step Computation

1. **Compute similarity scores:** QK^T gives a matrix of dot products between all query-key pairs
2. **Scale:** Divide by $\sqrt{d_k}$ (dimension of keys) to prevent extremely large values

3. **Softmax:** Convert to probabilities (rows sum to 1)
4. **Weighted sum:** Multiply by V to get the output

Example

Concrete Example: "Clouds drift across blue sky"
Let's trace attention for the word "sky":
Step 1: Compute Q for "sky"

$$Q_{\text{sky}} = H_{\text{sky}} \cdot W_Q = [q_1, q_2, \ldots, q_d] \tag{14}$$

Step 2: Compute scores with all keys

$$\text{score}_{\text{clouds}} = Q_{\text{sky}} \cdot K_{\text{clouds}}^T / \sqrt{d_k} \tag{15}$$
$$\text{score}_{\text{drift}} = Q_{\text{sky}} \cdot K_{\text{drift}}^T / \sqrt{d_k} \tag{16}$$
$$\vdots \tag{17}$$
$$\text{score}_{\text{sky}} = Q_{\text{sky}} \cdot K_{\text{sky}}^T / \sqrt{d_k} \tag{18}$$

Step 3: Apply softmax

$$[\alpha_{\text{clouds}}, \alpha_{\text{drift}}, \alpha_{\text{across}}, \alpha_{\text{blue}}, \alpha_{\text{sky}}] = \text{softmax}([\text{scores}]) \tag{19}$$

Example result: $[0.30, 0.10, 0.10, 0.30, 0.20]$
Step 4: Compute new representation

$$\text{New}_{\text{sky}} = 0.30 \cdot V_{\text{clouds}} + 0.10 \cdot V_{\text{drift}} + \ldots + 0.20 \cdot V_{\text{sky}} \tag{20}$$

The new representation of "sky" now incorporates context from related words like "blue" and "clouds"!

5.3.2 Why Scale by $\sqrt{d_k}$?

Warning

Numerical Stability:
For high-dimensional vectors, dot products can become very large. If $d_k = 512$:

- Expected magnitude of dot product: $\approx \sqrt{512} \approx 22.6$
- Softmax of large values $\rightarrow$ extremely peaked distribution
- Extremely peaked $\rightarrow$ near-zero gradients $\rightarrow$ no learning!

Dividing by $\sqrt{d_k}$ normalizes the variance back to 1.

5.4 Multi-Head Attention

Definition

Multi-Head Attention:
Instead of computing attention once, we compute it multiple times in parallel with different projections:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{21}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{22}$$

Typical configurations use $h = 8$ or $h = 16$ heads.

5.4.1 Why Multiple Heads?

Info

Benefits of Multi-Head Attention:

1. **Different Types of Relationships:** One head might learn syntactic relationships, another semantic
2. **Different Positions:** Different heads can focus on nearby vs. distant tokens
3. **Redundancy:** Multiple chances to capture important patterns
4. **Parallel Computation:** All heads can be computed simultaneously

Analogy: Like having multiple experts look at the same problem from different angles.

5.5 Positional Encoding

Warning

The Order Problem:
Unlike RNNs, Transformers process all positions simultaneously. Without any modification:
"The cat sat on the mat" and "mat the on sat cat The"
would produce **identical representations!**

5.5.1 The Solution: Add Position Information

We add a **positional encoding** to each embedding:

$$\text{Input}_{\text{final}} = \text{Embedding} + \text{PositionalEncoding} \tag{23}$$

The original Transformer uses sinusoidal functions:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{24}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{25}$$

where:

- pos = position in the sequence (0, 1, 2, ...)
- i = dimension index
- d_{model} = embedding dimension

Example

Example: Computing Positional Encoding

For position 0 with $d_{model} = 4$:

$$PE_{(0,0)} = \sin(0) = 0 \tag{26}$$

$$PE_{(0,1)} = \cos(0) = 1 \tag{27}$$

$$PE_{(0,2)} = \sin(0) = 0 \tag{28}$$

$$PE_{(0,3)} = \cos(0) = 1 \tag{29}$$

For position 1:

$$PE_{(1,0)} = \sin(1/10000^0) = \sin(1) \approx 0.84 \tag{30}$$

$$PE_{(1,1)} = \cos(1) \approx 0.54 \tag{31}$$

$$\vdots \tag{32}$$

Each position gets a unique encoding!

5.5.2 Why Sinusoidal?

- **Unique patterns:** Each position has a distinct encoding
- **Relative positions:** The model can learn to attend to relative positions
- **Generalization:** Can extrapolate to longer sequences than seen during training

5.6 Residual Connections (Skip Connections)

Definition

Residual Connection:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{33}$$

We add the input directly to the output of each sub-layer.

5.6.1 Why Residual Connections?

- **Gradient Flow:** Gradients can flow directly through the addition operation
- **Training Stability:** Prevents weights from diverging to extreme values
- **Learning Residuals:** Network learns the *difference* from the identity function

5.7 Masked Attention in the Decoder

Important

The Autoregressive Constraint:
During generation, we can only use information from tokens that have already been generated. We **cannot** look at future tokens!
Solution: Mask out (set to $-\infty$) attention scores for future positions before softmax.

Example

Masking Example:
For "clouds drift across blue sky":
If generating position 3 ("across"):

- Can attend to: clouds, drift, across (✓)
- Cannot attend to: blue, sky (×)

The attention scores for "blue" and "sky" are set to $-\infty$, which becomes 0 after softmax.

6 BERT and GPT: Transformer Applications

6.1 Splitting the Transformer

The full Transformer was designed for sequence-to-sequence tasks like translation. But we can use its parts for different purposes:

Definition

BERT (Bidirectional Encoder Representations from Transformers):

- Uses only the **Encoder** part
- Bidirectional: Can see both past and future context
- Great for: Classification, NER, Question Answering
- Pre-training: Masked Language Model + Next Sentence Prediction

Definition

GPT (Generative Pre-trained Transformer):

- Uses only the **Decoder** part

- Unidirectional: Can only see past context (uses masking)
- Great for: Text generation, completion
- Pre-training: Next token prediction

6.2 The Power of Transfer Learning

Important

Pre-training + Fine-tuning:

1. **Pre-train** on massive text corpora (Wikipedia, books, internet)
2. **Fine-tune** on specific downstream tasks

The expensive pre-training is done once. Fine-tuning is fast and cheap.
Why this works: The model learns general language understanding during pre-training, which transfers to specific tasks.

7 Implementation Details

7.1 Transformer in Code

Listing 1: Multi-Head Attention Layer

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class MultiHeadAttention(layers.Layer):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        self.depth = d_model // num_heads

        # Learnable weight matrices
        self.wq = layers.Dense(d_model)
        self.wk = layers.Dense(d_model)
        self.wv = layers.Dense(d_model)
        self.dense = layers.Dense(d_model)

    def split_heads(self, x, batch_size):
        # Split last dimension into (num_heads, depth)
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, q, k, v, mask=None):
        batch_size = tf.shape(q)[0]
```

```
25
26          # Linear projections
27          q = self.wq(q)  # (batch, seq_len, d_model)
28          k = self.wk(k)
29          v = self.wv(v)
30
31          # Split into heads
32          q = self.split_heads(q, batch_size)  # (batch, heads, seq, depth
                )
33          k = self.split_heads(k, batch_size)
34          v = self.split_heads(v, batch_size)
35
36          # Scaled dot-product attention
37          matmul_qk = tf.matmul(q, k, transpose_b=True)
38          dk = tf.cast(tf.shape(k)[-1], tf.float32)
39          scaled_attention = matmul_qk / tf.math.sqrt(dk)
40
41          if mask is not None:
42              scaled_attention += (mask * -1e9)
43
44          attention_weights = tf.nn.softmax(scaled_attention, axis=-1)
45          output = tf.matmul(attention_weights, v)
46
47          # Concatenate heads
48          output = tf.transpose(output, perm=[0, 2, 1, 3])
49          output = tf.reshape(output, (batch_size, -1, self.d_model))
50
51          return self.dense(output)
```

7.2 Positional Encoding Implementation

Listing 2: Positional Encoding

```
1  import numpy as np
2
3  def get_positional_encoding(max_seq_len, d_model):
4      """Generate␣sinusoidal␣positional␣encodings."""
5      position = np.arange(max_seq_len)[:, np.newaxis]
6      div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) /
           d_model))
7
8      pe = np.zeros((max_seq_len, d_model))
9      pe[:, 0::2] = np.sin(position * div_term)
10     pe[:, 1::2] = np.cos(position * div_term)
11
12     return tf.constant(pe, dtype=tf.float32)
13
14 # Example usage
15 max_len = 100
16 d_model = 512
```

```
pos_encoding = get_positional_encoding(max_len, d_model)
# Add to embeddings: embedded + pos_encoding[:seq_len, :]
```

7.3 Creating the Attention Mask

Listing 3: Creating Look-Ahead Mask for Decoder

```
def create_look_ahead_mask(size):
    """Create mask to prevent attending to future positions."""
    # Upper triangular matrix with ones above diagonal
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
    return mask  # (size, size)

# Example: for sequence length 5
# [[0, 1, 1, 1, 1],
#  [0, 0, 1, 1, 1],
#  [0, 0, 0, 1, 1],
#  [0, 0, 0, 0, 1],
#  [0, 0, 0, 0, 0]]
# 1s become -inf after scaling, 0 after softmax
```

8 Comparing Architectures

Table 1: Comparison of Sequence Modeling Architectures

Feature	RNN/LSTM	RNN + Attention	Transformer
Parallel computation	No	Partial	Yes
Long-range dependencies	Difficult	Better	Excellent
Variable-length inputs	Yes	Yes	Yes
Vanishing gradients	Problem	Problem	Solved
Position awareness	Implicit	Implicit	Explicit (PE)
Training speed	Slow	Slow	Fast
Interpretability	Low	Medium	High

9 Practical Considerations

9.1 Computational Complexity

Warning

Attention is Quadratic:
For sequence length n, self-attention requires $O(n^2)$ computation and memory.

- $n = 100$: 10,000 attention scores
- $n = 1000$: 1,000,000 attention scores
- $n = 10000$: 100,000,000 attention scores!

This is why models like GPT have maximum context lengths (e.g., 4096 tokens).

9.2 Model Dimensions

Info

Typical Transformer Sizes:

- **Original Transformer:** $d_{model} = 512$, 6 layers, 8 heads
- **BERT-base:** $d_{model} = 768$, 12 layers, 12 heads
- **GPT-3:** $d_{model} = 12288$, 96 layers, 96 heads, 175B parameters

10 One-Page Summary

Summary

Key Concepts from Lecture 12:

1. Problems with RNNs:

- Vanishing gradients make learning long-range dependencies difficult
- Sequential processing prevents parallelization
- Fixed-size bottleneck loses information

2. Attention Mechanism (2015):

- Dynamic context vector: $C_t = \sum_j \alpha_{tj} h_j$
- Attention weights α are computed (not learned parameters)
- Allows direct connections between distant positions

3. Transformer Architecture (2017):

- "Attention Is All You Need" — removes recurrence entirely
- Query, Key, Value: $Q = HW_Q$, $K = HW_K$, $V = HW_V$
- Scaled dot-product: Attention $= \text{softmax}(QK^T/\sqrt{d_k})V$
- Multi-head attention for richer representations
- Positional encoding for order information

4. Key Formulas:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$$

5. **BERT vs GPT:**
   - BERT = Encoder only (bidirectional, good for understanding)
   - GPT = Decoder only (unidirectional, good for generation)
6. **Why Transformers Dominate:**
   - Fully parallelizable training
   - No vanishing gradient problem
   - Excellent transfer learning capabilities
   - State-of-the-art on virtually all NLP tasks

11 Glossary

Attention Mechanism
A technique that allows models to focus on relevant parts of the input when producing each part of the output

Query (Q)
The representation of the current token that is "asking" for relevant information

Key (K)
The representation used to determine relevance/matching with the query

Value (V)
The actual content retrieved based on attention weights

Multi-Head Attention
Running multiple attention operations in parallel with different projections

Positional Encoding
Added signal that provides position information to the model

Transformer
Neural architecture using only attention (no recurrence)

Self-Attention
Attention where Q, K, V all come from the same sequence

Masked Attention
Attention that prevents looking at future positions

BERT
Bidirectional Encoder Representations from Transformers

GPT
Generative Pre-trained Transformer

Residual Connection
: Adding the input directly to the output of a layer

Layer Normalization
: Normalizing activations across features for each sample

Context Vector
: Weighted combination of hidden states based on attention

Alignment Score
: Raw similarity between query and key before softmax

Machine Translation: From Rules to Neural Networks

CSCI E-89B: Introduction to Natural Language Processing

Lecture 13

Lecture Information

Course: CSCI E-89B: Introduction to Natural Language Processing
Lecture: 13 – Machine Translation: Theory and Practice
Institution: Harvard Extension School
Topics: Machine Translation, Seq2Seq, Attention, Transformers, BLEU Score, Hybrid Models

Contents

1 Introduction to Machine Translation

Overview

This lecture provides a comprehensive exploration of **Machine Translation (MT)**—the task of automatically converting text from one language to another. We trace the evolution from rule-based systems through statistical methods to modern neural approaches, culminating in the transformer architecture that powers today's state-of-the-art translation systems.

1.1 What is Machine Translation?

Definition

Machine Translation (MT): The use of software to translate text or speech from one natural language to another, with the goal of producing output that is:

- Semantically accurate (preserves meaning)
- Grammatically correct (follows target language rules)
- Culturally appropriate (handles idioms and cultural references)
- Fluent and natural (reads like human-written text)

1.2 Why is Translation Difficult?

Translation is far more complex than word-for-word substitution. Consider the challenges:

1.2.1 1. Lexical Ambiguity

Example

The Word "Bank":

- "I deposited money at the bank." → Financial institution
- "I sat by the river bank." → Edge of water
- "Don't bank on it." → Rely/depend

Without context, the system cannot know which meaning to use!

1.2.2 2. Structural Differences

Different languages have different word orders:

- **English (SVO):** "The cat ate the fish."
- **Japanese/Korean (SOV):** "The cat the fish ate."
- **Arabic/Welsh (VSO):** "Ate the cat the fish."

Warning

The Reordering Problem:
To translate "I will go to school tomorrow" from English to Japanese, the system needs to:

1. Read the entire sentence
2. Understand the structure
3. Rearrange to: "I tomorrow school to will go"

This requires understanding the *entire* sentence before producing output!

1.2.3 3. Idiomatic Expressions

Example

Idioms Cannot Be Translated Literally:

English Idiom	Literal Translation	Correct Translation
"It's raining cats and dogs"	Animals falling	"It's raining heavily"
"Break a leg"	Fracture your bone	"Good luck"
"Shoot the breeze"	Fire at wind	"Chat casually"
"Hit the nail on the head"	Strike metal	"Exactly right"

1.2.4 4. Context and Coreference

Example

Pronoun Resolution:
"The trophy doesn't fit in the suitcase because it is too big."
What does "it" refer to?

- If "it" = trophy → The trophy is too large
- If "it" = suitcase → Wait, that doesn't make sense here

Compare: "The trophy doesn't fit in the suitcase because it is too small."

- Now "it" = suitcase (the suitcase is too small)

Resolving this requires world knowledge and reasoning!

Table 1: Evolution of Machine Translation Approaches

Era	Approach	Key Characteristics
1950s–1980s	Rule-Based (RBMT)	Hand-crafted rules, dictionaries, linguistic knowledge
1990s–2010s	Statistical (SMT)	Learn from parallel corpora, phrase-based, n-gram models
2014–Present	Neural (NMT)	End-to-end deep learning, Seq2Seq, Attention, Transformers

2 Historical Evolution of Machine Translation

2.1 Timeline Overview

2.2 Rule-Based Machine Translation (RBMT)

Definition

RBMT: Translation systems that use hand-crafted linguistic rules, bilingual dictionaries, and grammar specifications created by human experts.

2.2.1 How RBMT Works

1. **Analysis:** Parse the source sentence to understand its grammatical structure
2. **Transfer:** Apply rules to convert source structure to target structure
3. **Generation:** Produce the output sentence following target language grammar

Example

RBMT Rule Example:
English to Spanish rule for adjective placement:

```
IF: English has [Adjective] [Noun]
THEN: Spanish uses [Noun] [Adjective]

Input:  "the red car"
        [Det] [Adj] [Noun]
Output: "el coche rojo"
        [Det] [Noun] [Adj]
```

2.2.2 Advantages of RBMT

- **Precision:** Very accurate for well-defined domains (weather reports, legal documents)
- **Transparency:** Easy to understand why a translation was produced
- **No data required:** Works without large parallel corpora
- **Consistency:** Same input always produces same output

2.2.3 Disadvantages of RBMT

- **Labor-intensive:** Requires years of expert work to create rules
- **Incomplete coverage:** Cannot handle exceptions not anticipated by rule writers
- **Poor generalization:** Rules for one domain don't transfer to others
- **Unnatural output:** Often produces grammatically correct but stilted text

2.3 Statistical Machine Translation (SMT)

Definition

SMT: Translation systems that learn statistical patterns from large collections of parallel texts (texts and their human translations).

2.3.1 The Noisy Channel Model

SMT is based on Bayes' theorem. To find the best translation $\hat{e}$ of foreign sentence f:

$$\hat{e} = \arg\max_e P(e|f) = \arg\max_e P(f|e) \cdot P(e) \tag{1}$$

where:

- $P(f|e)$ = **Translation model**: How likely is f given e?
- $P(e)$ = **Language model**: How likely is e to be a valid sentence?

Example

SMT Intuition:
To translate French "le chat" to English:
Translation model says:

- P("le chat"|"the cat") = 0.8
- P("le chat"|"cat the") = 0.1

Language model says:

- P("the cat") = 0.01 (common phrase)
- P("cat the") = 0.0001 (ungrammatical)

Combined: "the cat" wins because it's both a good translation AND good English.

2.3.2 Phrase-Based SMT

The breakthrough came with **phrase-based models** that translate multi-word phrases as units:

- "in spite of" → "a pesar de" (as one unit)
- Better than word-by-word: "in" → "en", "spite" → "rencor", "of" → "de"

2.3.3 Advantages of SMT

- **Data-driven:** Learns from examples, no manual rules needed
- **Better coverage:** Can handle diverse vocabulary
- **Scalable:** More data = better translations

2.3.4 Disadvantages of SMT

- **Local context only:** Phrases are translated independently, losing global coherence
- **Reordering issues:** Hard to model long-distance word movement
- **Complex pipelines:** Many separate components to tune
- **Data hungry:** Needs millions of sentence pairs

2.4 Neural Machine Translation (NMT)

Definition

NMT: Translation systems that use deep neural networks to learn a direct mapping from source to target language in an end-to-end fashion.

2.4.1 Key Advantages of NMT

- **End-to-end learning:** Single model learns everything jointly
- **Continuous representations:** Words/phrases are vectors, enabling generalization
- **Global context:** Can consider entire sentence when translating each word
- **Fluent output:** Produces more natural-sounding translations

3 Sequence-to-Sequence Models

3.1 Architecture Overview

The Seq2Seq model, introduced in 2014, consists of two main components:

Definition

Seq2Seq Architecture:

1. **Encoder:** Reads the source sentence and compresses it into a fixed-length vector (context vector)
2. **Decoder:** Takes the context vector and generates the target sentence word by word

3.1.1 The Encoder

The encoder processes the input sequence one token at a time:

$$h_t = \text{RNN}(h_{t-1}, x_t) \tag{2}$$

$$c = h_T \quad \text{(final hidden state = context vector)} \tag{3}$$

where:

- x_t = input token at time t
- h_t = hidden state at time t
- c = context vector (summary of entire input)

3.1.2 The Decoder

The decoder generates output tokens one at a time:

$$s_t = \text{RNN}(s_{t-1}, y_{t-1}, c) \tag{4}$$

$$P(y_t|y_{<t}, x) = \text{softmax}(W_o \cdot s_t) \tag{5}$$

where:

- s_t = decoder hidden state
- y_{t-1} = previous output token
- c = context vector from encoder

3.2 The Bottleneck Problem

Warning

Critical Limitation:
The entire source sentence must be compressed into a single fixed-length vector!
Analogy: Imagine reading a 500-page novel and summarizing it in a single sentence, then asking someone to recreate the entire novel from just that sentence.
Consequences:

- Information loss, especially for long sentences
- Performance degrades significantly as sentence length increases
- No way to "go back" and look at specific parts of the input

Example

Empirical Evidence:
Early Seq2Seq models showed BLEU scores dropping dramatically for sentences longer

than 20 words. The context vector simply couldn't capture all the necessary information.

4 Attention Mechanism in Translation

4.1 The Core Idea

Important

Key Insight: Instead of compressing everything into one vector, let the decoder "look back" at the input at each step and focus on the most relevant parts!

Definition

Attention Mechanism: A method that allows the decoder to access all encoder hidden states and compute a weighted average based on relevance to the current decoding step.

4.2 How Attention Works for Translation

4.2.1 Step 1: Compute All Encoder Hidden States

Unlike basic Seq2Seq, we keep ALL hidden states:

$$H = [h_1, h_2, \ldots, h_T] \tag{6}$$

4.2.2 Step 2: Compute Attention Scores

For each decoder step t, compute how relevant each encoder state is:

$$e_{ti} = \text{score}(s_{t-1}, h_i) \tag{7}$$

Common scoring functions:

- **Dot product:** $e_{ti} = s_{t-1}^T h_i$
- **Bilinear:** $e_{ti} = s_{t-1}^T W h_i$
- **Additive:** $e_{ti} = v^T \tanh(W_s s_{t-1} + W_h h_i)$

4.2.3 Step 3: Convert to Probabilities

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{T} \exp(e_{tj})} \tag{8}$$

4.2.4 Step 4: Compute Context Vector

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i \tag{9}$$

4.2.5 Step 5: Generate Output

$$s_t = \text{RNN}(s_{t-1}, [y_{t-1}; c_t]) \tag{10}$$

$$P(y_t) = \text{softmax}(W_o[s_t; c_t]) \tag{11}$$

4.3 Attention Visualization

Example

English to French Translation:
Source: "The agreement on the European Economic Area was signed in August 1992"
When generating the French word "zone" (area), the attention mechanism assigns high weight to the English word "Area" and lower weights to other words.
This creates an **alignment matrix** showing which source words are most relevant for each target word.

	The	agreement	...	Area	1992
L'	**0.9**	0.05	...	0.01	0.01
accord	0.1	**0.8**	...	0.02	0.01
...	...	...	...	...	...
zone	0.01	0.05	...	**0.85**	0.02
1992	0.01	0.01	...	0.01	**0.95**

4.4 Benefits of Attention

Summary

Why Attention Revolutionized Translation:

1. **No bottleneck:** Each decoding step has access to all encoder states
2. **Long sentences:** Performance doesn't degrade with length
3. **Alignment learning:** Model learns word correspondences automatically
4. **Interpretability:** Can visualize what the model focuses on
5. **Variable-length handling:** Works for any sequence length

5 Transformers for Translation

5.1 From RNN+Attention to Pure Attention

Important

The key insight of Transformers: If attention is so powerful, **do we even need the RNN?**
Answer: **No!** The 2017 paper "Attention Is All You Need" showed that a model using *only* attention mechanisms can outperform RNN-based models.

5.2 Advantages Over RNN-Based Models

Table 2: Transformer Advantages for Translation

Feature	RNN+Attention	Transformer
Parallel training	No	Yes
Long-range dependencies	Difficult	Easy
Training speed	Slow	Fast
Gradient flow	Problematic	Stable

5.3 Self-Attention in Translation

In transformers, **self-attention** allows each word to attend to all other words in the same sentence:

Example

Self-Attention Example:
"The animal didn't cross the street because it was too tired."
Self-attention helps the model learn that "it" refers to "animal" (not "street") by allowing direct connections between these words.

5.4 The Encoder-Decoder Structure

Definition

Transformer for Translation:

- **Encoder:** 6 identical layers of (Self-Attention + Feed-Forward)
- **Decoder:** 6 identical layers of (Masked Self-Attention + Encoder-Decoder Attention + Feed-Forward)

5.4.1 Encoder Self-Attention

Every position in the source sentence can attend to every other position:

- Word "bank" can see "deposit" later in the sentence
- This helps disambiguate word meanings

5.4.2 Masked Self-Attention in Decoder

During generation, we can only attend to previously generated words:

- Prevents "cheating" by looking at future words
- Implemented by masking future positions with $-\infty$ before softmax

5.4.3 Encoder-Decoder Attention

Each decoder layer attends to the encoder output:

- Query: current decoder state
- Keys and Values: encoder outputs
- This is analogous to traditional attention over the source

5.5 Multi-Head Attention for Translation

Using multiple attention heads allows the model to capture different types of relationships:

Example

What Different Heads Learn:

- Head 1: Syntactic relationships (subject-verb agreement)
- Head 2: Semantic relationships (synonyms, related concepts)
- Head 3: Position-based patterns (nearby words)
- Head 4: Long-range dependencies (coreference)

5.6 Positional Encoding for Word Order

Since transformers process all positions simultaneously, word order must be explicitly encoded:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{12}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{13}$$

Info

Why Sinusoidal Functions?

- Create unique encoding for each position
- Allow model to learn relative positions
- Generalize to longer sequences than seen during training

6 Evaluating Machine Translation: BLEU Score

6.1 The Need for Automatic Evaluation

Human evaluation is the gold standard but:

- Expensive and time-consuming
- Not scalable for rapid development
- Subjective and inconsistent

Definition

BLEU (Bilingual Evaluation Understudy): An automatic metric that measures how similar machine translation output is to human reference translations.

6.2 How BLEU Works

6.2.1 Step 1: N-gram Precision

Count how many n-grams in the machine translation appear in the reference:

$$p_n = \frac{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}(\text{n-gram})} \tag{14}$$

where $\text{Count}_{\text{clip}}$ prevents counting the same reference n-gram multiple times.

Example

Calculating Unigram Precision:
Reference: "The cat sat on the mat"
Candidate: "The the the the"
Without clipping: precision = 4/4 = 100% (wrong!)
With clipping: "the" appears 2 times in reference
Clipped count = min(4, 2) = 2
Precision = 2/4 = 50%

6.2.2 Step 2: Brevity Penalty

Prevent gaming the system by outputting very short translations:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{15}$$

where c = candidate length, r = reference length.

6.2.3 Step 3: Final BLEU Score

$$\boxed{\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)} \tag{16}$$

Typically $N = 4$ and $w_n = 1/4$ (uniform weights).

Example

BLEU Score Interpretation:

BLEU Score	Quality Level
< 10	Almost useless
$10 - 19$	Hard to understand
$20 - 29$	Gist is clear
$30 - 39$	Understandable
$40 - 49$	Good quality
$50 - 59$	Very good quality
$60+$	Near human quality

Note: Human translators typically score 60-80 on BLEU against other humans.

6.3 Limitations of BLEU

Warning

BLEU Has Significant Limitations:

1. **Synonym blindness:** "beautiful" vs "pretty" are both correct but BLEU penalizes
2. **Word order flexibility:** Some languages allow multiple valid orderings
3. **Meaning ignorance:** Can't detect semantic errors
4. **Single reference bias:** One reference may not cover all valid translations

Example

BLEU Failure Case:
Reference: "The quick brown fox jumps over the lazy dog."
Candidate A: "A fast brown fox leaps over a lazy dog." (Semantically perfect)
BLEU: **Low** (different words)
Candidate B: "The quick brown jumps fox over lazy the dog." (Nonsensical)
BLEU: **Higher** (more matching n-grams!)

6.4 Alternative Metrics

- **METEOR:** Considers synonyms and stemming
- **TER (Translation Edit Rate):** Number of edits needed

- **COMET:** Neural metric trained on human judgments
- **BERTScore:** Uses BERT embeddings for semantic similarity

7 Hybrid Translation Systems

7.1 Why Combine Approaches?

Definition

Hybrid MT: Systems that combine multiple translation approaches (neural, statistical, rule-based) to leverage their respective strengths.

7.1.1 Weaknesses of Pure NMT

- **Rare words:** May hallucinate translations for uncommon terms
- **Domain-specific terminology:** Medical/legal terms need exact translations
- **Consistency:** May translate the same term differently
- **"Hallucination":** Sometimes generates fluent but completely wrong content

7.1.2 Strengths of Rule-Based Components

- **Precision:** Exact translation of technical terms
- **Consistency:** Same term always translates the same way
- **Controllability:** Can enforce specific rules

7.2 Hybrid Architecture Examples

1. **Pre-processing:** Use rules to identify and protect special terms before NMT
2. **Post-processing:** Use rules to fix grammar or terminology after NMT
3. **Ensemble:** Combine outputs from multiple systems
4. **Terminology injection:** Force specific translations into NMT output

Example

Medical Translation Pipeline:

1. **Step 1 (Rule-based):** Identify medical terms ("myocardial infarction")
2. **Step 2 (NMT):** Translate general text fluently
3. **Step 3 (Rule-based):** Replace medical terms with verified translations
4. **Step 4 (Rule-based):** Check grammar and formatting

Result: Fluent translation with guaranteed accurate medical terminology.

7.3 Real-World Hybrid Systems

- **SYSTRAN:** Pioneer in hybrid MT, combines neural with rules
- **Microsoft Translator:** Uses neural models with terminology databases
- **DeepL:** Primarily neural but with extensive post-processing
- **Google Translate:** Neural with fallbacks for rare languages

8 Practical Implementation

8.1 Building a Simple Seq2Seq Model

Listing 1: Basic Seq2Seq Encoder-Decoder

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Encoder
class Encoder(keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units):
        super().__init__()
        self.embedding = layers.Embedding(vocab_size, embedding_dim)
        self.lstm = layers.LSTM(enc_units, return_state=True)

    def call(self, x):
        x = self.embedding(x)
        output, state_h, state_c = self.lstm(x)
        return state_h, state_c

# Decoder with Attention
class Decoder(keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units):
        super().__init__()
        self.embedding = layers.Embedding(vocab_size, embedding_dim)
        self.lstm = layers.LSTM(dec_units, return_sequences=True,
                                return_state=True)
        self.fc = layers.Dense(vocab_size)
        self.attention = layers.Attention()

    def call(self, x, encoder_output, states):
        x = self.embedding(x)
        context = self.attention([x, encoder_output])
        x = tf.concat([context, x], axis=-1)
        output, state_h, state_c = self.lstm(x, initial_state=states)
        output = self.fc(output)
        return output, state_h, state_c
```

8.2 Using Pre-trained Translation Models

Listing 2: Using Hugging Face Transformers

```python
from transformers import MarianMTModel, MarianTokenizer

# Load pre-trained English to French model
model_name = 'Helsinki-NLP/opus-mt-en-fr'
tokenizer = MarianTokenizer.from_pretrained(model_name)
model = MarianMTModel.from_pretrained(model_name)

# Translate
text = "Machine translation has come a long way."
inputs = tokenizer(text, return_tensors="pt", padding=True)
translated = model.generate(**inputs)
output = tokenizer.decode(translated[0], skip_special_tokens=True)
print(output)
# "La traduction automatique a parcouru un long chemin."
```

8.3 Computing BLEU Score

Listing 3: BLEU Score Calculation

```python
from nltk.translate.bleu_score import sentence_bleu, corpus_bleu

# Single sentence BLEU
reference = [['the', 'cat', 'sat', 'on', 'the', 'mat']]
candidate = ['the', 'cat', 'is', 'on', 'the', 'mat']
score = sentence_bleu(reference, candidate)
print(f"Sentence BLEU: {score:.4f}")

# Corpus BLEU (multiple sentences)
references = [[['the', 'quick', 'brown', 'fox']],
              [['jumped', 'over', 'the', 'dog']]]
candidates = [['the', 'fast', 'brown', 'fox'],
              ['jumped', 'over', 'a', 'dog']]
corpus_score = corpus_bleu(references, candidates)
print(f"Corpus BLEU: {corpus_score:.4f}")
```

9 Current State and Future Directions

9.1 State-of-the-Art Systems

Info

Leading Translation Systems (2024):

- **Google Translate:** 100+ languages, neural-based
- **DeepL:** Known for quality in European languages

- **Microsoft Translator:** Integrated into Office products
- **GPT-4/Claude:** Large language models with translation capabilities

9.2 Remaining Challenges

1. **Low-resource languages:** Many languages lack training data
2. **Document-level translation:** Maintaining coherence across paragraphs
3. **Multimodal translation:** Combining text, speech, and images
4. **Real-time translation:** Simultaneous interpretation
5. **Cultural adaptation:** Beyond literal translation

9.3 Emerging Trends

- **Multilingual models:** Single model for many language pairs
- **Zero-shot translation:** Translating between languages not seen together in training
- **Large language models:** GPT-4 and similar models as general-purpose translators
- **Human-in-the-loop:** Interactive translation with human feedback

10 One-Page Summary

Summary

Key Concepts from Lecture 13:

1. Machine Translation Evolution:

- **RBMT** (1950s-80s): Hand-crafted rules, precise but limited
- **SMT** (1990s-2010s): Statistical patterns from parallel data
- **NMT** (2014-present): End-to-end neural networks

2. Translation Challenges:

- Lexical ambiguity (bank = financial vs. river)
- Structural differences (SVO vs. SOV word order)
- Idioms (literal translation fails)
- Context and coreference resolution

3. Seq2Seq Models:

- Encoder → Context Vector → Decoder
- Problem: Bottleneck—everything compressed into one vector

- Solution: Attention mechanism

4. **Attention:**
   - Dynamic weighted sum of encoder states
   - Context vector: $c_t = \sum_i \alpha_{ti} h_i$
   - Enables handling of long sentences
5. **Transformers:**
   - "Attention Is All You Need" (2017)
   - Self-attention for parallel processing
   - Multi-head attention for diverse relationships
   - Foundation for BERT, GPT, modern NMT
6. **BLEU Score:**

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

   - Measures n-gram overlap with reference
   - Brevity penalty prevents short outputs
   - Limitation: ignores synonyms and meaning
7. **Hybrid Systems:**
   - Combine NMT fluency with rule-based precision
   - Important for domain-specific translation (medical, legal)

11 Glossary

Machine Translation (MT)
Automatic conversion of text from one language to another

RBMT
Rule-Based Machine Translation using hand-crafted linguistic rules

SMT
Statistical Machine Translation learning from parallel corpora

NMT
Neural Machine Translation using deep learning

Seq2Seq
Sequence-to-Sequence model with encoder-decoder architecture

Encoder
Component that reads and encodes source sentence

Decoder
Component that generates target sentence

Context Vector
Fixed-length representation of input sequence

Attention
Mechanism allowing decoder to focus on relevant input parts

Transformer
Architecture using only attention, no recurrence

Self-Attention
Attention where sequence attends to itself

Multi-Head Attention
Multiple parallel attention operations

BLEU Score
Automatic metric measuring translation quality

N-gram Precision
Fraction of n-grams matching reference

Brevity Penalty
BLEU component penalizing short translations

Hybrid MT
Systems combining multiple translation approaches

Parallel Corpus
Collection of texts with human translations

Alignment
Correspondence between source and target words

12 Learning Checklist

- ☐ Can explain the differences between RBMT, SMT, and NMT
- ☐ Understand why translation is difficult (ambiguity, structure, idioms, context)
- ☐ Can describe Seq2Seq architecture and its bottleneck problem
- ☐ Understand how attention solves the bottleneck problem
- ☐ Know why Transformers are better than RNN-based models
- ☐ Can explain BLEU score calculation and its limitations
- ☐ Understand the role of hybrid systems in specialized domains