■ **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