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 ($\sim$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

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

```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., $784 \rightarrow 30$)
3. Train small classifier on compressed representations
4. Much fewer parameters = needs much less labeled data!

Example: Fashion MNIST

- Input: $28 \times 28 = 784$ 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

```python
# 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

```python
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

```
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