XCS224N Notes

Christopher Stewart — NLP with Deep Learning — Spring 2026

This document captures high-level ideas from Stanford Engineering's NLP with Deep Learning class (XCS224N) in Spring 2026. The document is organized by module. The course contains 10 modules and 5 assignments.

1 Exploring Word Embeddings

1.1 Word Embeddings

A **word embedding** is a dense vector representation of a word, typically in 50–300 dimensions. Unlike one-hot encodings (which are sparse and treat every word as equally different), embeddings encode semantic meaning: words with similar meanings end up close together in vector space.

Embeddings are the input layer for nearly every NLP model. Whether you're doing classification, generation, or retrieval, the quality of your embeddings directly affects downstream performance. Pre-trained embeddings (Word2Vec, GloVe, FastText) can be used off-the-shelf or fine-tuned.

1.2 Two Approaches to Building Embeddings

1.2.1 Count-Based: Co-occurrence Matrices

Build a matrix $\mathbf{M}$ where entry M_{ij} counts how often word i appears near word j within a context window of size n. Words that appear in similar contexts get similar row vectors.

Key properties: The matrix is symmetric ($M_{ij} = M_{ji}$). It's very large (vocabulary size squared) and sparse. Dimensionality reduction via techniques like SVD are used to make it feasible to use such approaches.

Context window: For a window size of n, you look at n words to the left and n to the right of each center word. Words near the edges of a document have smaller windows.

1.2.2 Prediction-Based: Word2Vec

Instead of counting co-occurrences, Word2Vec trains a shallow neural network to predict context words from a center word (Skip-gram) or vice versa (CBOW). The learned weight matrix becomes the embedding.

Key differences from co-occurrence: Word2Vec is trained on very large corpora (billions of words), captures more nuanced relationships, and produces dense vectors directly without needing SVD. However, it produces a single vector per word, so polysemous words (multiple meanings) are blended into one representation, one that is typically dominated by the most frequent meaning.

1.3 SVD and Dimensionality Reduction

Singular Value Decomposition factors any matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$ into three pieces:

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

$\mathbf{U}$ contains the left singular vectors (one per row/word), $\mathbf{D}$ is diagonal with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ measuring each component's importance, and $\mathbf{V}$ contains the right singular vectors.

The outer product form shows $\mathbf{A}$ as a sum of layers: $\mathbf{A} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$. Each layer $\sigma_i \mathbf{u}_i \mathbf{v}_i^T$ is a rank-1 matrix capturing one pattern. The first layer captures the most important pattern, the second adds the next, and so on.

Truncated SVD keeps only the top k layers: $\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T = \mathbf{U}_k \mathbf{D}_k \mathbf{V}_k^T$. This is the best rank-$k$ approximation of $\mathbf{A}$ (minimizes Frobenius norm error). In practice, reducing a co-occurrence matrix from thousands of dimensions to $k = 100$–300 produces useful word vectors.

Implementing SVD from scratch is uncommon. `sklearn.decomposition.TruncatedSVD` handles it. The key insight is understanding *why* it works: it finds the directions of maximum variance in the data and projects onto them.

1.4 Measuring Word Similarity

Cosine similarity measures the angle between two word vectors, ignoring magnitude:

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \, \|\mathbf{v}\|}$$

Ranges from -1 (opposite) to 1 (identical direction). This is preferred over Euclidean distance because it's invariant to vector length — a word that appears 10x more often shouldn't be considered 10x more "meaningful."

Cosine distance $= 1 - \cos(\mathbf{u}, \mathbf{v})$. Smaller distance means more similar.

Surprising finding: Antonyms often have *small* cosine distance (high similarity) because they appear in very similar contexts ("the weather is hot/cold", "the movie was good/bad"). Word vectors capture *distributional* similarity, not semantic opposition.

1.5 Word Analogies

Word embeddings capture relational structure. The classic example:

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

This works because the vector from "man" to "king" encodes the concept of "royalty," and adding it to "woman" lands near "queen." Analogies are solved by finding the word whose vector is closest (by cosine similarity) to the target expression.

Limitations: Analogies work well for certain relationships (gender, country-capital, tense) but fail for others. They also reflect biases in the training corpus (e.g., associating certain professions with genders).

1.6 Bias in Word Vectors

Word vectors inherit biases from their training data. If the corpus associates certain races, genders, or groups with particular properties, those associations are encoded in the vectors.

Any downstream model using biased embeddings will propagate those biases. Debiasing techniques exist but are imperfect. Being aware of this is essential when deploying NLP systems in production.

1.7 Key Takeaways for Later Modules

- Nearly every NLP architecture starts by converting words to vectors.
- Pre-trained embeddings (Word2Vec, GloVe) give you a strong starting point; fine-tuning adapts them to your task.
- Contextual embeddings like BERT and GPT, covered in later modules, solve the polysemy problem by giving each *occurrence* of a word a different vector based on its context.

- SVD and matrix factorization ideas recur in recommender systems, topic models, and understanding attention mechanisms.
- The "distributional hypothesis" (words are defined by their context) underlies both co-occurrence and Word2Vec, and later, transformer-based models.

Math Foundations Quick Reference

Vectors and Matrices: A vector $\mathbf{u} \in \mathbb{R}^n$ is a list of n numbers. A matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$ has n rows and d columns. The transpose $\mathbf{A}^T$ flips rows and columns.

Outer Product: A column vector times a row vector: $\mathbf{u}\mathbf{v}^T$ produces an $n \times d$ matrix. Each entry $(i, j) = u_i \cdot v_j$. This creates a rank-1 matrix — a single "pattern."

Orthonormality: Vectors $\mathbf{v}_1, \ldots, \mathbf{v}_r$ are orthonormal if $\mathbf{v}_i^T\mathbf{v}_j = 0$ when $i \neq j$ (perpendicular) and $\mathbf{v}_i^T\mathbf{v}_i = 1$ (unit length). In SVD, both $\mathbf{U}$ and $\mathbf{V}$ have orthonormal columns. This property is what makes terms cancel cleanly when you multiply by a single $\mathbf{v}_i$.

Projection: The projection of a vector $\mathbf{a}$ onto a subspace spanned by orthonormal $\mathbf{v}_1, \ldots, \mathbf{v}_k$ is $\sum_{i=1}^{k}(\mathbf{a}^T\mathbf{v}_i)\mathbf{v}_i$. In matrix form: $\mathbf{A}\mathbf{V}_k\mathbf{V}_k^T$ projects all rows of $\mathbf{A}$ at once. Truncated SVD *is* this projection.

Frobenius Norm: $\|\mathbf{M}\|_F = \sqrt{\sum_{i,j} M_{ij}^2}$ — the "size" of a matrix, analogous to vector length. Truncated SVD minimizes $\|\mathbf{A} - \mathbf{B}\|_F$ over all rank-k matrices $\mathbf{B}$.

Diagonal Matrix Multiplication: Multiplying $\mathbf{U}\mathbf{D}$ where $\mathbf{D}$ is diagonal just scales each column of $\mathbf{U}$: column i gets multiplied by σ_i. No mixing of columns occurs.

2 Word Vectors and Word2Vec

2.1 Core Activation Functions

Sigmoid. Maps any real number to $(0, 1)$. Used for binary gates and as a building block in many architectures.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \text{Derivative: } \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

The derivative being expressible in terms of σ itself is useful: once you've computed the forward pass, the backward pass is nearly free. Numerically, when x is very negative, compute $\sigma(x) = \frac{e^x}{1+e^x}$ instead to avoid overflow in e^{-x}.

Softmax. Generalizes sigmoid to V classes. Takes a vector of raw scores (called *logits*) $\mathbf{z} \in \mathbb{R}^V$ and converts it into a probability distribution. The formula for the probability assigned to word w is:

$$\hat{y}_w = \frac{\exp(z_w)}{\sum_{w'=1}^{V} \exp(z_{w'})}$$

Here's how to read this, step by step. The numerator $\exp(z_w)$ means "take $e \approx 2.718$ and raise it to the power of the score for word w." This turns every score—positive, negative, or zero—into a positive number. The denominator does the same thing for *every* word in the vocabulary and sums them up. Dividing the numerator by that sum guarantees all the probabilities are positive and add to 1.

Exponentiation amplifies differences: a score twice as large gets disproportionately more probability, so the highest-scored word dominates. In practice, subtract $\max(\mathbf{z})$ from all entries before exponentiating for numerical stability (this doesn't change the result because it cancels out in the fraction).

Example: Suppose we have $V = 3$ words with logits $\mathbf{z} = [2.0,\ 1.0,\ 0.5]$.

1. Exponentiate each score: $\exp(2.0) = 7.389, \quad \exp(1.0) = 2.718, \quad \exp(0.5) = 1.649$.
2. Sum the results: $7.389 + 2.718 + 1.649 = 11.756$.
3. Divide each by the sum:

$$\hat{y}_1 = \frac{7.389}{11.756} = 0.629, \quad \hat{y}_2 = \frac{2.718}{11.756} = 0.231, \quad \hat{y}_3 = \frac{1.649}{11.756} = 0.140$$

So the model assigns 62.9% probability to word 1, 23.1% to word 2, and 14.0% to word 3. Notice that $0.629 + 0.231 + 0.140 = 1.0$.

2.2 Cross-Entropy Loss

The standard loss for classification. Given a true one-hot distribution $\mathbf{y}$ and predicted distribution $\hat{\mathbf{y}}$:

$$J = -\sum_w y_w \log(\hat{y}_w) = -\log(\hat{y}_o)$$

where o is the index of the correct class. The simplification follows because $\mathbf{y}$ is one-hot ($y_o = 1$, all others 0). Intuition: penalizes the model by $-\log$ of the probability it assigned to the correct answer. Confident and correct $\rightarrow$ low loss. Confident and wrong $\rightarrow$ very high loss.

2.3 The Error Signal Pattern

A recurring motif in deep learning is that when you combine softmax with cross-entropy loss, the gradient takes the elegant form $(\hat{\mathbf{y}} - \mathbf{y})$. This is the *error signal*, i.e. the difference between what the model predicted and the truth. For the correct class, the entry is $(\hat{y}_o - 1) < 0$ (model didn't assign enough probability). For all other classes, the entry is $\hat{y}_w > 0$ (model assigned too much). This pattern appears in logistic regression, word2vec, and the output layers of most neural networks.

2.4 Word2Vec: Skip-Gram Model

Core idea: Learn word vectors by training a model to predict context words from a center word. Each word has two vectors: a center vector $\mathbf{v}_c$ (used when the word is the center) and an outside vector $\mathbf{u}_w$ (used when it's a context word).

Prediction: The probability of outside word o given center word c:

$$P(o \mid c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)}$$

The dot product $\mathbf{u}_o^\top \mathbf{v}_c$ measures alignment between vectors. Higher alignment $\rightarrow$ higher predicted probability.

Training: For each (center, context) pair, compute cross-entropy loss and update both vectors via SGD. The skip-gram treats each context word independently and sums losses across all context positions.

Gradients: Both gradients flow from the same error signal:

$$\frac{\partial J}{\partial \mathbf{v}_c} = \mathbf{U}^\top (\hat{\mathbf{y}} - \mathbf{y}) \qquad \frac{\partial J}{\partial \mathbf{U}} = (\hat{\mathbf{y}} - \mathbf{y})\, \mathbf{v}_c^\top$$

2.5 Negative Sampling

Naive softmax requires summing over the entire vocabulary for every training step, which is prohibitively expensive for large vocabularies (V can be 10^5–10^6). Negative sampling approximates this by only considering the true outside word and K randomly sampled "negative" words:

$$J = -\log \sigma(\mathbf{u}_o^\top \mathbf{v}_c) - \sum_{k=1}^{K} \log \sigma(-\mathbf{u}_k^\top \mathbf{v}_c)$$

First term: push $\mathbf{v}_c$ toward $\mathbf{u}_o$ (the true context word). Remaining terms: push $\mathbf{v}_c$ away from K random words. Computational cost drops from $O(V)$ to $O(K)$ per training step, with $K = 5$–20 typical in practice.

2.6 Stochastic Gradient Descent

The workhorse optimization algorithm. At each step, estimate the gradient from a small batch and update:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \, \nabla_{\mathbf{x}} J$$

where η is the learning rate. Key practical considerations: (1) learning rate annealing (reduce η over time to converge more precisely); (2) the gradient estimate is noisy but unbiased, which actually helps escape local minima; (3) modern variants (Adam, AdaGrad) adapt η per-parameter, but vanilla SGD remains competitive with proper tuning.

2.7 Engineering Takeaways

Softmax + cross-entropy is the default output layer for classification. The clean $(\hat{\mathbf{y}} - \mathbf{y})$ gradient makes implementation straightforward and numerically stable.

Negative sampling exemplifies a broader principle: when an exact computation is too expensive, sample a few "contrastive" examples instead. This idea recurs in contrastive learning, noise-contrastive estimation, and retrieval-augmented generation.

Dot-product similarity $(\mathbf{u}^\top \mathbf{v})$ as the basis for matching queries to candidates is the foundation of embedding-based retrieval, recommendation systems, and attention mechanisms in transformers.

Two-vector architectures (separate embeddings for different roles) appear beyond word2vec: query/key vectors in attention, user/item vectors in recommendation, and bi-encoder models in semantic search.