

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 $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r$ 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 $k = 100\text{--}300$ 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 $\mathbf{v}_1, \dots, \mathbf{v}_r$ 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 $\mathbf{v}_1, \dots, \mathbf{v}_k$ 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{UD} 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.

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

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: $\exp(2.0) = 7.389$, $\exp(1.0) = 2.718$, $\exp(0.5) = 1.649$.
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}_o (used when it's a context word).

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

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

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:

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

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.

3 Dependency Parsing and Neural Network Training

3.1 What Is Dependency Parsing?

A **dependency parser** analyzes the grammatical structure of a sentence by establishing binary, asymmetric relationships between words. Each relationship is an arrow from a **head** (governor or some other term depending on your syntactic / semantic school of thought) word to a **dependent** (modifier, *idem* for the previous terminological choice comment) word. For example, in “the cat sat,” “sat” is the head of the sentence (for the purposes of this class!), “cat” is its subject dependent, and “the” depends on “cat” as a determiner. A special ROOT node is added so that every word, including the main verb, is a dependent of exactly one head. The resulting structure is a tree.

This contrasts with **constituency parsing**, which groups words into nested phrases (NP, VP, etc.). Dependency structures are more natural for languages with free word order and are the basis for the Universal Dependencies annotation standard used across 100+ languages. The two approaches are not mutually exclusive, but dependency parsing has become dominant in modern NLP because it pairs well with neural methods and produces structures that are easy for downstream tasks to consume.

3.2 Approaches to Dependency Parsing

There are several families of algorithms for building dependency trees. The key distinction is the trade-off between accuracy and speed.

Graph-based parsers: These parsers treat parsing as a search over all possible trees. Given a sentence of n words, you assign a score to every possible head-dependent pair and then find the tree that maximizes the total score. The Eisner algorithm and the Chu-Liu-Edmonds algorithm are classic examples. These methods consider global structure (the entire tree at once) and can enforce well-formedness constraints, but they are slower, typically $O(n^3)$ or worse.

Transition-based parsers: Such parsers build the tree incrementally through a sequence of local decisions. At each step, the parser looks at its current state and chooses one action (shift a word, create a left arc, or create a right arc). This process is fast, i.e. linear time $O(n)$, but each decision is made greedily without seeing the full picture. The risk is error propagation: one bad decision early on can cascade through the rest of the parse.

Neural transition-based parsers: This final class combines the speed of transition-based parsing with the representation power of neural networks. Instead of hand-crafting indicator features, which is how traditional transition-based parsers worked, a neural network learns dense feature representations from the parser’s configuration and predicts the next transition. This is the approach used in Chen and Manning (2014). Implementing a neural transition-based parsers was the primary task in Assignment 3.

3.3 Transition-Based Parsing: The Arc-Standard System

The parser maintains three data structures at every step: a **stack** σ of words currently being processed (initialized with just ROOT), a **buffer** β of words waiting to be processed (initialized with all words of the sentence in order), and a set of **dependencies** A accumulated so far (initially empty). Formally, a parser state is the triple $c = (\sigma, \beta, A)$.

There are three transitions that move between states:

Shift: Remove the first word from the buffer and push it onto the stack. Precondition: the buffer is non-empty.

Left-Arc: The word on top of the stack becomes the head of the second-to-top word. Add the arc to A and remove the second-to-top word from the stack. Precondition: the stack has at least two items and the second-to-top item is not ROOT.

Right-Arc: The second-to-top word on the stack becomes the head of the top word. Add the arc to A and remove the top word from the stack. Precondition: the stack has at least two items.

Parsing terminates when the buffer is empty and only ROOT remains on the stack. At that point, A contains the complete dependency tree. For a sentence of n words, exactly $2n$ transitions are needed (each word is shifted once and removed by an arc once), giving the $O(n)$ runtime.

Example. Consider the sentence “parse this sentence.” The transition sequence S, S, S, LA, RA, RA produces the dependencies: (sentence, this), (parse, sentence), (ROOT, parse). The first three shifts move all words onto the stack. The left-arc makes “sentence” the head of “this.” The two right-arcs then attach “sentence” to “parse” and “parse” to ROOT.

3.4 From Features to Neural Networks

Traditional transition-based parsers used hand-engineered **indicator features**: binary flags like “is the top-of-stack word a verb?” or “does the first buffer word start with a capital letter?” These features were sparse (most are zero for any given configuration), required extensive domain expertise to design, and could not capture interactions between features without explicitly adding cross-product features, which made the feature space enormous.

The Chen and Manning (2014) neural dependency parser replaced this with a simple feedforward network that takes dense, continuous representations as input. The key insight is that word embeddings already encode rich information about each word, and a neural network can learn useful feature interactions automatically through its hidden layer.

3.5 Constructing the Input

The input to the neural network is a **feature vector** extracted from the current parser configuration. This vector is a list of m token identifiers $[w_1, w_2, \dots, w_m]$, where each w_i is the index of a specific token in the vocabulary. These tokens include items like the top three words on the stack, the first three words in the buffer, the leftmost and rightmost children of the top two stack words, and so on. There are 36 features in total in the assignment.

Each token index is looked up in a pre-trained **embedding matrix** $\mathbf{E} \in \mathbb{R}^{|V| \times d}$, where $|V|$ is the vocabulary size and d is the embedding dimension. The resulting m embedding vectors are concatenated into a single input vector:

$$\mathbf{x} = [\mathbf{E}_{w_1}, \mathbf{E}_{w_2}, \dots, \mathbf{E}_{w_m}] \in \mathbb{R}^{md}$$

With $m = 36$ features and $d = 50$ -dimensional embeddings, the input is a 1800-dimensional vector. This is still much smaller and denser than the millions of sparse indicator features used in traditional parsers.

3.6 The Network Architecture

The parser uses a single-hidden-layer feedforward network. Given the concatenated input \mathbf{x} , the computation is:

$$\mathbf{h} = \text{ReLU}(\mathbf{x}\mathbf{W} + \mathbf{b}_1) \quad \mathbf{l} = \mathbf{h}\mathbf{U} + \mathbf{b}_2 \quad \hat{\mathbf{y}} = \text{softmax}(\mathbf{l})$$

where $\mathbf{W} \in \mathbb{R}^{md \times h}$ projects the input to a hidden layer of size h , $\mathbf{U} \in \mathbb{R}^{h \times 3}$ projects the hidden layer to 3 output classes (one per transition), and $\hat{\mathbf{y}}$ gives the predicted probability of each transition.

ReLU (Rectified Linear Unit) is the activation function $\text{ReLU}(z) = \max(z, 0)$. It introduces non-linearity, allowing the network to learn feature interactions that a linear model cannot. Without the non-linearity, stacking linear layers would collapse to a single linear transformation. ReLU is preferred over sigmoid for hidden layers because it avoids the *vanishing gradient problem*: for positive inputs, its derivative is exactly 1, so gradients flow through without shrinking. For negative inputs the derivative is 0, which can cause “dead neurons,” but in practice ReLU works well and is computationally cheap.

3.7 Key PyTorch Components

nn.Linear. A linear layer implements the transformation $\mathbf{y} = \mathbf{x}\mathbf{W}^T + \mathbf{b}$. When you write `nn.Linear(in, out)`, PyTorch creates a weight matrix of shape `(out, in)` and a bias vector of shape `(out,)`. The matrix multiplication projects the input from `in` dimensions down to `out` dimensions. You don't construct the weight matrix yourself. Instead, the layer manages it internally and makes it available for gradient computation.

Xavier Uniform Initialization. The default random initialization of weight matrices can cause activations to explode or vanish as they propagate through layers. Xavier initialization draws weights from a uniform distribution scaled by the fan-in and fan-out of the layer. Fan-in is the number of inputs to a layer (how many values come in). Fan-out is the number of outputs (how many values go out):

$$W_{ij} \sim \text{Uniform}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

This keeps the variance of activations roughly constant across layers, which helps the network train from the start. In PyTorch, `nn.init.xavier_uniform(layer.weight)` applies this initialization in place.

Dropout. During training, dropout randomly sets a fraction p of the hidden units to zero on each forward pass. This prevents the network from relying too heavily on any single neuron and acts as a form of regularization. During evaluation, dropout is turned off and all units are active. In PyTorch, calling `model.train()` enables dropout and `model.eval()` disables it.

nn.CrossEntropyLoss. Combines softmax and cross-entropy loss into a single, numerically stable operation. Given raw logits \mathbf{l} and the index of the correct class o , it computes:

$$J = -\log\left(\frac{\exp(l_o)}{\sum_i \exp(l_i)}\right) = -l_o + \log\left(\sum_i \exp(l_i)\right)$$

The second form avoids computing softmax explicitly, which prevents numerical overflow. This is why the network's forward pass returns logits (raw scores) rather than probabilities — `CrossEntropyLoss` handles the softmax internally.

3.8 The Training Loop

Training a neural network follows a four-step cycle repeated for every minibatch:

- 1. Forward pass.** Feed the input through the network to produce logits. In PyTorch, calling `model(input)` invokes the model's `forward` method, which chains together the embedding lookup, linear layers, ReLU, and dropout.
- 2. Compute loss.** Compare the predicted logits to the true labels using cross-entropy loss. The loss is a single scalar measuring how far the predictions are from the truth.
- 3. Backpropagation.** Calling `loss.backward()` computes the gradient of the loss with respect to every parameter in the network. PyTorch tracks all operations during the forward pass in a computational graph and walks backward through it, applying the chain rule at each node. This is the key insight of backpropagation: you don't derive gradients by hand for each layer; the chain rule decomposes the total gradient into a product of local derivatives that can be computed layer by layer.
- 4. Parameter update.** The optimizer (Adam in this assignment) uses the computed gradients to update each weight. Adam maintains per-parameter running averages of the gradient and its square, which gives it adaptive learning rates. Parameters that have been updated a lot get smaller steps, and vice versa. Before the next iteration, `optimizer.zero_grad()` clears the accumulated gradients so they don't carry over.

Minibatching. Rather than processing one example at a time (slow) or the entire dataset at once (memory-intensive), we process batches of examples simultaneously. This exploits GPU parallelism and

produces gradient estimates that are less noisy than single-example SGD but still computationally tractable. The assignment uses a batch size of 1024 and the minibatch dependency parsing algorithm processes multiple partial parses in parallel, feeding the current batch of parser configurations to the model and applying the predicted transitions simultaneously.

3.9 Evaluation: Unlabeled Attachment Score

The standard metric for dependency parsing is **UAS** (Unlabeled Attachment Score): the percentage of words that are assigned the correct head. “Unlabeled” means we only check whether the arc exists, not what grammatical relation it represents (subject, object, etc.). The labeled variant (LAS) additionally requires the relation label to be correct.

In the assignment, the neural parser achieves over 87% UAS on the dev set after full training (about one hour). For comparison, the original Chen and Manning (2014) paper reported 92.5% UAS, and modern parsers using contextual embeddings from transformers exceed 96%.

3.10 Key Takeaways for Later Modules

- The feedforward parser is a *fixed-context* model: it sees a snapshot of the current configuration (a fixed number of stack/buffer positions) and makes a prediction. This limits what it can capture. RNNs (next module) will address this by processing sequences of arbitrary length, maintaining a hidden state that summarizes everything seen so far.
- The embedding lookup followed by a linear layer is the same pattern that appears everywhere in deep NLP. Transformers scale this idea up with *attention*, which dynamically selects which embeddings to focus on rather than concatenating a fixed set.
- Backpropagation through a computational graph is how all modern deep learning frameworks (PyTorch, TensorFlow, JAX) compute gradients. Understanding the chain rule at the level of individual layers is essential for debugging training issues, designing custom architectures, and reasoning about gradient flow problems like vanishing or exploding gradients.
- Dropout, Xavier initialization, and the Adam optimizer are standard tools in the NLP w/ Deep Learning toolkit. They recur in virtually every model from this point forward, including RNNs, transformers, and LLMs.
- The progression from sparse hand-engineered features to dense learned representations is a recurring theme in NLP’s history. The same shift happened in machine translation (phrase tables to seq2seq), sentiment analysis (bag-of-words to LSTMs), and question answering (template matching to reading comprehension models). Pre-training and fine-tuning (covered in later modules) take this further by learning representations from massive unlabeled corpora.