Word Embeddings

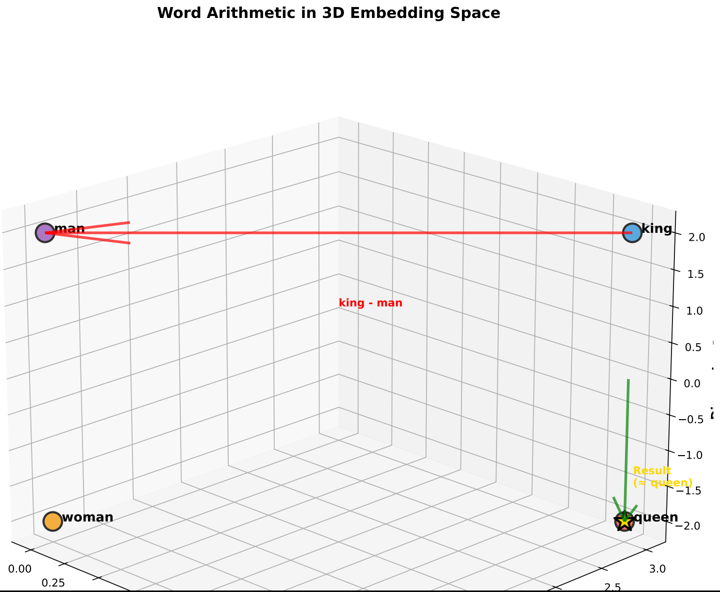
Week 2 - When Words Became Vectors

NLP Course 2025

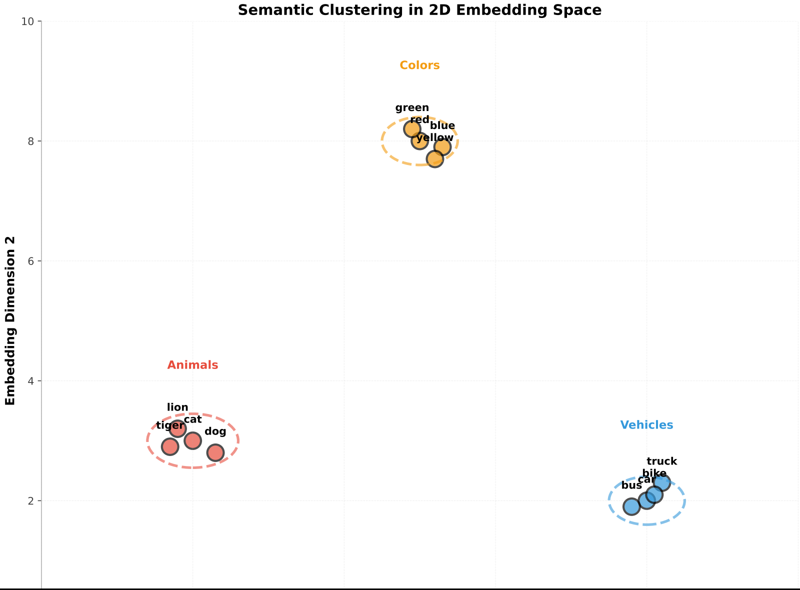
October 27, 2025

Two-Tier BSc Discovery Presentation

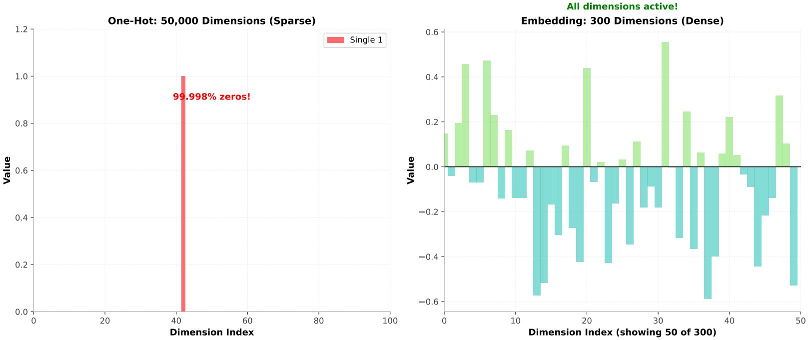
Hook #1: Words That Do Algebra



Hook #2: Similarity That N-grams Miss



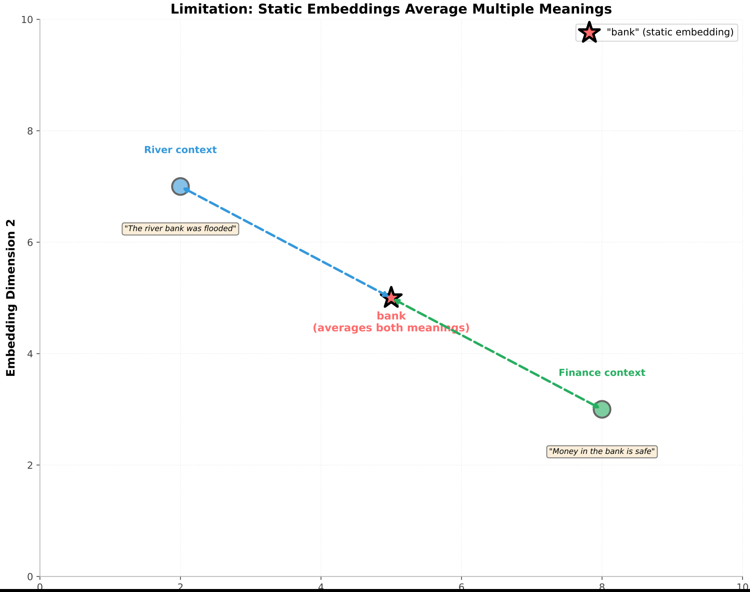
Hook #3: Compression That Improves Quality



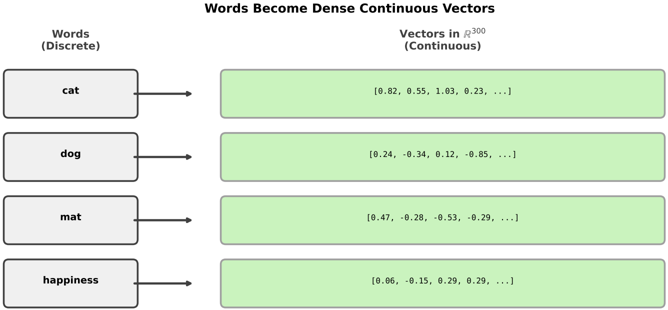
Key Insight: 50,000 sparse $\rightarrow$ 300 dense with MORE information

Dense representations are more powerful than sparse ones

Hook #4: Words With Multiple Meanings



What Are Word Embeddings?



Definition: Dense, low-dimensional, continuous vector representations of words

Words become points in semantic space

From Sparse One-Hot to Dense Embeddings

One-Hot Encoding (Old Way):
Each word $=$ vector of size $|V|$

Example ($V = 5$):

- "cat" $=$ [1, 0, 0, 0, 0]
- "dog" $=$ [0, 1, 0, 0, 0]
- "mat" $=$ [0, 0, 1, 0, 0]

Problems:

- Huge dimensionality (50K typical)
- All words equally distant
- No similarity information
- Sparse (99.998% zeros)

Dense Embeddings (New Way):
Each word $=$ vector of size d (300 typical)

Example ($d = 3$):

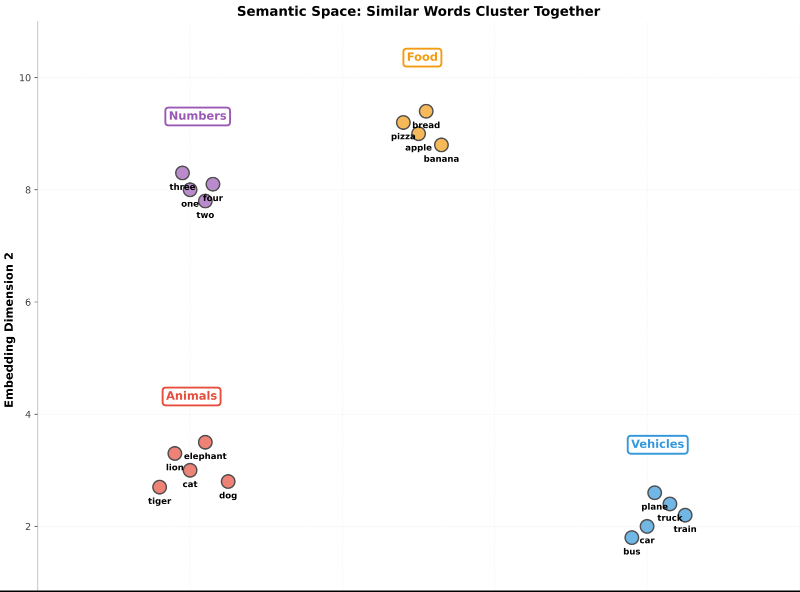
- "cat" $=$ [0.2, 0.8, -0.3]
- "dog" $=$ [0.1, 0.7, -0.2]
- "mat" $=$ [-0.5, 0.1, 0.6]

Advantages:

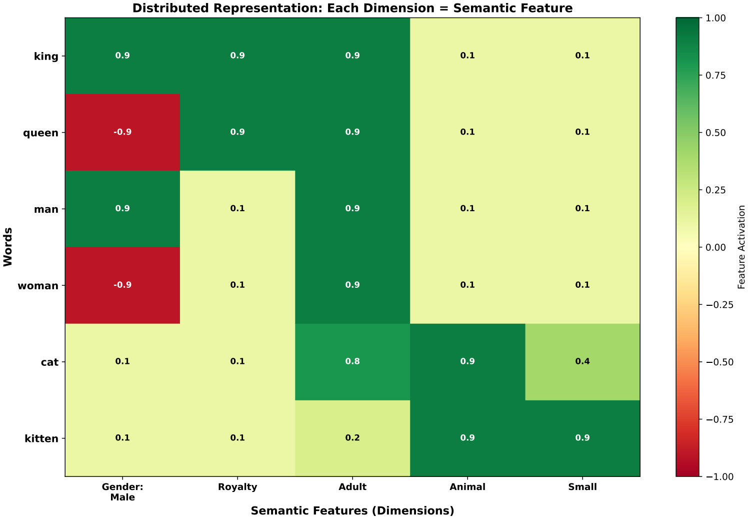
- Low dimensionality (100-300)
- Similarity encoded (cat $\approx$ dog)
- Continuous values
- Information-dense

Dense $<$ Sparse but contains MORE information - this is the magic

The Vector Space Model



Why Distributed Representations Work



Key Insight: Each dimension captures some semantic feature

Word2Vec: The Core Idea

"You shall know a word by the company it keeps"

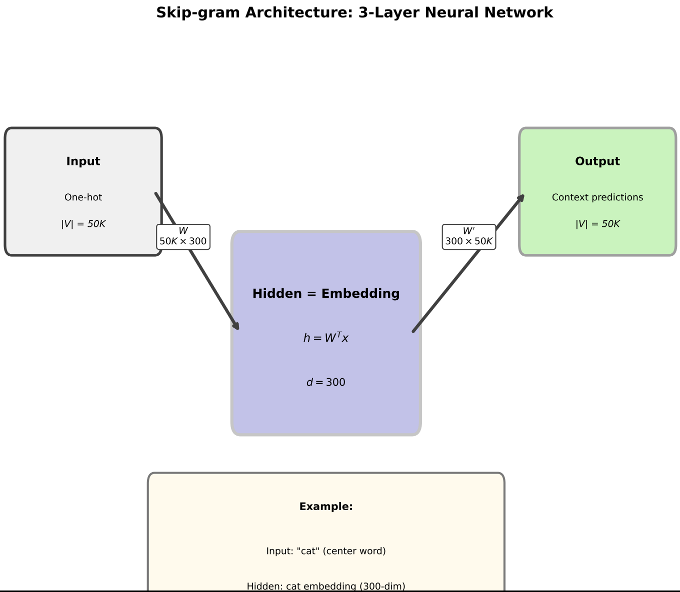
- J.R. Firth (1957)

Distributional Hypothesis:
Words appearing in similar contexts have similar meanings

Word2Vec Approach:
Learn word vectors by predicting context

Context prediction forces similar words to have similar vectors

Skip-gram: Predict Context from Word



Skip-gram: The Architecture

Input: Center word (one-hot)
$x \in \mathbb{R}^{|V|}$

Hidden Layer: Embedding lookup
$h = W^T x \in \mathbb{R}^d$
This IS the word embedding!

Output Layer: Context predictions
$y = W' h \in \mathbb{R}^{|V|}$
Softmax for probabilities

Parameters:

- W: $|V| \times d$ (input embeddings)
- W': $d \times |V|$ (output weights)

Training Objective:
Maximize probability of context words

$$\max \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j} | w_t)$$

Example:
Sentence: "The cat sat on the mat"
Center: "cat"
Context window $c = 2$:
Predict: "the", "sat"

Key Trick:
Share embeddings! W is what we keep after training

Simple 3-layer network - embeddings are the weights

Worked Example: Skip-gram Forward Pass

Given: "The cat sat", center = "cat", predict "sat"

Step 1: One-hot encode center word
"cat" = word ID 3797
$x = [0, 0, \ldots, 1_{3797}, \ldots, 0] \in \mathbb{R}^{50000}$

Step 2: Embedding lookup (hidden layer)

$$h = W^T x = W_{3797} \in \mathbb{R}^{300}$$

This is just row 3797 of W! (Lookup, no multiplication needed)
Example: $h = [0.23, -0.41, 0.15, \ldots, 0.08]$

Step 3: Compute output scores

$$score(\text{``}sat'') = W'_{sat} \cdot h = 0.85$$

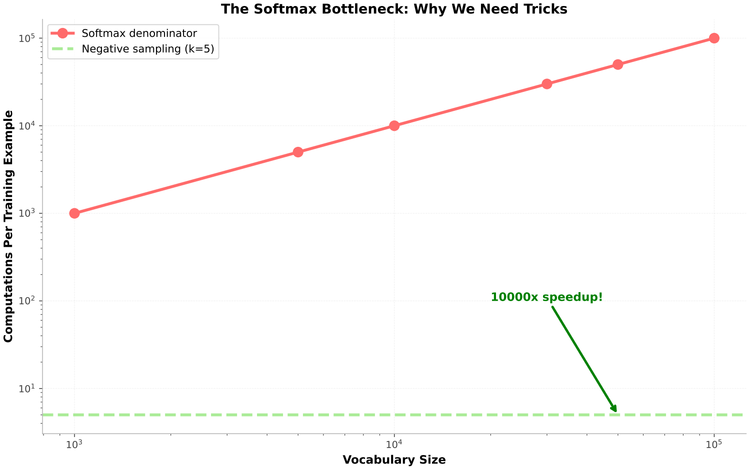
Step 4: Softmax over vocabulary

$$P(\text{``}sat''|\text{``}cat'') = \frac{\exp(0.85)}{\sum_{w \in V} \exp(score(w))} = 0.0023$$

Step 5: Compute loss

$$Loss = -\log(0.0023) = 6.07$$

The Computational Bottleneck



Key Insight: Computing softmax over 50K words is prohibitively expensive

Softmax denominator requires summing over entire vocabulary

Negative Sampling: The Trick That Makes It Practical

The Problem: Softmax over 50K words per training example

$$P(context|word) = \frac{\exp(score)}{\sum_{w=1}^{50000} \exp(score_w)}$$

Requires 50K exponentials per example!

The Solution: Negative Sampling

- 1 positive pair: ("cat", "sat") - actual context
- k negative pairs: ("cat", "xylophone"), ("cat", "democracy"), ... - random words
- Typical: $k = 5$ for small datasets, $k = 2 - 5$ for large

New Objective:

$$\log \sigma(v'_{sat} \cdot v_{cat}) + \sum_{i=1}^{k} \log \sigma(-v'_{w_i} \cdot v_{cat})$$

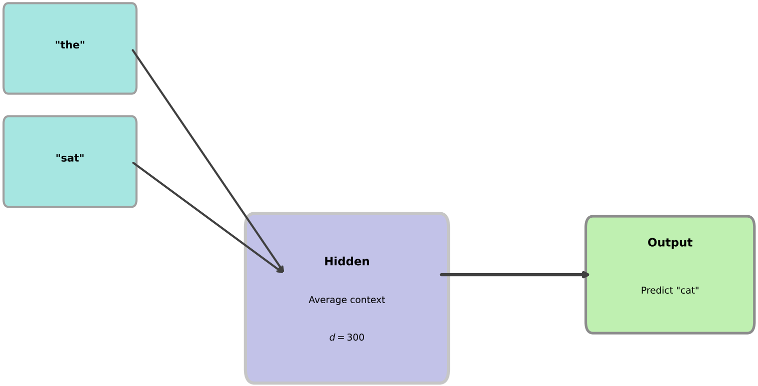
Only $k + 1$ computations instead of 50,000!

Example: Positive pair + 5 negatives = 6 computations vs 50,000
Speedup: **8,333x faster!**

Negative sampling approximates softmax - critical for practical training

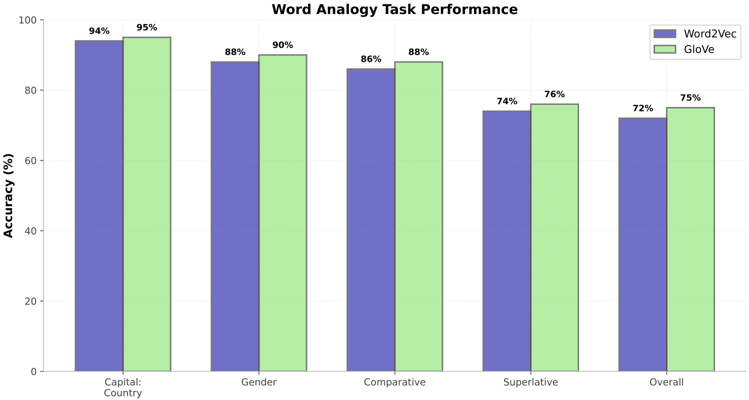
CBOW: The Reverse Approach

CBOW Architecture: Predict Word from Context



CBOW vs Skip-gram

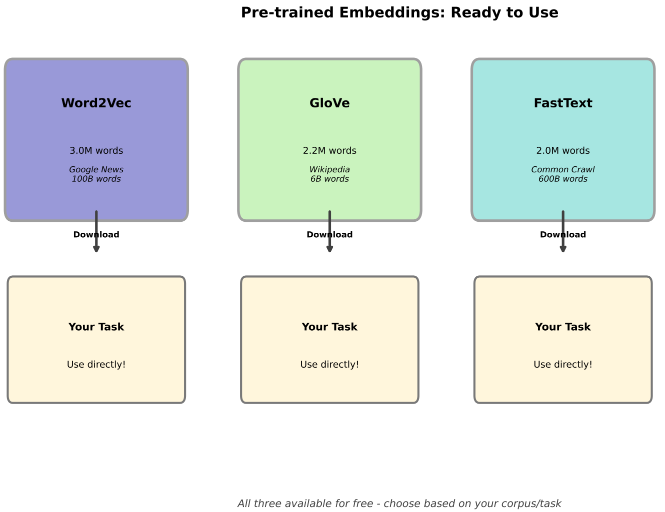
Evaluation: Word Analogies



Key Insight: Good embeddings solve analogies via vector arithmetic

king:queen :: man:woman achieved 72% accuracy (Word2Vec, 2013)

Pre-trained Embeddings



Key Insight: Use pre-trained embeddings as starting point

Key Takeaways

1. **Embeddings as dense vectors**
   Words $\rightarrow$ continuous vectors in $\mathbb{R}^d$ (typically $d = 300$)
2. **Skip-gram predicts context from word**
   Train by predicting surrounding words in large corpus
3. **Negative sampling enables efficient training**
   Approximate softmax with k negative samples (8000x speedup)
4. **Geometric semantics**
   Similarity = cosine, analogies = vector arithmetic
5. **Foundation for neural NLP**
   All neural models start with embedding layer

Embeddings revolutionized NLP in 2013 - still used everywhere today

Next: Visual Exploration

Lab Activities:

- Load pre-trained Word2Vec
- Visualize in 2D and 3D
- Perform word arithmetic
- Find most similar words
- Explore semantic clusters
- Compare Word2Vec vs GloVe
- Visualize training evolution

Visualizations You'll Create:

- 2D PCA projections
- Interactive 3D scatter plots
- Analogy arrows in 2D
- Similarity heatmaps
- Semantic cluster plots
- Training convergence

Goal: Build intuition through visualization

See embeddings come alive!

Understanding embeddings requires seeing them - lab is essential

Technical Appendix

Complete Mathematical Treatment

Appendix A1: Skip-gram Objective Function

Goal: Maximize probability of context words given center word

Objective:

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-c\leq j\leq c\\ j\neq 0}} \log P(w_{t+j}|w_t)$$

where:

- T: Total words in corpus
- c: Context window size (typically 5)
- w_t: Center word at position t
- w_{t+j}: Context word at offset j

Conditional Probability (Naive Softmax):

$$P(w_O|w_I) = \frac{\exp(v'_{w_O}\cdot v_{w_I})}{\sum_{w=1}^{|V|}\exp(v'_w\cdot v_{w_I})}$$

where v_{w_I} is input embedding, v'_{w_O} is output embedding

Problem: Denominator sums over entire vocabulary - $O(|V|)$ per example
For $|V| = 50K$, $T = 1B$ words, $c = 5$: 500 trillion softmax computations!

This objective is correct but computationally infeasible

Appendix A2: Negative Sampling Mathematics

Key Idea: Binary classification instead of multi-class

Reformulation:
Instead of predicting which word from vocabulary,
Predict: Is this word in context (yes/no)?

Negative Sampling Objective:

$$\log \sigma(v'_{w_O} \cdot v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)}[\log \sigma(-v'_{w_i} \cdot v_{w_I})]$$

where:

- $\sigma(x) = \frac{1}{1+\exp(-x)}$ (sigmoid)
- w_O: Actual context word (positive example)
- w_i: Sampled negative words
- $P_n(w)$: Noise distribution (typically $P(w)^{0.75}$)
- k: Number of negative samples (5-20)

Why $P(w)^{0.75}$?
Raises probability of rare words, lowers frequent words
More balanced negative sampling

This approximates softmax with k samples instead of $|V|$ computations

Appendix A3: Hierarchical Softmax Alternative

Different Approach: Binary tree instead of flat softmax

Key Idea:

- Arrange vocabulary in binary tree
- Prediction = path through tree
- Each node: Binary decision (left vs right)
- Depth: $\log_2 |V|$ decisions instead of $|V|$ computations

Complexity:

- Naive softmax: $O(|V|)$ per example
- Hierarchical softmax: $O(\log |V|)$ per example
- For $|V| = 50K$: $O(50000)$ vs $O(16)$ - 3000x speedup!

Trade-offs:

- Faster than naive softmax
- Slower than negative sampling
- Exact (no approximation)
- Tree construction matters

Usage: Less common than negative sampling

Appendix A4: Training via Gradient Descent

Optimization: Stochastic Gradient Descent (SGD)

Gradients for Skip-gram with Negative Sampling:
For positive pair (w_I, w_O):

$$\frac{\partial}{\partial v_{w_I}} = (1 - \sigma(v'_{w_O} \cdot v_{w_I}))v'_{w_O}$$

For negative pairs (w_I, w_i):

$$\frac{\partial}{\partial v_{w_I}} = -\sigma(-v'_{w_i} \cdot v_{w_I})v'_{w_i}$$

Update Rule:

$$v_{w_I}^{new} = v_{w_I}^{old} - \eta \cdot \frac{\partial L}{\partial v_{w_I}}$$

where η is learning rate (typically 0.025, decays to 0.0001)

Training Details:

- Mini-batch size: 100-1000 word pairs
- Epochs: 5-15 over corpus
- Learning rate decay: Linear
- Convergence: 1-3 days on CPU for 1B words

Appendix A5: Computational Complexity Analysis

Method	Per Example	Training Time	Quality
Naive Softmax	$O(\|V\| \cdot d)$	Weeks	Best
Hierarchical Softmax	$O(\log \|V\| \cdot d)$	Days	Good
Negative Sampling	$O(k \cdot d)$	Hours	Good

Typical: $|V| = 50K$, $d = 300$, $k = 5$, corpus=1B words

Memory Requirements:

- Embeddings: $|V| \times d \times 4$ bytes = 50K $\times$ 300 $\times$ 4 = 60MB
- Context matrix: Another 60MB
- Total: 120MB (fits in RAM easily)

Parallelization:

- Word2Vec easily parallelizable (independent windows)
- Multi-threading: Near-linear speedup
- GPU: 10-50x faster than CPU

Efficient algorithm + modern hardware = practical at scale

Appendix A6: GloVe - Global Vectors for Word Representation

Different Philosophy: Explicit matrix factorization

Key Idea:

- Word2Vec: Local context window (implicit matrix factorization)
- GloVe: Global co-occurrence statistics (explicit matrix factorization)

Co-occurrence Matrix X:

$$X_{ij} = \text{number of times word } j \text{ appears in context of word } i$$

Example snippet: Count how often "cat" and "dog" appear near each other across entire corpus

GloVe Objective:

$$\mathcal{L} = \sum_{i,j=1}^{|V|} f(X_{ij})(v_i^T v_j + b_i + b_j - \log X_{ij})^2$$

where $f(x)$ is weighting function (down-weight rare co-occurrences)

GloVe combines global statistics with local context

Appendix A7: GloVe Objective Function Breakdown

Goal: Dot product of vectors should match log co-occurrence

$$v_i^T v_j \approx \log X_{ij}$$

Weighted Least Squares:

$$\min \sum_{i,j=1}^{|V|} f(X_{ij})(v_i^T v_j + b_i + b_j - \log X_{ij})^2$$

Weighting Function:

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

Typical: $x_{max} = 100$, $\alpha = 0.75$

Why Weighting Matters:

- Very rare co-occurrences: Noisy, unreliable
- Very frequent: Dominate loss ("the the", "of the")
- Middle ground: Most informative

Weighting function is critical for GloVe performance

Appendix A8: Matrix Factorization Connection

Insight: Both Word2Vec and GloVe factorize co-occurrence matrix

Pointwise Mutual Information (PMI):

$$PMI(i,j) = \log \frac{P(i,j)}{P(i)P(j)} = \log \frac{X_{ij} \cdot |X|}{\sum_k X_{ik} \cdot \sum_k X_{kj}}$$

Measures how much more likely words co-occur than by chance

Connection:
Word2Vec (Skip-gram with negative sampling) implicitly factorizes shifted PMI matrix:

$$v_i^T v_j \approx PMI(i,j) - \log k$$

GloVe explicitly factorizes log co-occurrence matrix

Unifying View:
Both methods learn low-rank approximation of word-context statistics
Different loss functions, similar result

This explains why Word2Vec and GloVe produce similar embeddings

Appendix A9: GloVe Training Algorithm

Steps:

1. Build co-occurrence matrix X from corpus (count pairs within window)
2. Initialize word vectors v_i and biases b_i randomly
3. Optimize via AdaGrad:
   For each (i, j) pair with $X_{ij} > 0$:
   $$v_i^{new} = v_i - \eta \frac{\partial L}{\partial v_i}$$
   where gradient includes $f(X_{ij})$ weighting
4. Iterate until convergence (50-100 epochs typical)
5. Final embeddings: v_i (can optionally average with v_j)

Hyperparameters:

- Embedding dimension d: 100-300
- Context window: 10-15 (larger than Word2Vec's 5)
- x_{max}: 100
- α: 0.75
- Learning rate: 0.05 with AdaGrad

Training Time: Similar to Word2Vec (hours to days)

GloVe implementation simpler than Word2Vec (no neural network)

Appendix A10: Word2Vec vs GloVe - When to Use Each

Aspect	**Word2Vec**	**GloVe**
Method	Local context window	Global co-occurrence
Objective	Predict context	Factorize matrix
Complexity	$O(k \cdot d)$ per pair	$O(nnz)$ total
Training	Online (streaming)	Batch (requires X)
Memory	Low (embeddings only)	High (needs matrix)
Quality	Excellent	Excellent
Speed	Fast	Moderate
Rare words	Better (Skip-gram)	Moderate
Analogies	72%	75%
Best for	Large corpora, streaming	Small/medium corpora

Empirical Results (on same corpus):

- Performance: Nearly identical (GloVe 3-5% better on some tasks)
- Training time: Word2Vec faster (no matrix construction)
- Implementation: Word2Vec simpler (fewer hyperparameters)

Recommendation: Start with Word2Vec, try GloVe if you have time

Both are excellent - choice matters less than proper training

Appendix A11: Implementation Details and Best Practices

Corpus Preparation:

- Lowercase everything (or preserve case)
- Remove rare words (¡ 5 occurrences)
- Subsampling frequent words: $P(w_i) = 1 - \sqrt{t/f(w_i)}$ where $t = 10^{-5}$
- Helps balance frequent/rare words

Hyperparameter Choices:

Parameter	Small Corpus	Large Corpus
Embedding dim d	100-200	300-500
Window size c	3-5	5-10
Negative samples k	5-10	2-5
Min word count	5	10
Learning rate η	0.025	0.025
Epochs	10-20	5-10

Debugging Tips:

- Check: Loss should decrease steadily
- Test: Run analogies after each epoch
- Validate: Hold out 10% for validation

These details matter - bad hyperparameters give poor embeddings

Appendix A12: FastText - Character N-grams

Motivation: Word2Vec/GloVe ignore word morphology

FastText Innovation (Facebook AI, 2017):
Represent words as bags of character n-grams

Example: "playing" = ¡pl, pla, lay, ayi, yin, ing, ng¿

Embedding:

$$v_{playing} = \sum_{g \in ngrams(``playing'')} v_g$$

Sum of character n-gram embeddings

Advantages:

- Handle OOV words (unseen in training)
- Capture morphology ("play" in "playing", "player")
- Better for morphologically rich languages
- Small vocabulary (can't memorize all words)

Trade-offs:

- Handles rare/unseen words
- More parameters (n-grams)
- Morphological awareness

Appendix A13: ELMo - Deep Contextualized Embeddings

Limitation of Word2Vec/GloVe: One vector per word (no context)
"bank" always has same embedding (averages river and money meanings)

ELMo Solution (2018):

- Embeddings from Language Model (ELMo)
- BiLSTM reads sentence
- Each word gets different embedding depending on context

Example:

- "The bank of the river" $\rightarrow v_{bank}^{river}$
- "Money in the bank" $\rightarrow v_{bank}^{money}$
- $v_{bank}^{river} \neq v_{bank}^{money}$

Connection to Week 6:
ELMo $\rightarrow$ BERT $\rightarrow$ GPT progression
All use context to modify embeddings

Note: Static embeddings (Word2Vec/GloVe) still useful for many tasks

ELMo bridged static embeddings to contextual (BERT) - important milestone

Appendix A14: Evaluation Metrics

Intrinsic Evaluation (embedding quality directly):

- **Word Similarity**: Correlation with human judgments (WordSim-353, SimLex-999)
- **Word Analogies**: Accuracy on a:b :: c:d tasks (Google analogy dataset)
- **Clustering**: Do semantic categories cluster?

Extrinsic Evaluation (downstream task performance):

- Use embeddings in actual NLP task
- Sentiment classification accuracy
- Named entity recognition F1
- Question answering performance

Trade-offs:

- Intrinsic: Fast, but doesn't guarantee downstream success
- Extrinsic: Slow, but measures real usefulness

Best Practice: Use both - intrinsic for development, extrinsic for final validation

Good intrinsic scores usually (but not always) lead to good extrinsic performance

Appendix A15: From Word2Vec to Transformers

The Evolution (2013-2024):

Year	Model	Innovation
2013	Word2Vec	Static distributed representations
2014	GloVe	Matrix factorization perspective
2017	FastText	Subword embeddings
2018	ELMo	Contextualized (BiLSTM)
2018	BERT	Transformer encoder (Week 6)
2018	GPT	Transformer decoder (Week 6)
2024	GPT-4/Claude	1T+ parameters, multimodal

What Stayed from Word2Vec:

- Embedding layer (first layer of all neural models)
- Distributional hypothesis
- Pre-training on large corpora
- Vector arithmetic intuition

What Changed:

- Static $\rightarrow$ Contextualized
- Single vector $\rightarrow$ Different vectors per context
- Window $\rightarrow$ Full sentence attention