**Lecture 2 : Word Vectors, Word Senses, and Neural Network Classifier**

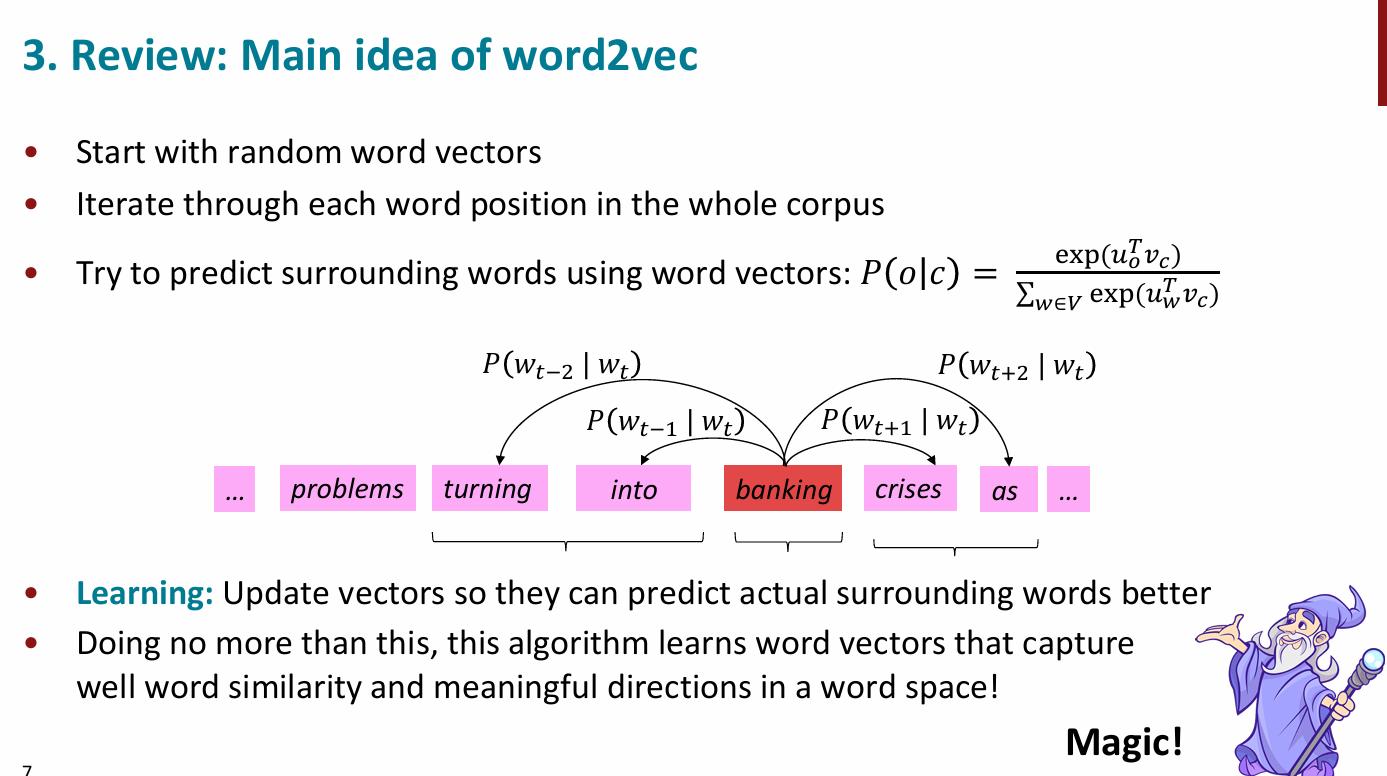
Stochastic Gradient Descent (SGD) Page 1

Review: Main idea of word2vec Page 1-3

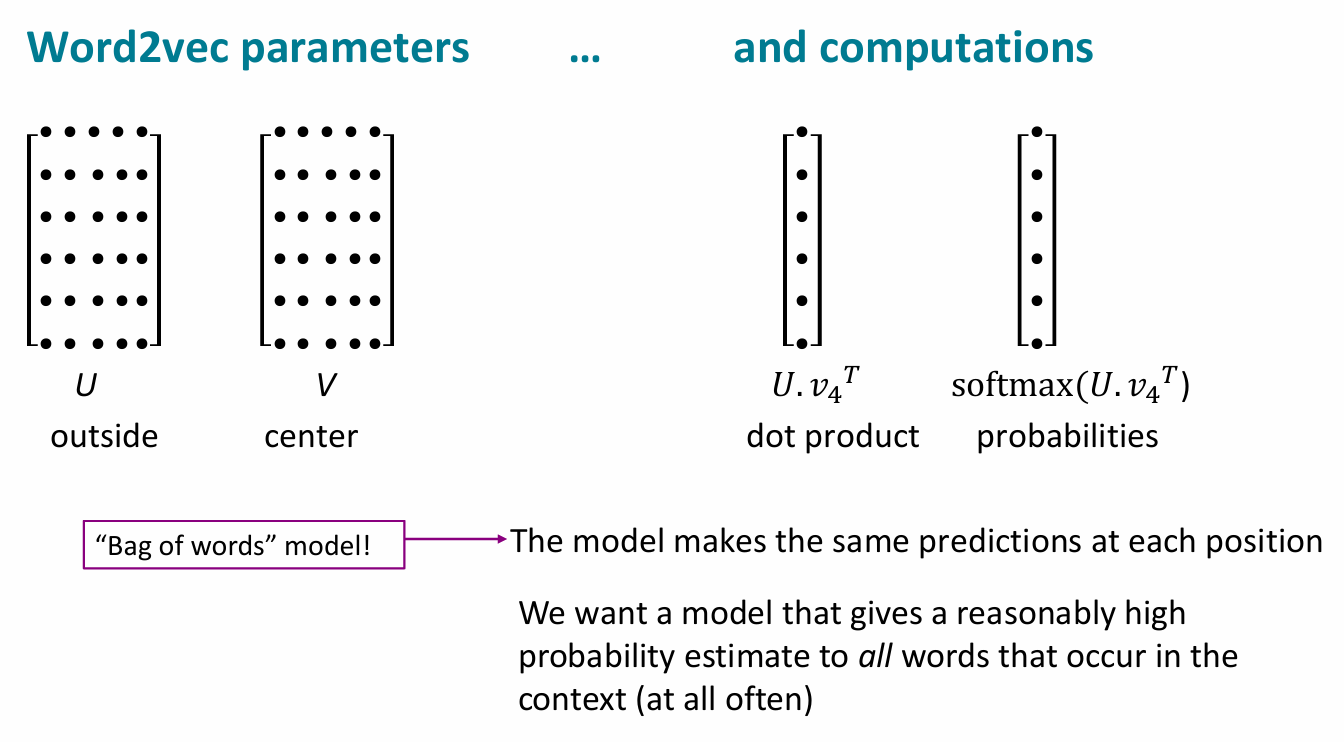
Word2vec algorithm family: More details Page 4-7

Skipgram , Skipgram Negative Sampling

**Stochastic Gradient Descent (SGD),** especially in mini-batches, is faster but also more resource-intensive. While it introduces some noise into the training process, this noise can actually help neural networks perform better. The randomness helps the model escape local minima and explores the solution space more effectively. So, despite the added noise, SGD not only speeds up training significantly but also often leads to better optimization results for neural networks.

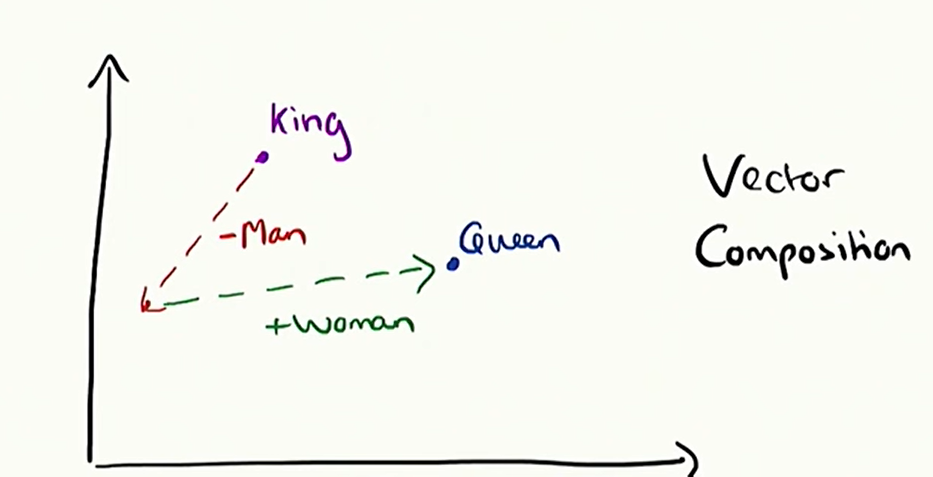


* In Word2Vec, **we start** by assigning each word a vector filled with small **random numbers**. This random initialization is essential—if all vectors are set to zero, the model can't learn because everything looks the same, creating **false symmetries**. **Random** **values break this symmetry** and allow the model **to learn meaningful differences** between words.
* We go through each word in the corpus and, using our current word vectors, try to predict the surrounding context words. This process defines an **objective function**.
* By measuring the prediction errors, we compute gradients and update the vectors to improve their predictions. Remarkably, simply doing this allows the model **to learn word vectors that capture meaningful relationships and semantic similarities between words**.



In Word2Vec, the **only model parameters are the word vectors:** one set for center words and another for outside (context) words, which are treated separately. For each center word, **we compute the dot product with all possible outside** words to get a **probability distribution over likely context words**. We then compare this to the actual context word **to measure the error**.

This model is considered a **bag-of-words** approach—**it ignores word order and sentence structure.** It treats words to the left and right of the center word the same and *focuses only on which words tend to appear nearby*.



One powerful feature of Word2Vec is that it captures **semantic components** of meaning in vector form, allowing us to **combine or manipulate them in meaningful ways**—for example, using vector arithmetic like:

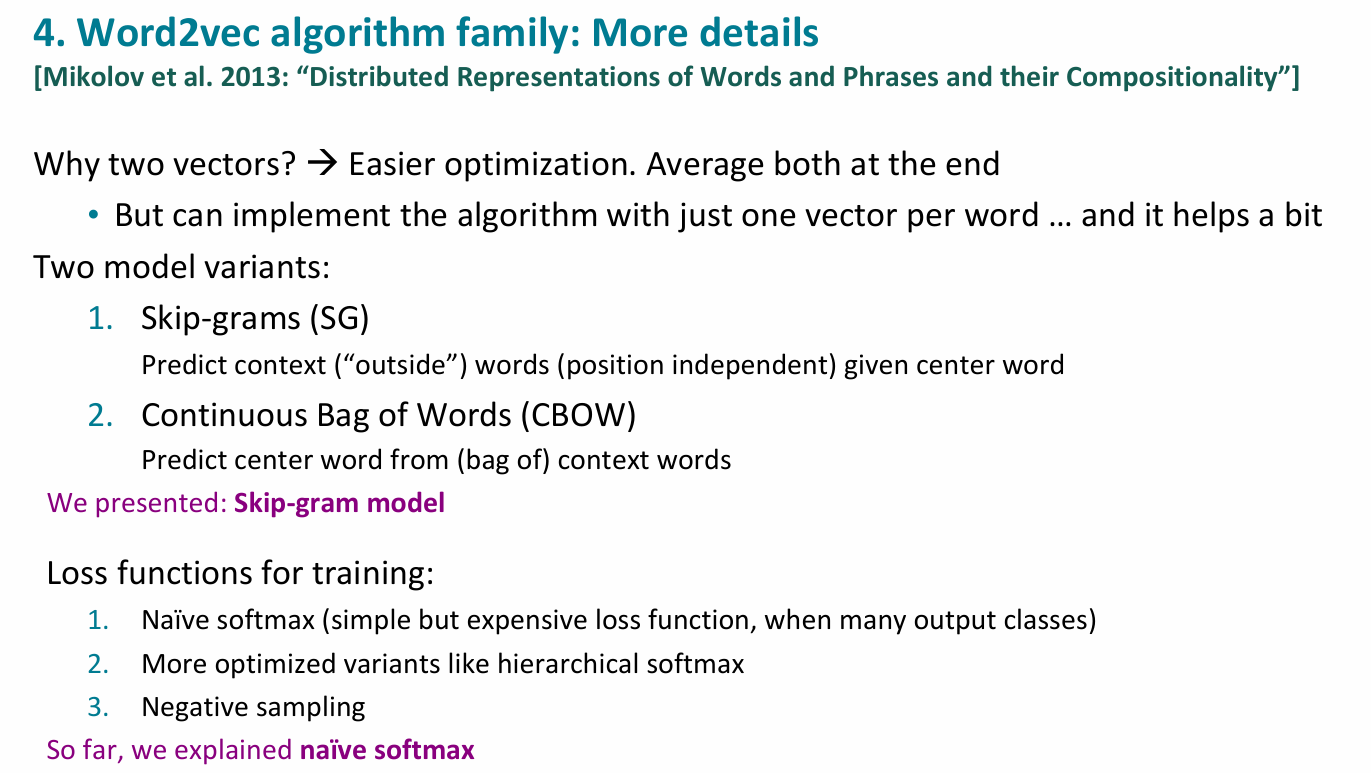
"king"−"man"+"woman"≈"queen"

**Q: In Word2Vec, each word has two vectors (U and V). How do we combine them into one?**

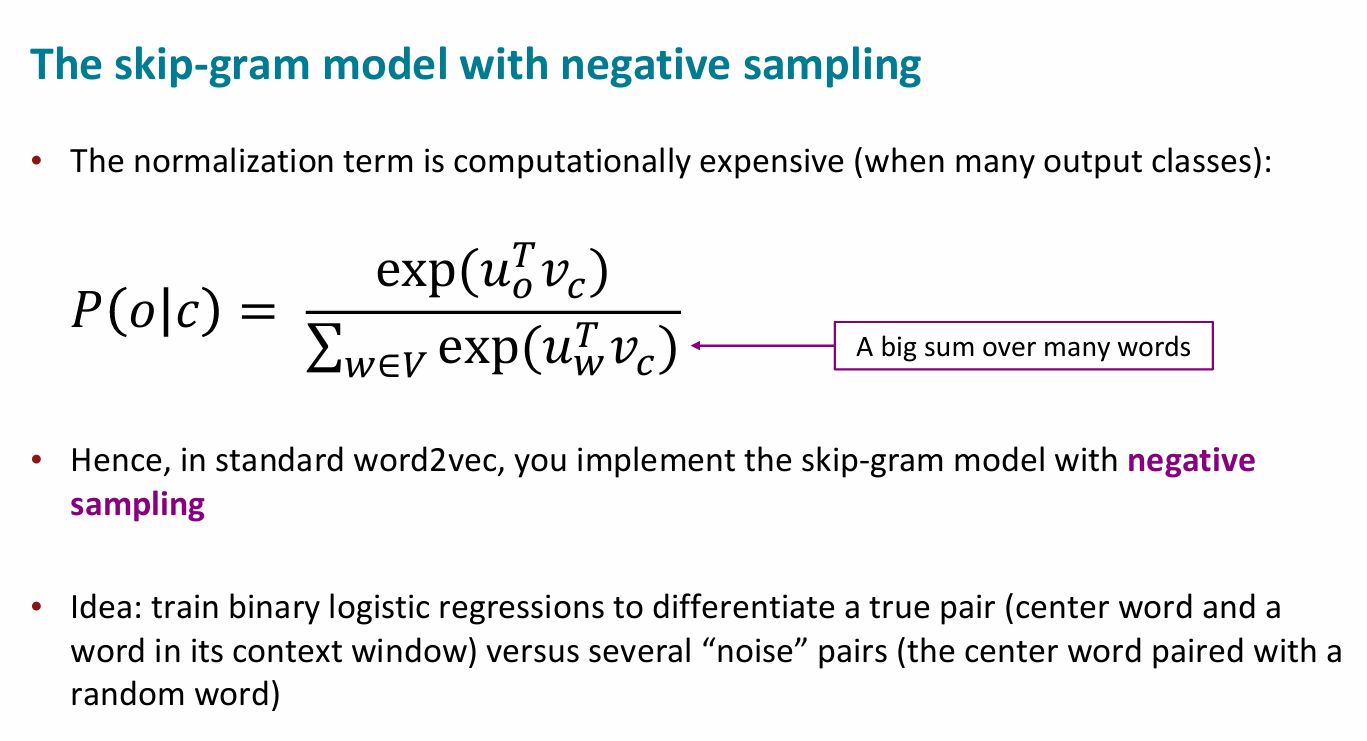
A common practice is to simply **average the two vectors**. This works well because during training, each word appears both as a center word and a context word. For example, in the sentence “the octopus has legs,” the word “octopus” might be the center with “legs” in the context, and later “legs” might be the center with “octopus” in the context. As a result, the two vectors for a word end up quite similar, so averaging them is a practical and effective solution.

**Q: Why does Word2Vec use two separate sets of vectors for each word?**

Using two sets of vectors—one for center words and one for context words—**simplifies the math**. If the same vector were used for both, you'd end up with quadratic **terms (*like x2) during normalization*** when a word appears as both center and context (e.g., “octopus” with itself), which ***complicates the gradient calculations***. Keeping the vectors separate avoids this issue and makes **optimization cleaner**. While using a single vector set can work slightly better if handled carefully, in practice, it's common to train them separately and then **average them at the end**.

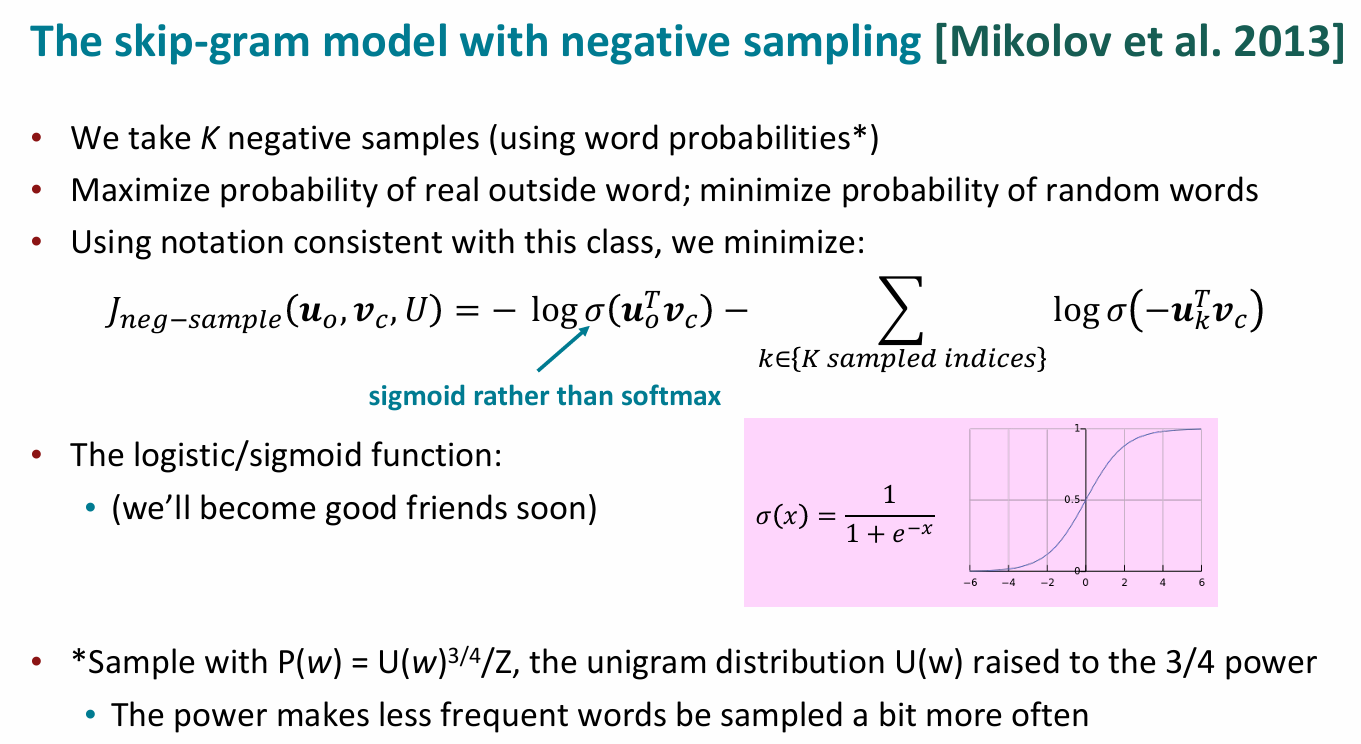


Naive softmax, just consider every possible choice of a context word and just run the entire math.



In the standard softmax setup we used earlier, the denominator requires summing over **every word in the vocabulary**—which could mean calculating dot products and exponentials for **hundreds of thousands of words**. That’s computationally expensive.

**Negative sampling** offers a shortcut: instead of computing the full softmax, we train a set of **small logistic regression tasks**. For each real (center, context) word pair, the model is trained to score it highly, while **randomly sampled “negative” words** are scored low. This is the core idea behind **Skip-gram with Negative Sampling (SGNS)**—efficiently teaching the model which words *should* and *shouldn't* appear together.



✅ **S kip-Gram with Negative Sampling (SGNS) – Explained Simply**

In the **standard softmax**, we had to compute probabilities over all words in the vocabulary, which is very expensive (e.g., 400,000 words).  
To make this faster, **Negative Sampling** simplifies the task.

**🧠 What’s the Idea?**

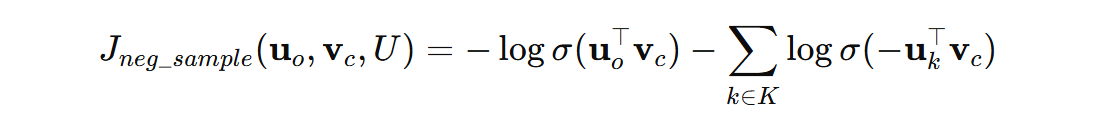
* For each **real word pair** (center word and actual context word), we:
  + **maximize** the probability that they appear together.
* For a few **randomly chosen "negative" words**, we:
  + **minimize** the probability that they appear with the center word.

This creates a **binary classification problem**:

* Is this word a true context word (positive sample)?
* Or a random one (negative sample)?

**🧮 Mathematical Formulation**

We minimize the following loss:

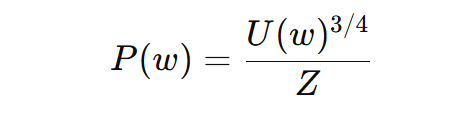


* : **Sigmoid function**
* : vector for the true context word
* : vectors for **K negative samples**
* : vector for the center word

✔ We want the dot product with the **true context word** to be **high** (→ sigmoid ≈ 1)  
✖ We want the dot product with **random words** to be **low** (→ sigmoid ≈ 0)

**⚖️ Sampling Negative Words**

* We don't choose negative words uniformly.
* We use the **unigram distribution** raised to the **3/4 power**:



* This means **frequent words (like “the”) are downsampled a bit**, and **less frequent words are sampled more often**.
* This creates a balance between completely uniform and frequency-based sampling.

**🔍 Why the Sigmoid Instead of Softmax?**

* **Softmax** needs computation over the whole vocabulary → expensive.
* **Sigmoid** is simpler: it gives a probability for **each pair independently**.

**💡 Why Is This Efficient?**

* **Sigmoid function** works on **individual word pairs**.
* You only compute:
  + One **positive pair**:
  + k **negative pairs**:
* So each training step involves **1 + k** dot products and sigmoid operations—not millions.

When training Word2Vec using **Stochastic Gradient Descent (SGD)** with **Negative Sampling**, **only a few word vectors are updated at each step**, not the entire embedding matrix.

Because only a few word vectors are updated, we must **optimize how we update** them.

**🗣️ Analogy:**

Imagine training the model to recognize "real friends" (true word pairs) and "random strangers" (negative samples). You want it to confidently pick out friends from the crowd, without needing to scan every single stranger.

**5. Why not capture co-occurrence counts directly? (Slides 15 to 23 for details)**

## 🧠 Why Not Just Count Word Co-Occurrences?

At first glance, it seems intuitive to build word vectors by simply **counting how often words appear near each other**. For example, count how often "swim" or "fish" occurs next to "octopus." This gives you a **co-occurrence matrix**.

### 📊 Example:

From a small corpus like:

* "I like deep learning"
* "I like NLP"
* "I enjoy flying"

You can build a co-occurrence matrix where each cell shows how often a word appears near another. This is simple and effective **in theory**.

## ❌ But There Are Issues...

### 1. ****Huge Size****

* If you have a vocabulary of 400,000 words, your matrix is **400,000 × 400,000**.
* That's **160 billion entries** — impractical to store or compute.

### 2. ****Sparse and Inefficient****

* Most entries are **zeros** (words rarely co-occur).
* Models that use this are **slow and memory-heavy**.

## ✅ Dimensionality Reduction as a Fix

To handle this, we apply **Singular Value Decomposition (SVD)**:

* It breaks the large matrix into smaller matrices capturing the **most important patterns**.
* We **keep only the top k dimensions** (e.g., 100–300), reducing size and noise.
* This idea was used in **Latent Semantic Analysis (LSA)**.

## ⚙️ COALS: Improving Raw Count Matrices

Doug Rohde (2005) proposed **COALS**, which refined this further:

* Apply **log scaling** to frequencies (to reduce dominance of common words like “the”).
* Use **Pearson correlation** instead of raw counts.
* Weight **nearby words more heavily** than distant ones (ramped windows).
* Result: cleaner, more useful semantic vectors.

He even found **linear meaning components**—for example, vector differences could represent relationships like:

* "teach" - "teacher" ≈ "learn" - "student"

## 💡 From COALS to GloVe: Generalizing the Idea

Inspired by this, Chris Manning and Jeffrey Pennington developed **GloVe** (Global Vectors for Word Representation).

### GloVe's Key Insight:

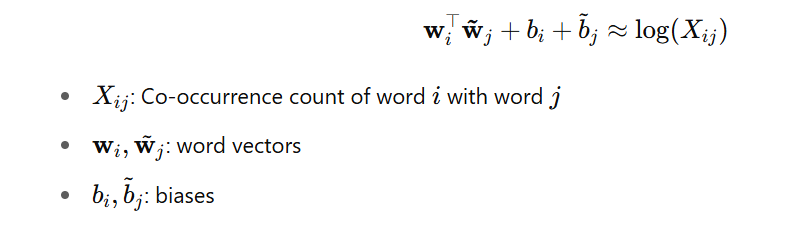
* **Ratios of co-occurrence probabilities** (instead of just counts) encode **semantic meaning**.

For example:

The ratio is high → solid is more associated with ice than steam.

**✏️ GloVe Model:**

Instead of using neural nets like Word2Vec, GloVe uses a **log-bilinear model**:



This gives **linear meaning components** like Word2Vec does, but from a **matrix factorization approach**.

## 🎯 Why GloVe Is Powerful

* Combines **global statistics** (co-occurrence counts) with the **semantic power** of word vectors.
* Fast to train and scales to large corpora.
* Produces high-quality embeddings that support analogies and similarity tasks.

## 🔑 Summary

| **Method** | **Key Idea** | **Problem** | **Solution** |
| --- | --- | --- | --- |
| Raw co-occurrence | Count nearby words | Matrix too big & sparse | SVD or COALS |
| Word2Vec | Predict context via SGD | Ignores full co-occurrence stats | Efficient training with SGNS |
| GloVe | Use co-occurrence ratios | Combine best of both worlds | Log-bilinear regression model |

**6. How to evaluate word vectors?**

## 🧠 Why Evaluate Word Vectors?

In NLP, we need to **evaluate** models and components to know **what’s working** and **how to improve**. For word vectors (like those from Word2Vec or GloVe), we want to test whether they truly capture **semantic** and **syntactic** meaning.

## 🔍 Two Types of Evaluation

### 1. ****Intrinsic Evaluation****

* Tests **word vectors alone**, using small, focused tasks.
* **Fast and interpretable**, but may not always reflect real-world usefulness.

#### 🔧 Common Intrinsic Methods:

**A. Word Analogies (Semantic + Syntactic)**  
Example:

man : woman :: king : ?  
The model should predict **queen** based on vector math:

king – man + woman ≈ Queen

* Measures how well vectors **capture relationships**.

**B. Word Similarity (Correlation with Humans)**

Compare cosine similarity between word pairs with **human judgments**.  
Example human scores:

| **Word 1** | **Word 2** | **Human Similarity** |
| --- | --- | --- |
| tiger | cat | 7.35 |
| plane | car | 5.77 |
| stock | jaguar | 0.92 |

The model’s scores are then **correlated (e.g., Spearman rank)** with human ratings.

### 2. ****Extrinsic Evaluation****

* Tests word vectors by using them in a **real NLP task**.
* Measures whether they help improve **downstream system performance**.

#### 🔧 Example Extrinsic Task:

**Named Entity Recognition (NER)**

* Identify named entities like people, places, organizations in text.  
  Example:

"Chris Manning lives in Palo Alto"  
Should detect:

* **Chris Manning** → Person
* **Palo Alto** → Location

By **adding word vectors as features**, NER systems typically perform better.

## 📊 Evaluation Results (Summary from Tables)

| **Model** | **Intrinsic (WordSim)** | | **Extrinsic (NER F1)** |
| --- | --- | --- | --- |
| SVD | Lower performance | ~85.7 | |
| CBOW | Moderate | ~88.2 | |
| Skip-Gram | Better | ~88.3 | |
| GloVe | Best overall | **88.3–93.2** | |

* **GloVe vectors** consistently outperformed others on both intrinsic and extrinsic evaluations.
* Shows **meaningful linear relationships** and boosts real-world task accuracy.

## 🧪 Extra Insights from Model Analysis

* **Vector Size**: Improvements plateau around 200–300 dimensions.
* **Context Window**:
  + **Small windows** (e.g., ±2) → better for **syntax**.
  + **Large windows** → better for **semantics**.
* **Corpus Size**:
  + Bigger corpora help, but sometimes smaller curated datasets (like Wikipedia) do better on **semantic tasks** due to higher-quality data.

## ⚖️ Summary

| **Intrinsic Evaluation** | **Extrinsic Evaluation** |
| --- | --- |
| Tests isolated word vectors | Tests in real NLP pipelines |
| Fast & interpretable | Slower but more realistic |
| Example: analogies, similarity | Example: NER, QA, summarization |
| May not reflect final task performance | Directly shows practical value |

✅ **Use both**: Intrinsic for development/debugging, Extrinsic for final performance validation.

**7. Word senses and word sense ambiguity**