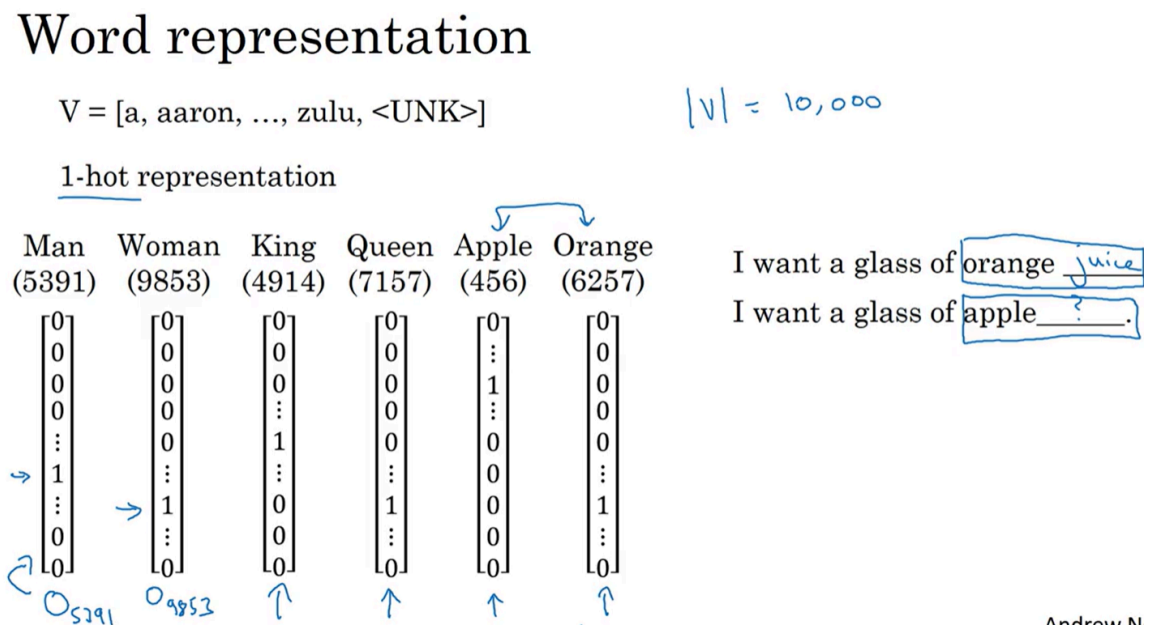
WEEK 2: NLP & Word Embeddings

Word Representation

Word Representation: One-Hot vs Embedding Notation

- **Vocabulary & One-Hot Encodings:**



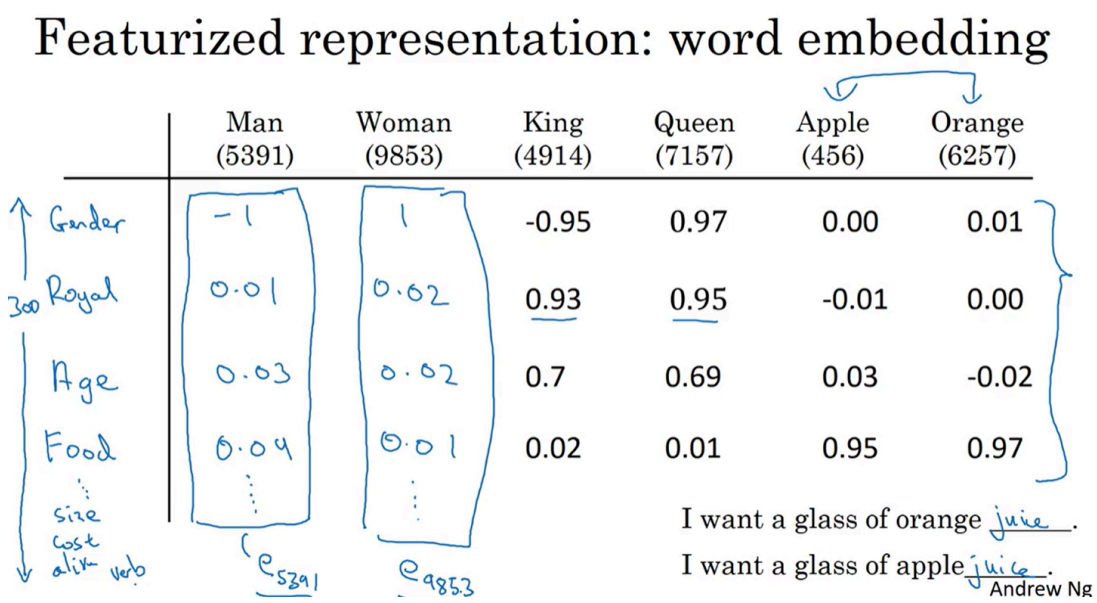
  - Suppose the vocabulary contains V words (e.g., V=10,000).
  - If “man” is word #5391, represent as **one-hot vector** O_{5391}: only the 5391^{th} entry is 1, all others are 0.
  - “woman” (word #9853): O_{9853}, one at position 9853, zeros elsewhere.
  - **Note:** For any two distinct words, their one-hot vectors are orthogonal: $O_i^\top O_j = 0,\ \forall\, i \neq j$

    and their Euclidean distance is always the same (does not capture semantic similarity).

- **Limitations:**
  - All words are equally unrelated. For example, “apple” and “orange” are as far apart as “apple” and “king”.

- Model cannot generalize: learning “orange juice” ≠ learning “apple juice”.

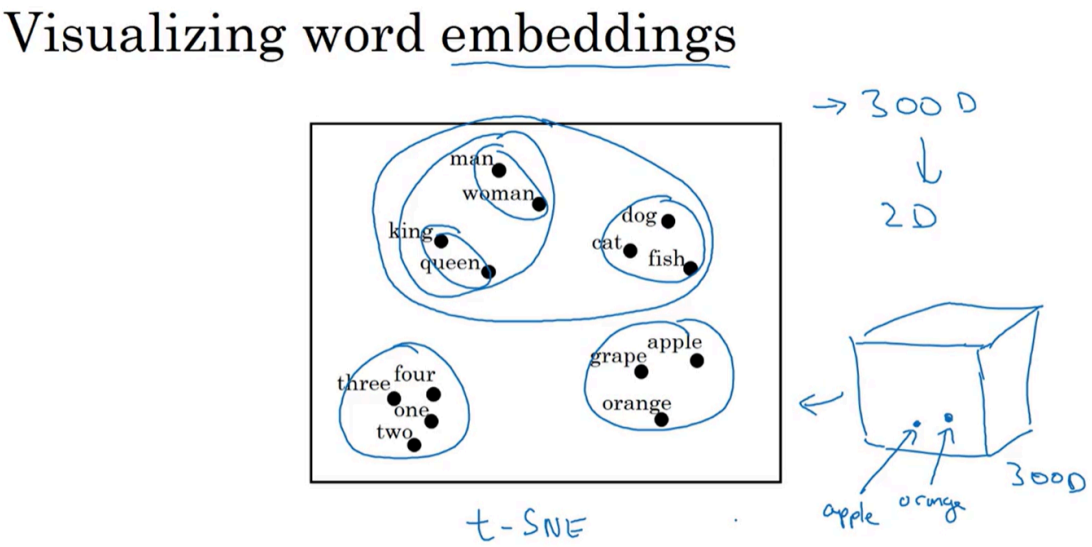
- **Featurized (Embedding) Representation:**



  - Instead, learn a real-valued vector $e_i \in \mathbb{R}^n$(e.g., n=300): $e_{5391} =$ embedding of ’man’
    $e_{9853} =$ embedding of ’woman’
  - Features can correspond (sometimes) to:
    - *gender* (−1 for male, +1 for female)—e.g., “man” = −1, “woman” = +1
    - *royalty* (e.g., “king”, “queen” >0), *is food* (“apple”, “orange” = 1), etc.
  - In reality, these 300 dimensions are learned automatically (not directly interpretable).

- **Advantages:**
  - Similar words (fruits, royals, gender) are **close** in embedding space:
    $\|e_{\text{apple}} - e_{\text{orange}}\|^2 \ll \|e_{\text{apple}} - e_{\text{king}}\|^2$
  - **Inner product** or **cosine similarity** between embeddings captures “relatedness”.

- **Visualization:**



- Use t-SNE algorithm (Laurens van der Maaten & Geoff Hinton) to reduce 300D embeddings to 2D for intuitive plotting.
- Clustering: “man”/“woman”, “king”/“queen”, “apple”/“orange”, numbers, animals, etc. appear close in reduced space.

- **Notation Table:**

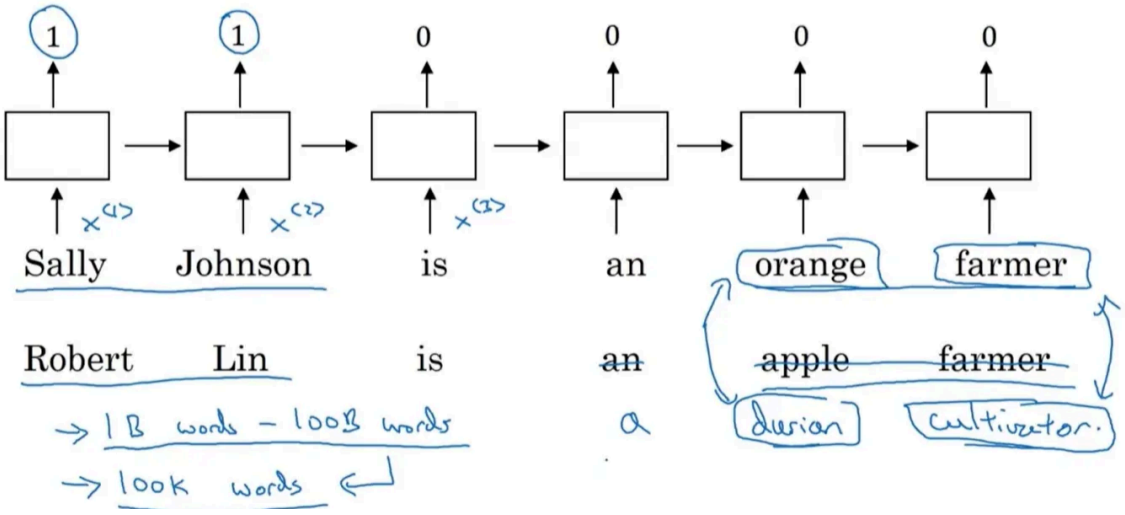
Symbol	Meaning
O_i	One-hot vector for word index i
e_i	Embedding vector for word index i
V	Vocabulary size
n	Embedding dimension (commonly 300)

- **Crucial Point:**

  Embeddings grant models the power to generalize, relate, and cluster words by real-world similarity, which is impossible with one-hot encodings.

Using Word Embeddings

Transfer Learning Paradigm



Example: NER Problem

- **Problem Context**: Named Entity Recognition (NER) task
  - Example: "Sally Johnson is an orange farmer" → Identify "Sally Johnson" as person (output = 1)
  - Traditional approach: One-hot vectors $x^{(1)}, x^{(2)}, \ldots$ for each word

💡 What is Transfer Learning?

Transfer learning is a powerful technique in machine learning where you take a model trained on one task (often with a large dataset) and adapt it for another, usually related, task with less data. The core idea is to reuse learned knowledge (such as useful feature representations) so you can train better models, faster, especially when labeled data is scarce.

Transfer learning is widely used in fields like computer vision (using models like ResNet or EfficientNet trained on ImageNet) and natural language processing (using models like BERT or GPT trained on billions of words).

Why use transfer learning?

- **Save time and resources:** You don't have to train a model from scratch.
- **Boost performance:** Leverages the general features learned from big datasets, which helps on smaller or niche tasks.
- **Often outperforms training from scratch** when your target task has limited labeled data.

Steps of Transfer Learning (in NLP/word embeddings or deep learning)

1. **Start with a pre-trained model:**
    - Example: Pre-trained word embeddings (like GloVe, Word2Vec) or neural network weights trained on large datasets.
2. **Transfer to your new task:**
    - Use the learned features (the model weights or embeddings) as the starting point for your own task.
3. **(Optional) Fine-tune on your target data:**
    - If you have enough data, you adjust ("fine-tune") the model weights a bit more so the model adapts to the special characteristics of your dataset.
    - If not much data, you can "freeze" most weights and only train the last layers (so you don't overfit).

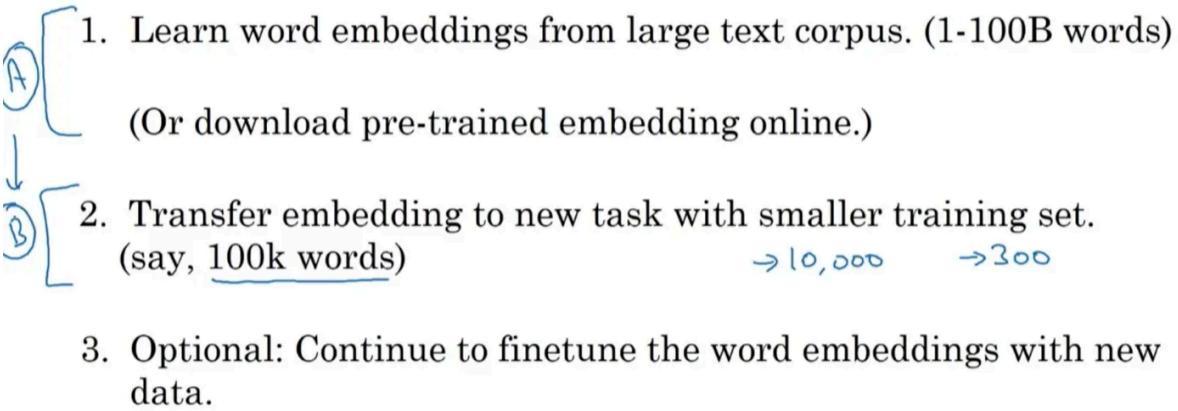
Generalization Power of Embeddings

- **Training Example**: "Sally Johnson is an orange farmer"
- **Test Generalization**: "Robert Lin is an apple farmer"
  - **Key Insight**: Since $\text{sim}(\mathbf{e}_{\text{orange}}, \mathbf{e}_{\text{apple}}) > \text{threshold}$, (sim means similarity) model generalizes that "Robert Lin" is also a person
- **Extreme Generalization**: "Robert Lin is a durian cultivator"
  - **durian**: Rare fruit (popular in Singapore)
  - **cultivator**: Someone who cultivates ≈ farmer
  - Model generalizes even with unseen words due to embedding similarities

Data Scale Advantage

- **Unlabeled Text Corpus**: 1 billion → 100 billion words
  - Source: Internet text (essentially free)
  - Discovers semantic relationships: $\mathbf{e}_{\text{orange}} \approx \mathbf{e}_{\text{durian}}$(both fruits)
  - Discovers functional relationships: $\mathbf{e}_{\text{farmer}} \approx \mathbf{e}_{\text{cultivator}}$ (both occupations)
- **Labeled NER Dataset**: ~100,000 words (much smaller)

Transfer Learning Steps



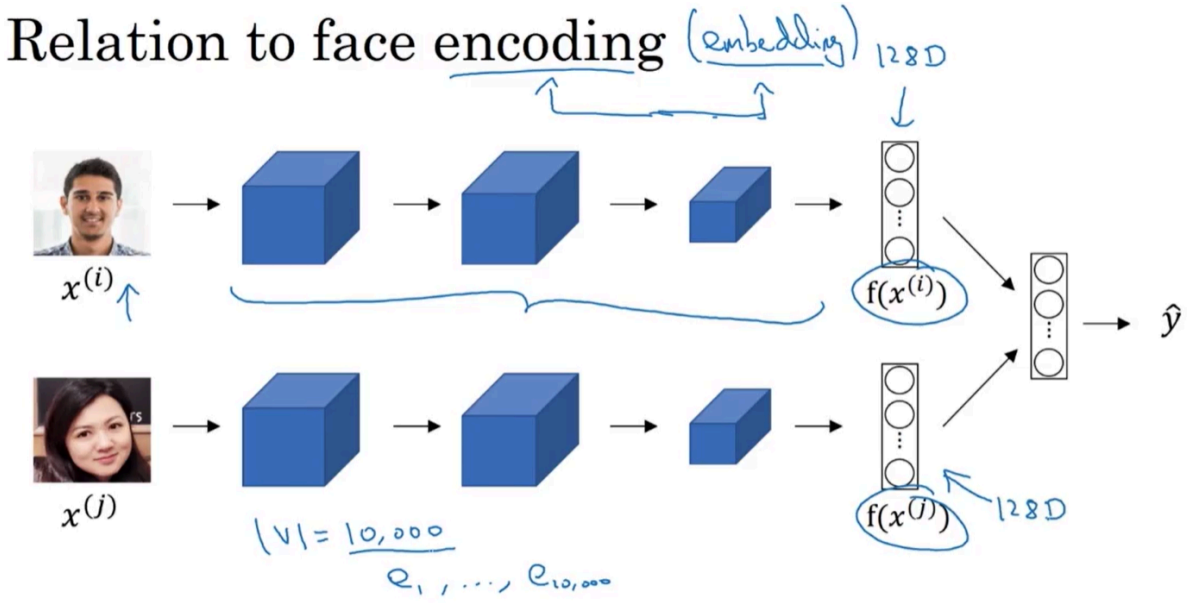
1. **Step 1**: Learn word embeddings from large text corpus
   - Option: Download pre-trained embeddings (permissive licenses available)

2. **Step 2**: Transfer embeddings to new task with smaller labeled dataset
    - Use d=300 dimensional dense vectors instead of $|V|$=10,000 dimensional one-hot
3. **Step 3** (Optional): Fine-tune embeddings on new task
    - **Condition**: Only if labeled dataset for task 2 is sufficiently large
    - **If small dataset**: Skip fine-tuning to avoid overfitting

Task Effectiveness Analysis

- **High Effectiveness**: Small training data scenarios
    - Named Entity Recognition
    - Text Summarization
    - Co-reference Resolution
    - Parsing
- **Lower Effectiveness**: Large training data scenarios
    - Language Modeling
    - Machine Translation
- **Transfer Learning Principle**: Most useful when $|\text{Data}_A| \gg |\text{Data}_B|$

Embeddings vs. Encodings Comparison



- **Face Recognition Encodings**:
  - Network: $f(x^{(i)}) \rightarrow \mathbb{R}^{128}$ for any face image
  - Capability: Handle unlimited, unseen face images
- **Word Embeddings**:
  - Fixed vocabulary: $\{e_1, e_2, ..., e_{10000}\}$
  - Limitation: Unknown words → "UNK" token
- **Terminology**: "Encoding" and "embedding" used interchangeably
  - Difference is algorithmic, not terminological

Architecture Note

- **Shown**: Unidirectional RNN for simplicity
- **Recommended**: Bidirectional RNN for actual NER implementation

Key Mathematical Insight

$$\text{Semantic Similarity} \propto \cos(\mathbf{e}_i, \mathbf{e}_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{||\mathbf{e}_i|| \cdot ||\mathbf{e}_j||}$$

Where embeddings enable generalization: $\cos(\mathbf{e}_{\text{orange}}, \mathbf{e}_{\text{apple}}) > \cos(\mathbf{e}_{\text{orange}}, \mathbf{e}_{\text{king}})$

Properties of Word Embeddings

Introduction to Word Embeddings

- Word embeddings represent words as dense vectors in a continuous vector space (e.g., 300 dimensions).
- These vectors capture semantic and syntactic properties of words, enabling NLP tasks like analogy reasoning.
- Contrast with one-hot vectors: one-hot vectors are orthogonal and fail to capture similarity or semantic relationships.

Vector Representation and Notation

- Vocabulary size: V (e.g., 10,000 words)
- Embedding dimension: n (e.g., 300 dimensions)

- Embedding vector for word indexed i:$e_i \in \mathbb{R}^n$
- One-hot vector for word i: $O_i \in \{0,1\}^V$ with 1 at position i and 0 elsewhere

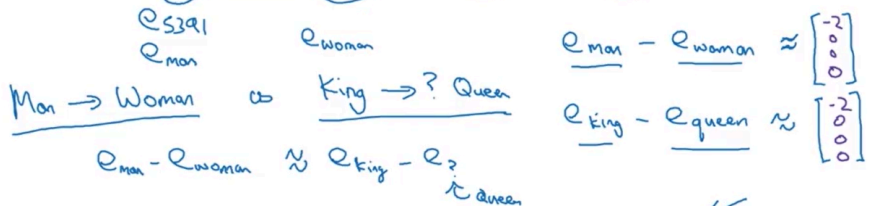
Key Properties of Word Embeddings

- **Semantic Similarity:** Vectors for similar words are close in embedding space, e.g., $\|e_{\text{apple}} - e_{\text{orange}}\|_2 \ll \|e_{\text{apple}} - e_{\text{king}}\|_2$
- **Relation Encoding:** Vector differences encode relations such as gender or royalty.
  - Example: $e_{\text{man}} - e_{\text{woman}} \approx e_{\text{king}} - e_{\text{queen}}$
  - This captures the concept of gender as a direction in embedding space.

Mathematical Formulation of Analogy Task

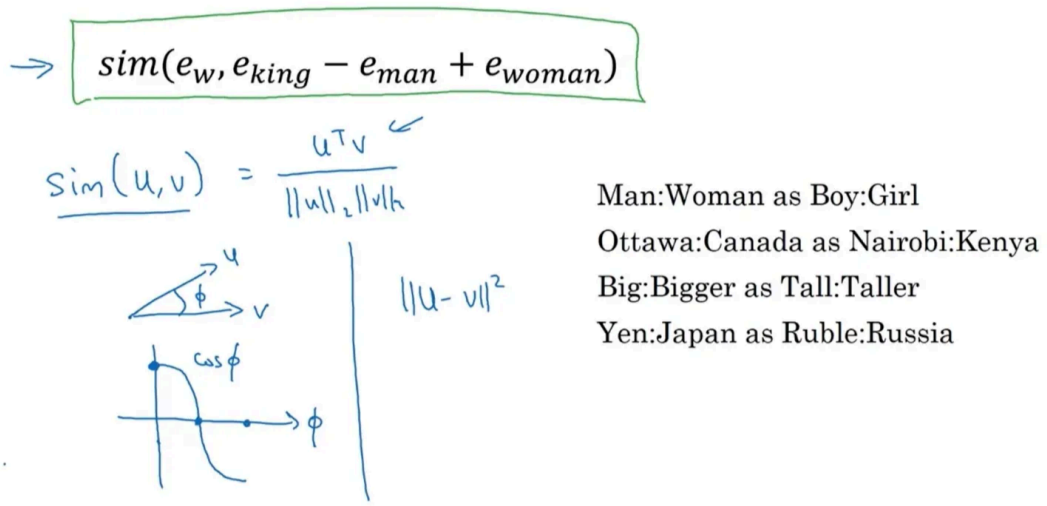
Analogies

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	−1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97



- Given analogy $\text{man} : \text{woman} :: \text{king} :?$, goal is to find word w such that: $e_{\text{king}} - e_{\text{man}} + e_{\text{woman}} \approx e_w$
- Find $w^* = \arg\max_w \ \text{sim}\big(e_w, e_{\text{king}} - e_{\text{man}} + e_{\text{woman}}\big)$
- $\text{sim}(u, v)$ denotes similarity measure, commonly **cosine similarity**.

Cosine Similarity



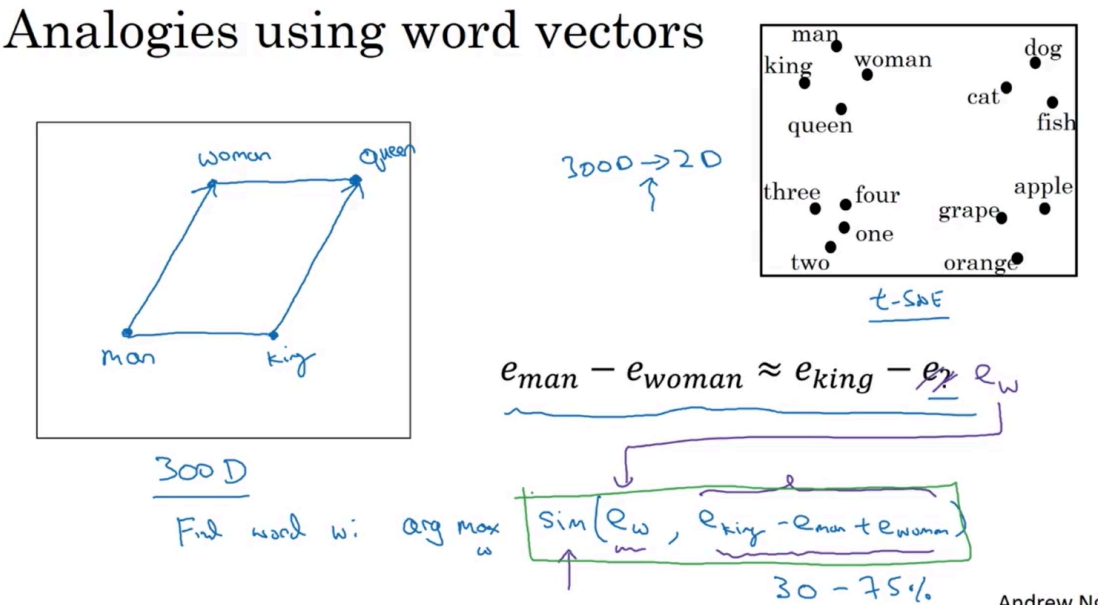
- Defined as:$\text{cosine similarity}(u, v) = \frac{u^\top v}{\|u\|\|v\|} = \cos(\phi)$

  where ϕ is the angle between vectors u and v.
- Properties:
  - 1 if vectors point in the same direction ($\phi = 0^\circ$)
  - 0 if orthogonal ($\phi = 90^\circ$)
  - 1 if opposite direction ($\phi = 180^\circ$)

> 💡 for Cosine Similarity only thing about angle not about length
> why because smaller angle means higher similarity.

- Used for analogy and similarity tasks due to length-normalization.

Visualization Techniques



- Techniques like t-SNE reduce high-dim embeddings (e.g., 300D) to 2D for visualization.
- However, t-SNE's nonlinear projection may distort vector difference and analogy relationships.
- Parallelogram property (vector arithmetic for analogy) holds reliably in original embedding space, not always in low-dimensional projections.

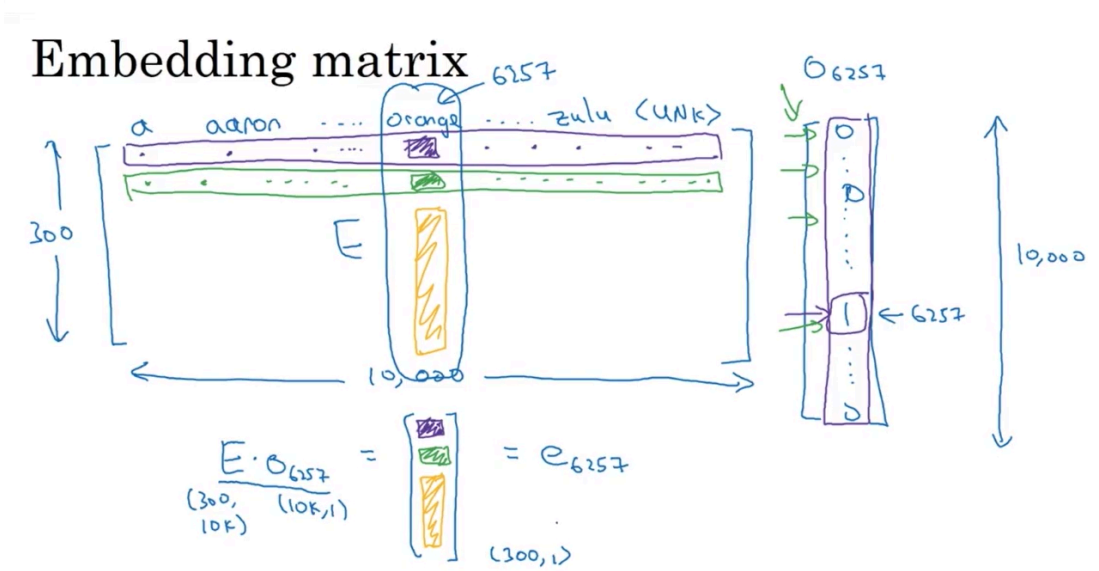
Generalization Examples of Analogies Learned

- Gender analogies: man : woman :: boy : girl
- Capital cities: Ottawa : Canada :: Nairobi : Kenya
- Comparatives: big : bigger :: tall : taller
- Currency-country relations: yen : Japan :: ruble : Russia

Summary and Intuition

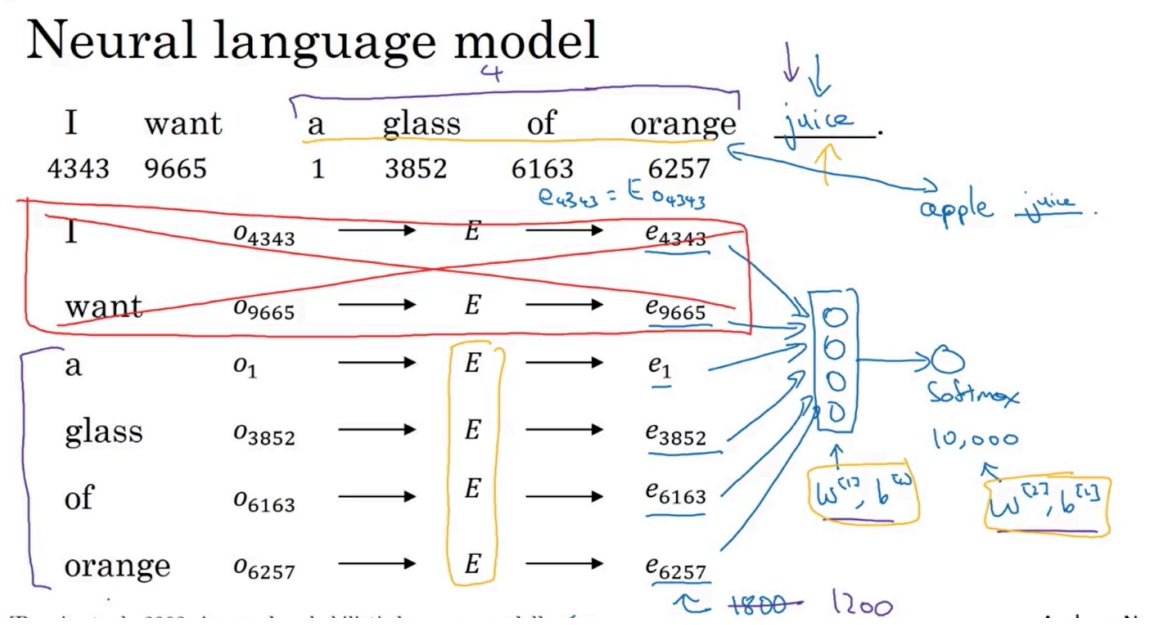
- Word embeddings capture rich feature-like properties allowing machines to reason about semantic relationships.
- Vector arithmetic encodes complex linguistic analogies.
- Cosine similarity is a crucial tool to measure embedding relatedness.
- This enables NLP models to generalize from data in meaningful ways beyond one-hot encoding limitations.

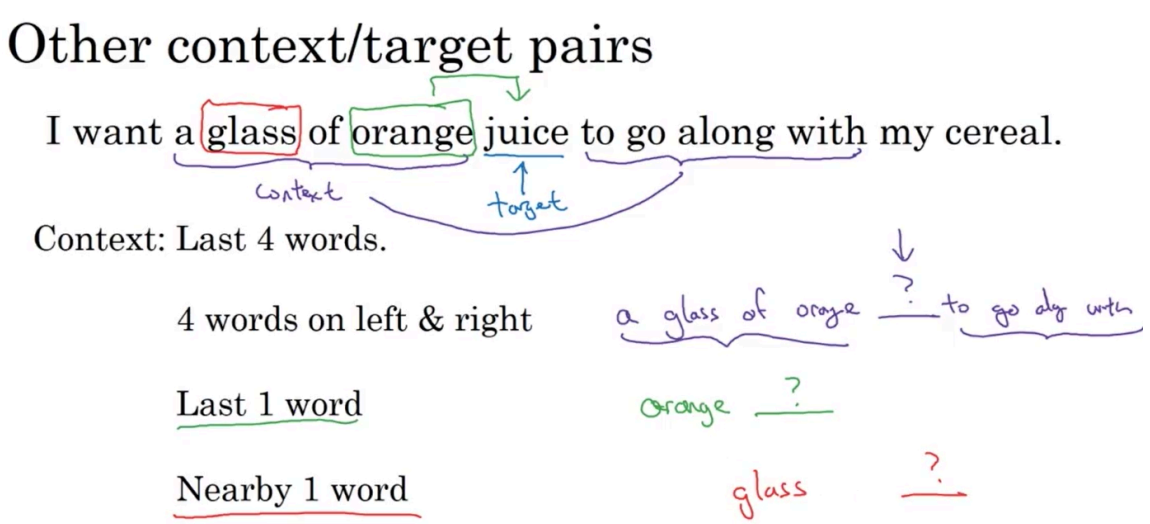
Embedding Matrix



$$E_{300\times 10k} \cdot O_{10k\times 1} = e_{300\times 1}$$

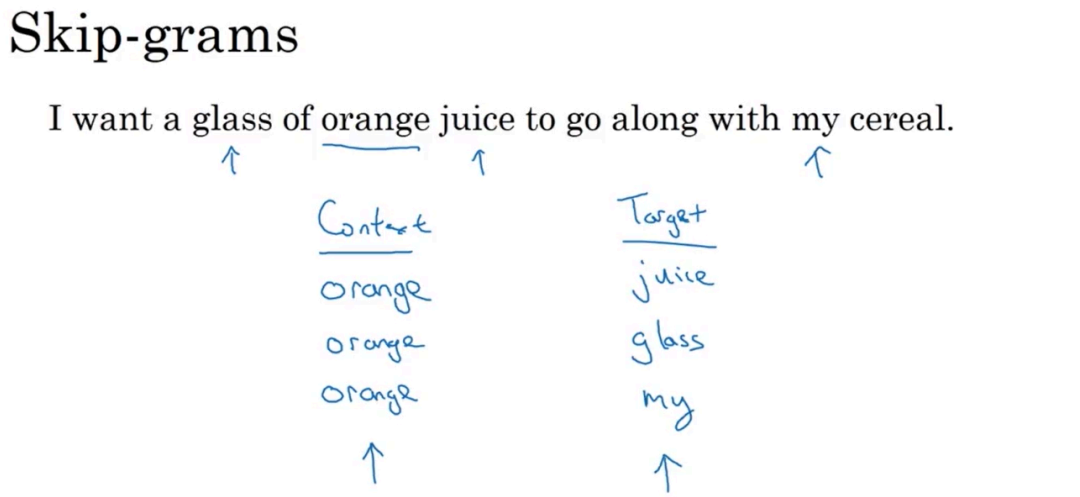
Learning Word Embeddings





Word2Vec Algorithm

1. **Skip-Gram Model**

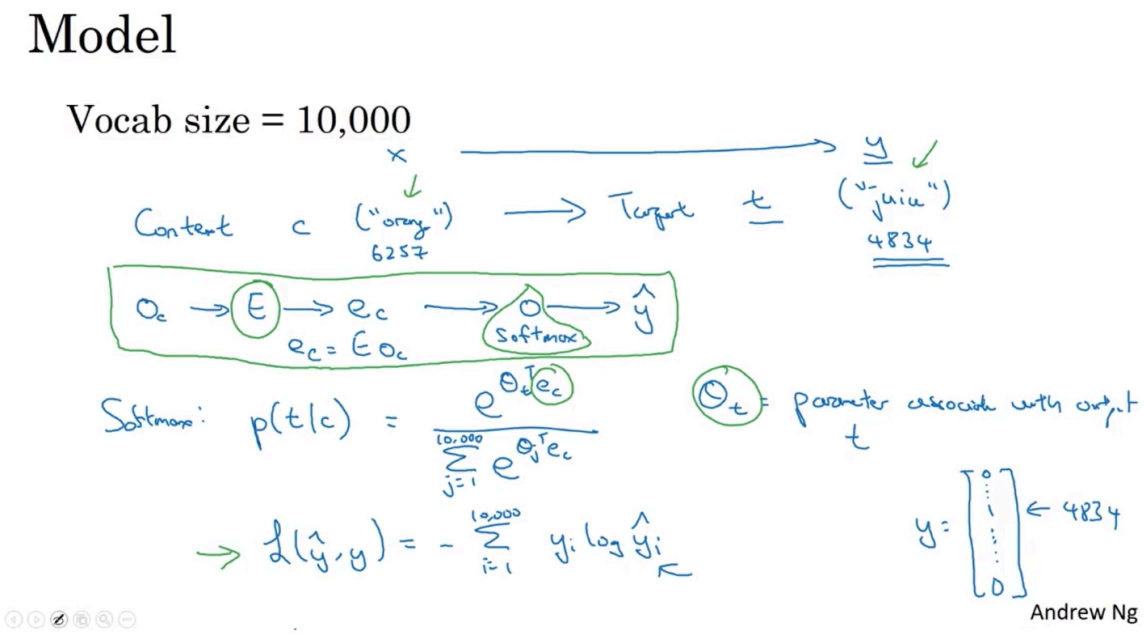


- **Goal:** Learn word embeddings by predicting context words given a target word.
- **Setup:**
  - Given a sentence, for each word (context c), randomly select another word (target t) within a window (e.g., ±5 or ±10 words).
  - Form supervised pairs: (context, target).

2. **Notation**

- V: Vocabulary size (e.g., 10,000)
- O_c: One-hot vector for context word c (size V×1)
- E: Embedding matrix (size n×V, e.g., 300×10,000)
- $e_c = EO_c$: Embedding vector for context word c (size n×1)
- θ_t: Softmax parameter vector for target word t

3. **Model Architecture**



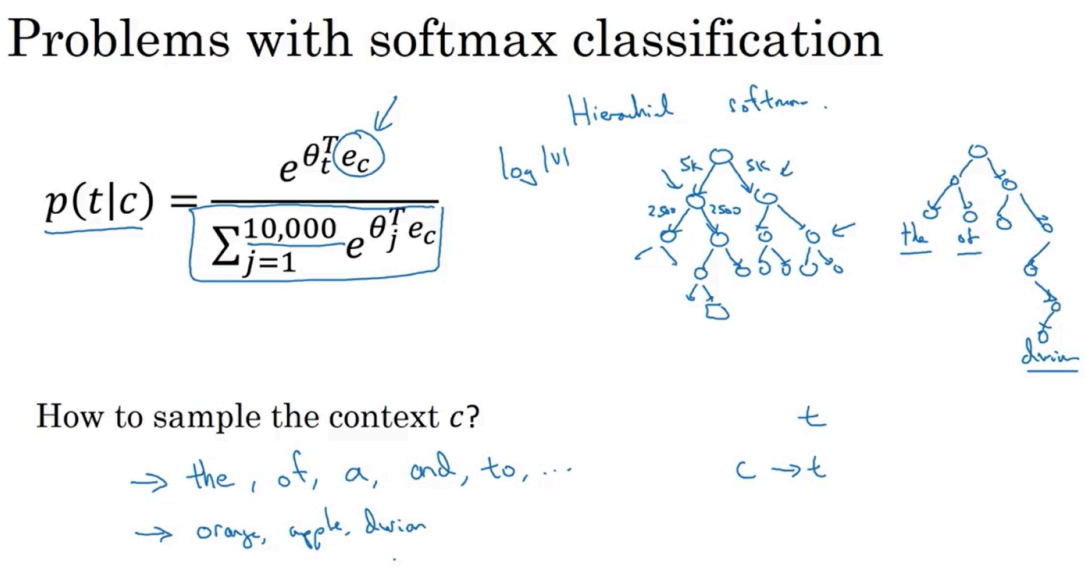
- **Input:** One-hot vector $O_c$for context word c
- **Embedding lookup:** $e_c = EO_c$
- **Softmax output:**
  $P(t|c) = \frac{e^{(\theta_t^T e_c)}}{\sum_{j=1}^{V} e^{(\theta_j^T e_c)}}$
  - Predicts probability of each word in vocab as the target.

4. **Loss Function**

- **Cross-entropy loss:**
  $L = -\sum_{i=1}^{V} y_i \log \hat{y}_i$
  - y: One-hot vector for true target word

- $\hat{y}$: Softmax output vector

5. **Computational Challenge**



- **Softmax denominator:** Requires summing over all V words (slow for large vocabularies).
- **Solution:**
  - **Hierarchical Softmax:** Organize words in a binary tree; each node is a binary classifier. Reduces computation from $O(V)$ to $O(\log V)$.
  - **Practical tree:** Common words near the root (faster access), rare words deeper.

6. **Context Sampling**

- **Uniform sampling:** Over-represents frequent words (e.g., "the", "of").
- **Heuristics:** Adjust sampling to balance frequent and rare words, ensuring all embeddings are updated.

7. **CBOW (Continuous Bag of Words) Variant**

- **Alternative:** Predict target word from surrounding context words (opposite of skip-gram).
- **Both** skip-gram and CBOW are used in Word2Vec; each has pros/cons.

8. **Key Takeaways**

- **Word2Vec** learns embeddings by predicting context-target pairs.
- **Skip-gram:** Input = context, predict = target.

- **CBOW:** Input = context window, predict = center word.
- **Embeddings** enable generalization and capture semantic similarity.

In the skip-gram with softmax, **the denominator is recalculated each time** — it depends on the context word's embedding and changes for every prediction.
Let's clarify:

Softmax Formula :

$$P(\text{target word} = j \mid \text{context word}) = \frac{\exp(\theta_j^\top e_c)}{\sum_{k=1}^{V} \exp(\theta_k^\top e_c)}$$

- **Numerator:** $\exp(\theta_j^\top e_c)$ — this changes depending on which target word j you're evaluating.
- **Denominator:** $\sum_{k=1}^{V} \exp(\theta_k^\top e_c)$ — you must sum this over **all words in the vocabulary** for each center word.

Key Point:

- The **denominator is the same for all target words** for a particular (center) context word, since it only depends on e_c for that context word.
- But **when you move to a different context word**, or during another prediction, the denominator recalculates because e_c is different.

During Training:

- **For a single center word and its windowed context:** Compute the denominator **once** (it's the normalization constant for that center/context word).
- **For each true target word in that window:** Numerator is just the dot product for that specific word.
- **For negative sampling or tricks like hierarchical softmax:** Full denominator computation is avoided.

To Summarize:

- **Yes, denominator is shared for all possible target words for one context word prediction, and you only need to sum once per prediction.**
- But, it **must be recomputed for every new context word (center word) prediction** — not reused globally across all steps.

This is why calculating it is expensive for large vocabularies, and why tricks like negative sampling and hierarchical softmax are used!

After applying the softmax, you get:

- **A 10000×1 vector p(t|c)**
  - Each element corresponds to the probability of one word (out of 10,000) appearing as the target word, given your context word.
  - All values are between 0 and 1, and the entire vector **sums to 1.**

This vector represents the full probability distribution over your whole vocabulary for that specific prediction step.

💡 **NOTE:** always remember about the embedding matrix it the key to bond relationship between words (similarities)

Industry Example: Skip-Gram with Softmax

Let's walk through a **real-world, industry-standard flow** for skip-gram Word2Vec with softmax—using examples you'd actually see in pre-training for digital assistants, search, or large-scale NLP at companies like Google or Microsoft.

---

1. Data Preparation: Building Training Pairs

Imagine you have a large text corpus from online news articles:

- Example sentence: *"The apple juice was served in a glass."*

Suppose our **vocabulary size** V=10,000 and **embedding size** N=300.

- **Sliding Window:** Let's use a context window of 2.
- **Generate training pairs (center, target):**
  - ("apple", "The")
  - ("apple", "juice")
  - ("apple", "was")
  - ("apple", "served")

Repeat for every possible center word and its neighbors in the corpus.

---

2. Neural Network Step

For each training pair:

- **Input:** Center word (e.g., "apple"), represented as a one-hot vector (all zeros except 1 at the position for "apple").
- **Embedding Lookup:** Multiply with the embedding matrix E to get the context embedding e_c, of shape 300×1.
- **Softmax Layer:** For every target word t in the vocabulary (t=1 to 10,000), calculate:$P(t \mid c) = \frac{\exp(\theta_t^\top e_c)}{\sum_{j=1}^{10,000} \exp(\theta_j^\top e_c)}$

    - θ_j is the output embedding (a 300D vector) for word j.
    - The numerator is highest for target words related to "apple" (like "juice").
- **Prediction Output:** You get a 10,000×1 softmax vector—each position is the model's predicted probability that the context word "apple" appears next to each possible word.

---

3. Training (Supervised Signal)

- The "true" target (say, "juice") is known from your sliding window pairs.
- Compute cross-entropy loss: Softmax output is compared to the one-hot vector for the true target (only "juice" is 1; rest zeros).
- **Backpropagation:** Gradients update both input embedding matrix **and** output embedding matrix—that is, both e_c and θ_t are learned.

---

4. Industry Application

- **Google Search Autocomplete:** This process steers embeddings so that words that often occur together (“apple juice”) have similar embeddings. The same process (on huge text data) trains embeddings to power search, query suggestions, and digital assistant understanding.
- **Production Example:**
    - Center word: "bank" (context: "river bank is flooded")
    - Model learns “river” and “bank” are related, not just “bank” and “money”
- **Scale:** Industry systems use billions of training pairs, huge vocabularies, and often replace vanilla softmax with negative sampling or hierarchical softmax for speed—but the pairing/softmax step remains the same in principle.

---

5. Key Intuition Review

- **For every (context, target) pair, softmax predicts the probability of every word being the target.**
- **The true target is from your training pairs, not guessed.**
- *Training forces words that appear in similar contexts to have embeddings that are close together.

Computational Bottleneck

- Computing the denominator for each prediction requires summing over all V words.
- For large vocabularies (10K, 100K, or >1M), this is costly.

Hierarchical Softmax

- Replaces the flat softmax with a **binary tree of classifiers** (Huffman tree built from word frequencies).
- Each **word is a leaf node** in the tree.
- To compute $P(w|e_c)$, traverse from root to leaf for w, multiply binary decisions.
- At each internal node n_i, compute :
  $P(\text{go left}|e_c) = \sigma(n_i^\top e_c), \quad P(\text{go right}|e_c) = 1 - \sigma(n_i^\top e_c)$
- Final probability:$P(w|e_c) =$
  $\prod_{i=1}^{L} \begin{cases} \sigma(n_i^\top e_c), & \text{if left branch } 1 - \sigma(n_i^\top e_c), \text{ if right branch} \end{cases}$
- Computational cost is reduced to O(logV) compared to O(V).

Tree Structure

- Huffman tree built from word frequency.
- Most frequent words have shortest paths (near root).
- Each **internal node stores a trainable vector** for binary classification.
- Leafs represent words (word ID), no parameters stored at leaves.

Intuition

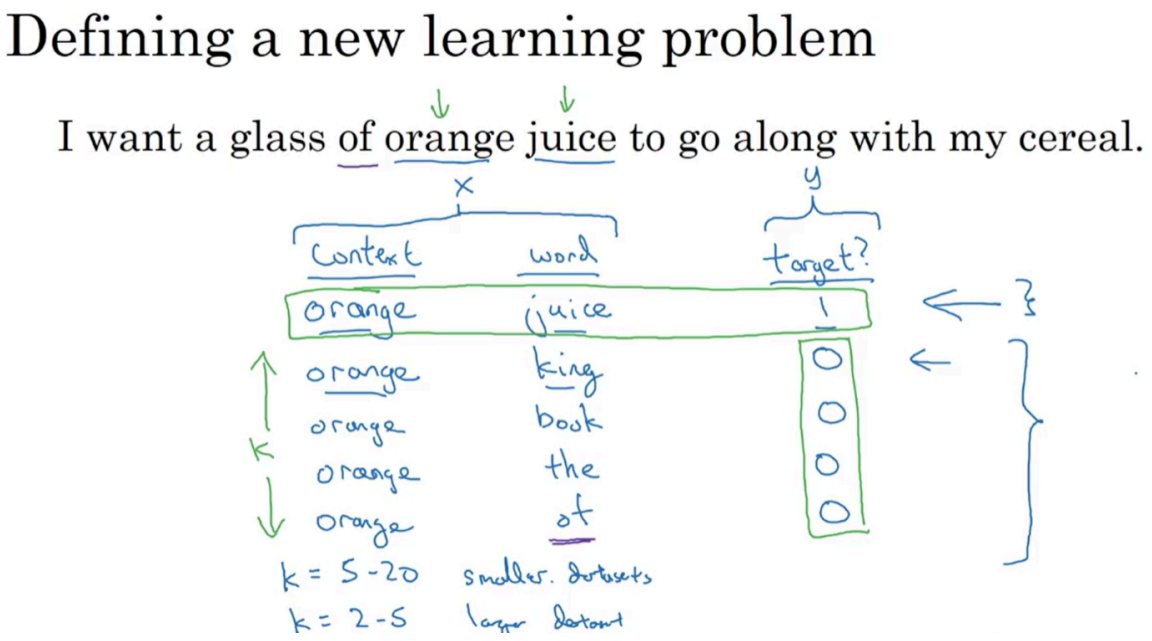
- Embeddings capture semantic meaning by placing similar words closer in vector space.
- The softmax model learns to assign high probabilities to likely context-target pairs.
- Hierarchical softmax speeds this up by breaking prediction into a sequence of binary decisions.

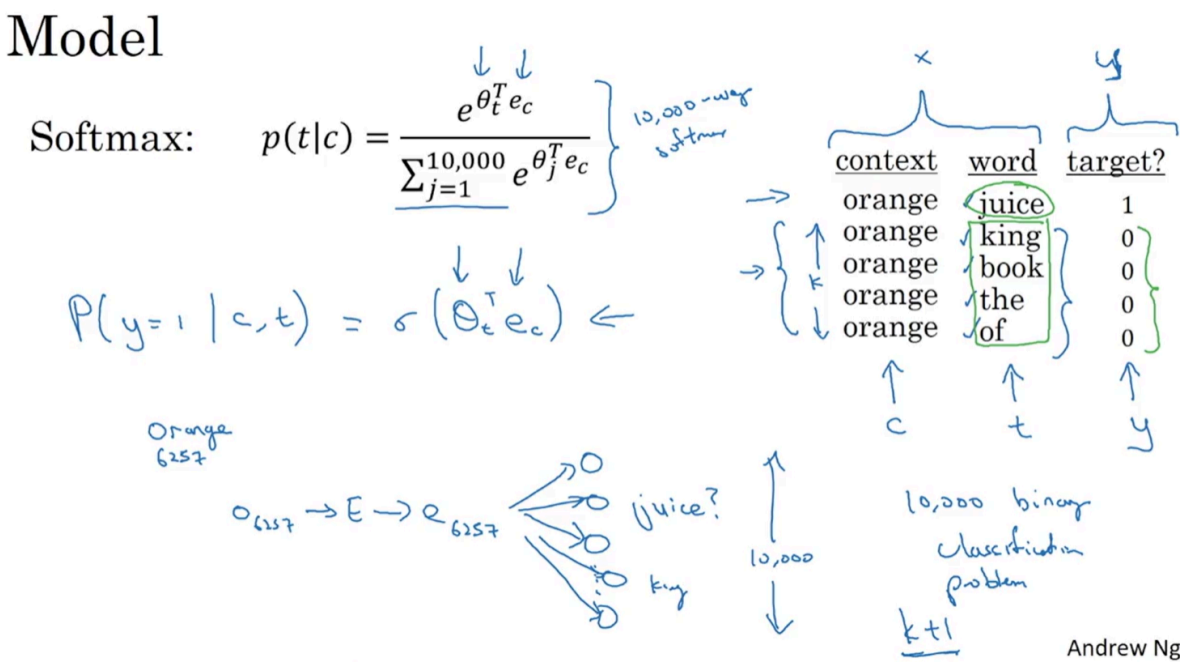
Summary

Concept	Description
Input Encoding	One-hot vector for context word

Concept	Description
Embedding	Dense vector e_c extracted from matrix
Softmax Output	Probability over vocabulary for target word
Problem	Softmax expensive for large vocabularies
Hierarchical Softmax	Binary tree classifier to replace full softmax
Efficiency Gain	O(logV)

Negative Sampling





Objective:

Efficiently learn word embeddings by converting multi-class prediction (via softmax) into multiple cheap binary classification tasks.

Training Flow:

- Start with a large text corpus, fixed vocabulary (e.g., 10,000 words), embedding dimension (e.g., 300).
- For each sentence:
  - Select a **center word**.
  - Within a window, pick **real context words** → form positive pairs (label = 1).
  - For each positive, **sample K negatives** (random words from vocabulary) → negative pairs (label = 0).
- Only update the K+1 relevant embeddings at each step.

How Labels Are Assigned:

- **Positive label (1):** Center word with true context (from window).
- **Negative label (0):** Center word with random non-context word, sampled K times per positive.

Training Step:

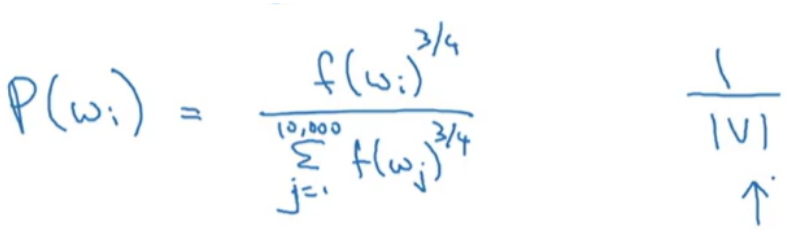
- Each training batch = 1 positive + K negatives for a center.
- Train a simple **logistic regression (sigmoid)** model to predict if the pair is genuine (label 1) or fake (label 0).

Efficiency:

- Instead of updating all classifiers (for each word) per step, you only update those for K+1 involved words, keeping computation cheap and scalable.

Negative Sample Selection:

- Optimal negative selection uses:
- Probability of picking a word $\propto frequency(word)^{3/4}$ in corpus (empirical heuristic works best).



Result:

- After sweeping the corpus:
  - Each word gets a vector representation.
  - Similar words end up close in embedding space.
  - Embeddings can power analogy, similarity, NLP tasks.

Core Takeaways:

- **Labels are set automatically (never manual):** genuine context = 1, sampled negatives = 0.
- **All words' embeddings are learned with sparse updates, but everyone gets refined by many passes.**
- **You do not mark all words outside the window as 0—only the sampled K negatives.**
- The technique allows you to train fast, generalize, and use pre-trained vectors if needed.

Summary formula:

- For center word c:
  - For each context word t in window:

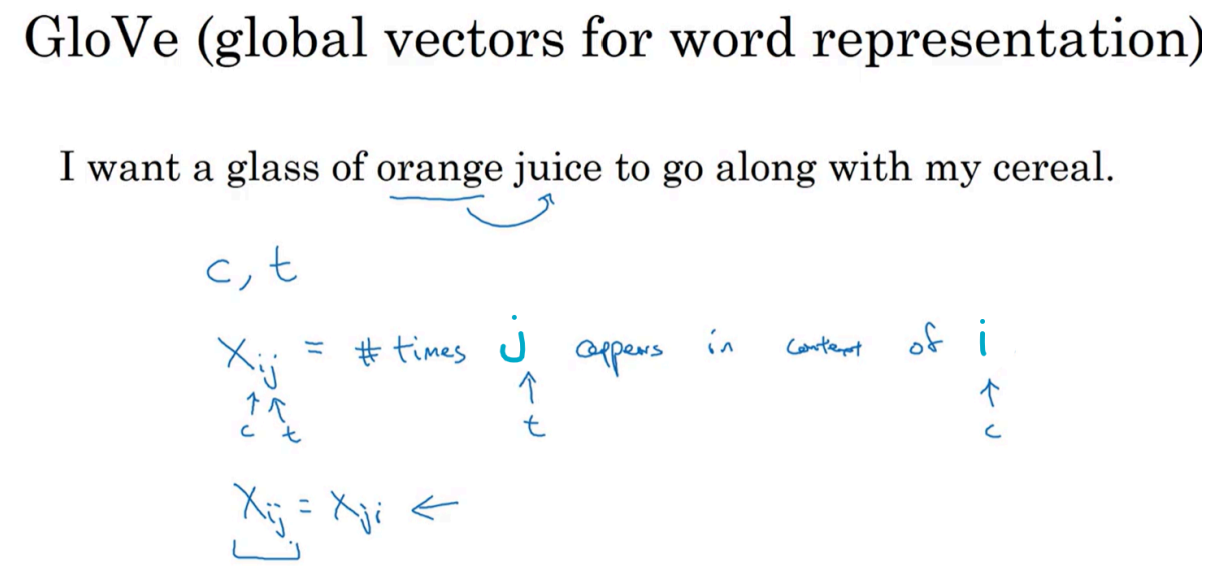
    pair = (c, t), label = 1
  - For each of K negative words n (random):

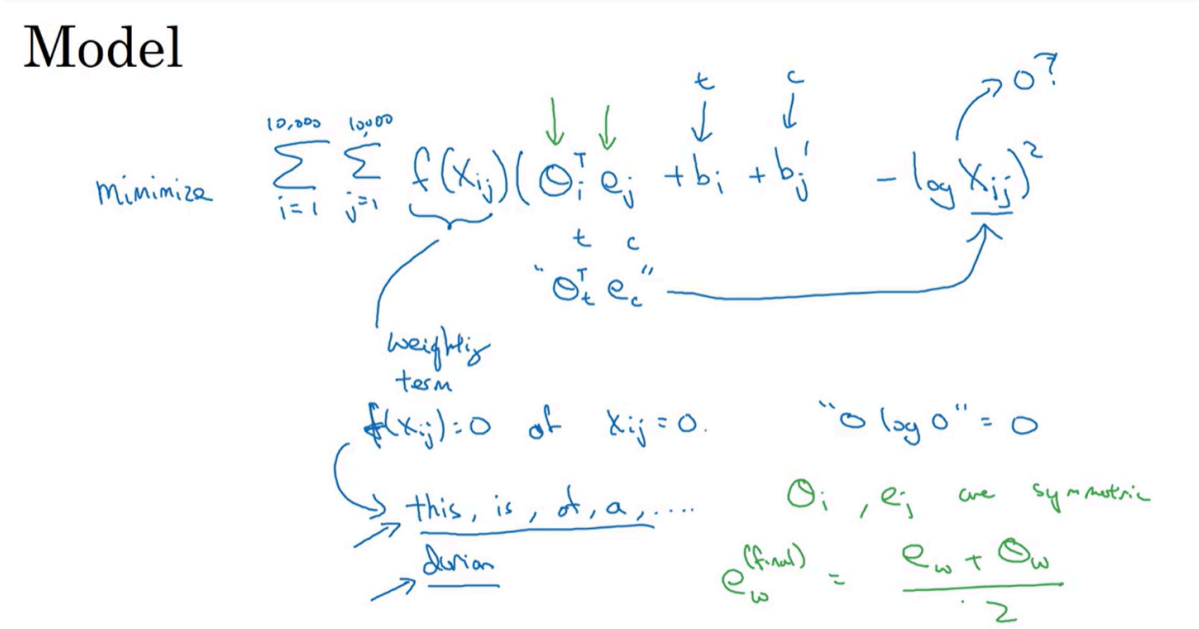
    pair = (c, n), label = 0

End Result:

- Embedding matrix of shape (vocabulary size × embedding dimension).
- Semantic richness, efficiency, fast generalization.

Global Vectors for word representation (GloVe)





GloVe (Global Vectors for Word Representation) is a popular word embedding algorithm designed to capture global word relationships from a text corpus. Here's everything you need for a speedy, strong understanding:

1. Core Idea

- **Counts Co-occurrences**: GloVe builds on the insight that the way words co-occur (appear together) in text reveals their meaning. It directly counts how often word i and word j appear together within a window (e.g., ±10 words).
- **Co-occurrence Matrix**: Defines X_{ij} as the number of times word j (target word) appears in the context of word i (context word).

2. Objective Function

- **Goal**: Learn word vectors so that their inner product approximates the log of their co-occurrence count: Minimize $\sum_{i,j=1}^{V} f(X_{ij}) \left[\theta_i^T e_j - \log(X_{ij})\right]^2$
  - θ_i, e_j: Word vector representations (symmetric roles in GloVe!)
  - $f(X_{ij})$: Weighting function, gives less importance to rare pairs, avoids heavy dominance from frequent ("stop") words
  - Only sums over pairs where $X_{ij} > 0$

3. Symmetry & Averaging

- Unlike Word2Vec, GloVe treats the two sets of word vectors symmetrically. After training, for each word, you take the **average** of its two learned vectors as the final embedding:

$$\text{Final embedding} = \frac{\theta_{\text{trained}} + e_{\text{trained}}}{2}$$

4. Key Points About the Algorithm

A note on the featurization view of word embeddings

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	
Gender	−1	1	-0.95	0.97	←
Royal	0.01	0.02	0.93	0.95	←
Age	0.03	0.02	0.70	0.69	←
Food	0.09	0.01	0.02	0.01	←

$$\text{minimize} \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$$

- **Simple Loss = Effective**: Despite a seemingly basic cost function, GloVe produces high-quality embeddings.
- **Weighting Function f**: Needs careful tuning. It's designed to not ignore infrequent words and not overemphasize ultra-common ones.
- **Not Interpretable Axes**: The dimensions of the embeddings are not guaranteed to represent clear semantic features (like gender or royalty), due to invariance under certain linear transformations. You can't point to "dimension 1 = gender."

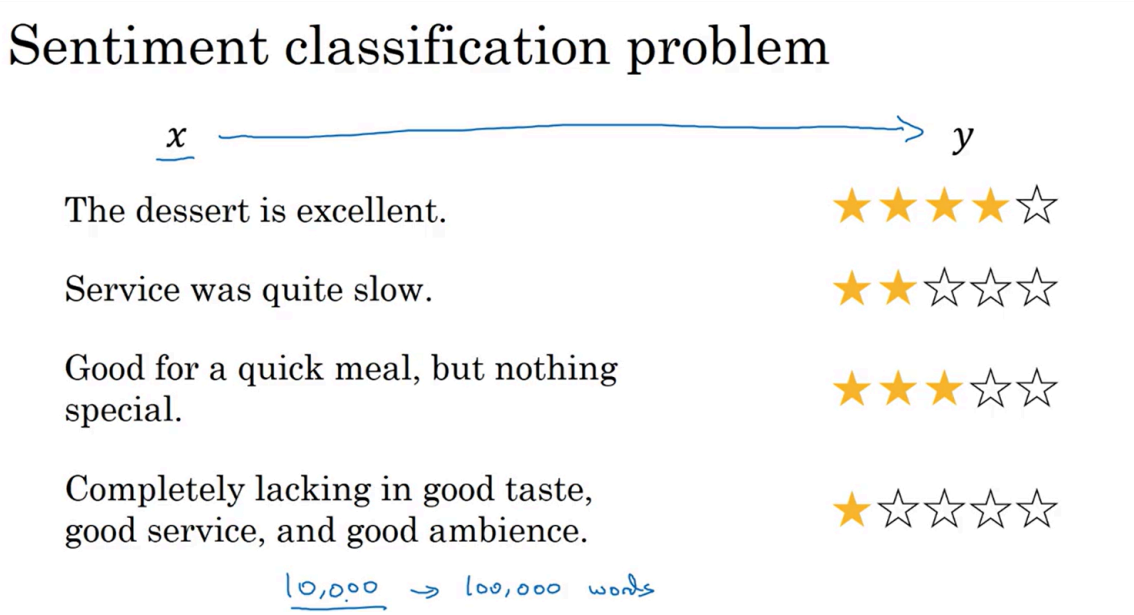
5. Practical Advantages

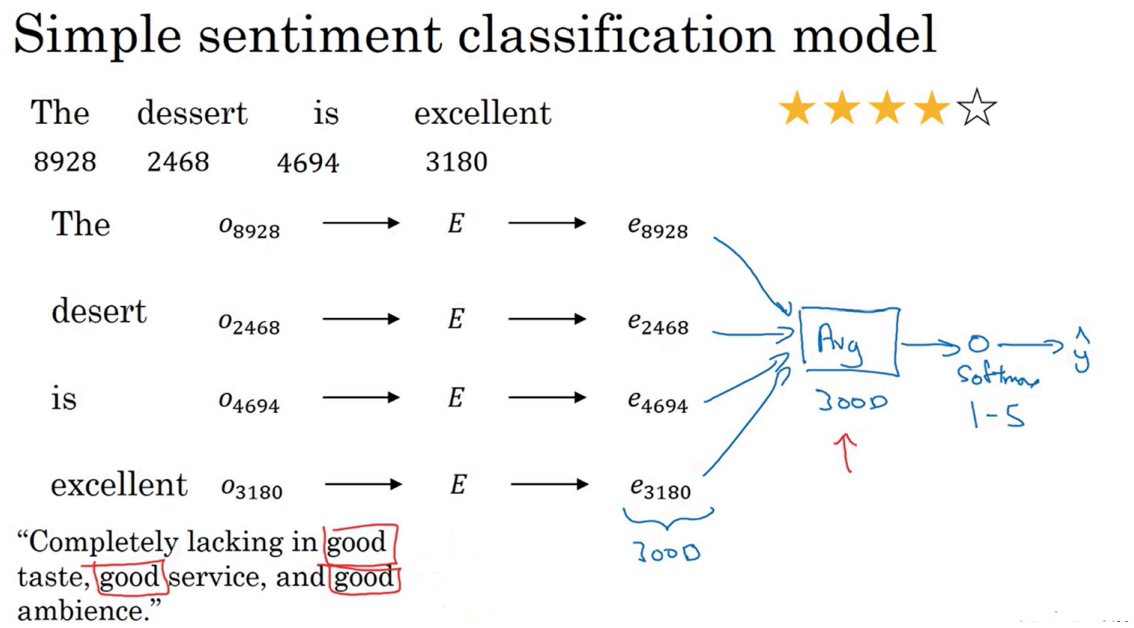
- **Captures Both Local & Global Structure**: Uses global counts (not just local context as with Word2Vec), often resulting in richer semantic relationships.
- **Widely Used**: Pretrained GloVe vectors are available for many languages/tasks. Often used as a first step in NLP pipelines.

6. Analogies & Feature Relationships

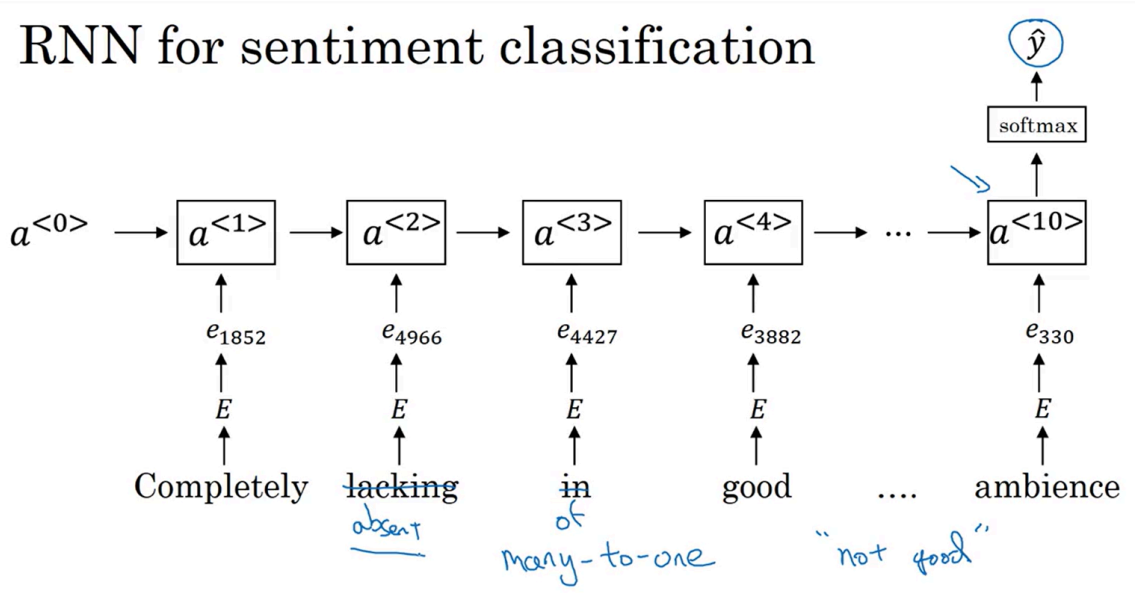
- Operations like $queene_{king} - e_{man} + e_{woman} \approx e_{queen}$ still work, even though the axes aren't human-interpretable.

Sentiment Classification





this is very bad model because it only takes average statically ok but lack the context and word order leads to bad results



Debiasing Word Embeddings

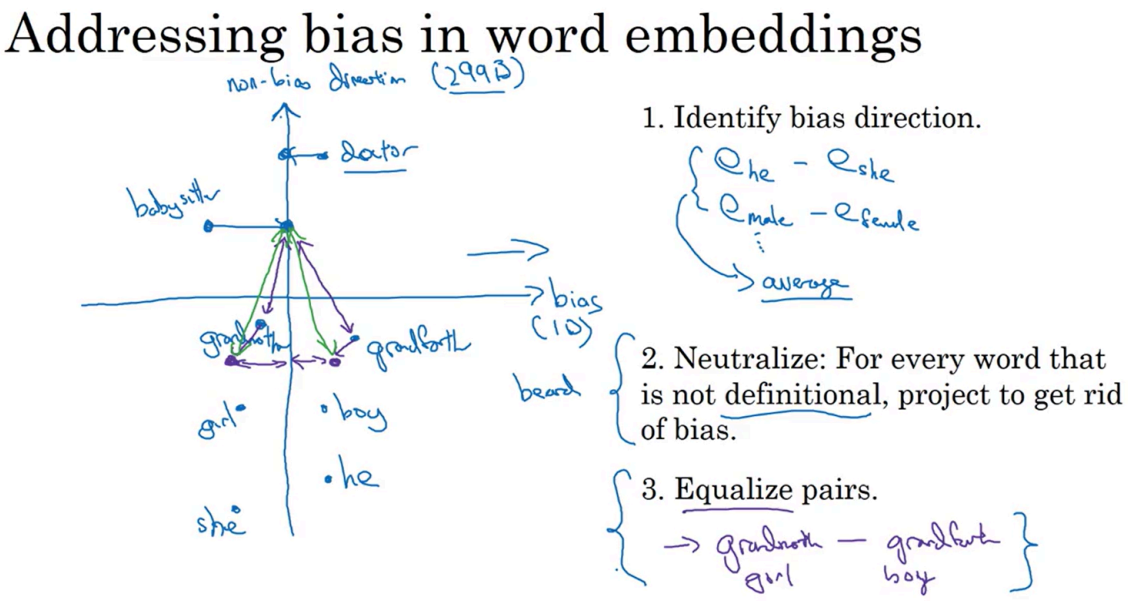
The problem of bias in word embeddings

Man:Woman as King:Queen

Man:Computer_Programmer as Woman:Homemaker

Father:Doctor as Mother:Nurse

Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model.



Goal:

- Reduce or eliminate unwanted biases (e.g., gender, ethnicity) from word embeddings.

Common problem:

- Word embeddings can encode social biases from training text.
- Example:

  $$\text{man : computer programmer :: woman :?}$$

  returns "homemaker" instead of "computer programmer".

Notation

- e_w: Embedding vector for word w.
- B: Bias direction (vector or subspace).
- D: Set of definitional word pairs (e.g., (he, she), (man, woman)).
- N: Set of neutral words (should be bias-free, e.g., "doctor").

Debiasing Algorithm (Bolukbasi et al.)

Step 1: Identify bias direction

- Compute difference vectors: $e_{w_1} - e_{w_2}$ for definitional pairs (e.g., (he, she)).
- Use PCA or SVD on these vectors to get bias direction B.
  - For gender:

    $$B = \text{principal component of } \{e_{he} - e_{she}, e_{man} - e_{woman}, ...\}$$

Step 2: Neutralization

- For each neutral word $w \in N$:
  - Remove bias component:$e'_w = e_w - \text{proj}_B(e_w)$
  - $\text{proj}_B(e_w) = \frac{e_w \cdot B}{\|B\|^2}$
- Applies to all words not inherently defined by the bias (e.g., "doctor", "nurse").

Step 3: Equalization

- For each pair of words (w_1, w_2) in D:
  - Move them to be equidistant from the bias direction.
  - Let $\mu = \frac{e_{w_1} + e_{w_2}}{2}$, the midpoint.
  - Set their embeddings so:
    - Both have same projection on B, but opposite sign
    - Both are equally similar to neutral words

Selecting Neutral and Definitional Words

- Definitional: Gender/ethnicity/other trait is intrinsic (e.g., "grandfather", "grandmother").
- Neutral: Should have no bias (e.g., "doctor", "babysitter").
- Usually, a linear classifier or manual curation is used.

Main Points

- **Bias direction:** Key axis along which to debias (B).
- **Neutralization:** Remove bias from non-definitional embeddings.
- **Equalization:** Make pairs (like "man"/"woman") symmetric around bias axis.
- **Practical:** Only a small subset of words is definitional; most can be neutralized.
- **General:** Method applicable to any bias once a direction is identified.