Week 1

Comprehensive Guide to Word Embeddings

Contents

1 Introduction to Word Embeddings

Humans communicate using natural language, a system where words act as the basic units of meaning. To enable computers to understand this data, we must convert these discrete symbols into numerical representations.

1.1 The Problem with One-Hot Vectors

Historically, words were represented using **one-hot vectors**. If a vocabulary $\mathcal{V}$ has size N, the i-th word is represented by a vector $\mathbf{x} \in \{0,1\}^N$ where the i-th component is 1 and all others are 0.

While easy to construct, this approach has a critical flaw: it treats all words as equidistant. For any two distinct words x and y, the dot product $\mathbf{x}^\top \mathbf{y} = 0$, meaning the cosine similarity is always 0. This representation cannot capture that "dog" and "puppy" are semantic neighbors while "dog" and "microwave" are not.

1.2 Distributed Representations

The solution is **word embedding**, which maps words to dense vectors in a lower-dimensional real space $\mathbb{R}^d$ (where $d \ll N$, typically 50 to 1000). These are often called distributed representations because the "meaning" of a word is distributed across the dimensions of the vector. The central hypothesis driving this is the *distributional hypothesis*: words that appear in similar contexts tend to have similar meanings.

2 Mathematical Foundations: Vector Calculus for NLP

Before diving into specific models, it is crucial to establish the mathematical framework used to train them. Training embedding models involves optimizing a loss function $J(\theta)$ with respect to high-dimensional vector parameters.

2.1 Notation and Basics

We denote vectors with bold lowercase letters (e.g., $\mathbf{x}$) and matrices with bold uppercase letters (e.g., $\mathbf{W}$).

- Gradient vector: $\nabla_{\mathbf{x}} f(\mathbf{x})$ is a vector of partial derivatives.
- Jacobian matrix: If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, then $\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$.

2.2 Key Identities for Backpropagation

The following identities are essential when deriving gradients for Word2Vec and GloVe.

2.2.1 Gradient of a Dot Product

Let $\mathbf{a}$ be a constant vector and $\mathbf{x}$ be a variable vector. The derivative of their dot product is:

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^\top \mathbf{a}) = \mathbf{a} \tag{1}$$

This rule is used constantly when differentiating the score $s = \mathbf{u}^\top \mathbf{v}$.

2.2.2 Gradient of Euclidean Norm Squared

For GloVe, we minimize squared errors. The gradient of a squared norm is:

$$\frac{\partial}{\partial \mathbf{x}} \|\mathbf{x} - \mathbf{a}\|_2^2 = 2(\mathbf{x} - \mathbf{a}) \tag{2}$$

2.3 Calculus of the Softmax Function

Word2Vec relies heavily on the Softmax function. Let $\mathbf{z}$ be a vector of scores (logits), and $\mathbf{y} = \text{softmax}(\mathbf{z})$.

$$y_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

The partial derivative of the output y_i with respect to the input z_j is given by the Jacobian:

$$\frac{\partial y_i}{\partial z_j} = y_i(\delta_{ij} - y_j) \tag{3}$$

where δ_{ij} is the Kronecker delta (1 if $i = j$, 0 otherwise).

- If $i = j$: $\frac{\partial y_i}{\partial z_i} = y_i(1 - y_i)$
- If $i \neq j$: $\frac{\partial y_i}{\partial z_j} = -y_i y_j$

2.4 Gradient of Log-Softmax (Cross Entropy)

In Skip-Gram, we minimize $-\log P(w_o|w_c)$. Let P be the probability vector from softmax. The gradient of the negative log-probability of the true class o with respect to the input logits $\mathbf{z}$ is simply:

$$\nabla_{\mathbf{z}}(-\log y_o) = \mathbf{y} - \mathbf{e}_o \tag{4}$$

where $\mathbf{e}_o$ is the one-hot vector for the true class. This result—that the gradient is simply "prediction minus target"—is ubiquitous in machine learning.

3 Word2Vec: Skip-Gram and CBOW

The **word2vec** tool, introduced by Mikolov et al. (2013), revolutionized NLP by providing efficient methods to learn these embeddings from massive raw corpora using self-supervised learning. It includes two primary architectures: Skip-Gram and Continuous Bag of Words (CBOW).

3.1 The Skip-Gram Model

The Skip-Gram model flips the traditional language modeling problem. Instead of using context to predict a word, it uses a **center word** to predict its **surrounding context words**.

3.1.1 Probabilistic Model

Consider a text sequence of length T, where the word at time step t is $w^{(t)}$. We define a context window of size m. For a specific center word w_c, the model tries to generate the context words w_o that fall within distance m:

$$\text{Context} = \{w^{(t+j)} \mid -m \leq j \leq m, j \neq 0\}$$

We assume that given the center word, the context words are generated independently. The likelihood function for the entire corpus is the probability of generating all context words given all center words:

$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w^{(t+j)} \mid w^{(t)})$$

3.1.2 Neural Architecture View

To understand the mechanics, we can view Skip-Gram as a shallow neural network with three layers:

1. **Input Layer:** A one-hot encoded vector $\mathbf{x} \in \mathbb{R}^{|\mathcal{V}|}$ representing the center word.
2. **Projection (Hidden) Layer:** A weight matrix $\mathbf{W} \in \mathbb{R}^{d \times |\mathcal{V}|}$. The hidden state is $\mathbf{h} = \mathbf{W}\mathbf{x}$. Since $\mathbf{x}$ is one-hot, this is equivalent to simply selecting the column corresponding to the center word (lookup table operation). There is **no activation function** here. $\mathbf{h}$ is exactly the center word embedding $\mathbf{v}_c$.
3. **Output Layer:** A second weight matrix $\mathbf{U} \in \mathbb{R}^{|\mathcal{V}| \times d}$ maps the hidden state to a score vector $\mathbf{z} = \mathbf{U}^\top \mathbf{h}$. We then apply Softmax to $\mathbf{z}$ to get probabilities.

This architecture highlights that we are training a dedicated logistic regression classifier for every word in the vocabulary to distinguish its context.

3.1.3 Parameterization

Each word i in the vocabulary $\mathcal{V}$ is associated with two vectors:

1. $\mathbf{v}_i \in \mathbb{R}^d$: The vector representation when word i is a **center** word.
2. $\mathbf{u}_i \in \mathbb{R}^d$: The vector representation when word i is a **context** word.

The probability $P(w_o|w_c)$ is modeled using the **softmax** function applied to the dot product of the vectors. The dot product $\mathbf{u}_o^\top \mathbf{v}_c$ measures the similarity between the center word c and the context word o:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \tag{5}$$

3.1.4 Training Objective

To learn the embeddings, we minimize the negative log-likelihood loss function:

$$J = -\sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w^{(t+j)} \mid w^{(t)})$$

Using the definition of softmax, the logarithmic loss for a single pair (w_c, w_o) is:

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right) \tag{6}$$

3.1.5 Gradients and Derivation

To train via Stochastic Gradient Descent (SGD), we need the gradient of the loss with respect to the parameters.

Mathematical Derivation: Skip-Gram Gradient w.r.t Center Vector

Let $l = -\log P(w_o \mid w_c)$. We want $\frac{\partial l}{\partial \mathbf{v}_c}$. Substitute the log-softmax equation:

$$l = -\left(\mathbf{u}_o^\top \mathbf{v}_c - \log \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$$

Differentiate with respect to $\mathbf{v}_c$:

$$\frac{\partial l}{\partial \mathbf{v}_c} = -\mathbf{u}_o + \frac{\partial}{\partial \mathbf{v}_c} \log \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)$$

Using the chain rule for the log term ($\frac{\partial}{\partial x} \log f(x) = \frac{1}{f(x)} f'(x)$):

$$\frac{\partial l}{\partial \mathbf{v}_c} = -\mathbf{u}_o + \frac{1}{\sum_i \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \cdot \sum_{j \in \mathcal{V}} \frac{\partial}{\partial \mathbf{v}_c} \exp(\mathbf{u}_j^\top \mathbf{v}_c)$$

The derivative of the exp term is $\exp(\mathbf{u}_j^\top \mathbf{v}_c) \cdot \mathbf{u}_j$:

$$\frac{\partial l}{\partial \mathbf{v}_c} = -\mathbf{u}_o + \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_i \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j$$

Recognizing the softmax probability term inside the sum:

$$\frac{\partial l}{\partial \mathbf{v}_c} = -\mathbf{u}_o + \sum_{j \in \mathcal{V}} P(w_j \mid w_c) \mathbf{u}_j \tag{7}$$

This reveals an intuitive update rule: we move $\mathbf{v}_c$ closer to the observed context word vector $\mathbf{u}_o$ and push it away from the expected context vector (weighted sum of all words).

3.2 The Continuous Bag of Words (CBOW) Model

CBOW works in the opposite direction. It predicts the **center word** w_c based on the aggregate of its **context words** $w_{o_1}, \ldots, w_{o_{2m}}$.

3.2.1 Model and Loss

Since the context contains multiple words, their vectors are averaged:

$$\bar{\mathbf{v}}_o = \frac{1}{2m} \sum_{k=1}^{2m} \mathbf{v}_{o_k}$$

The probability of generating the center word is:

$$P(w_c \mid \text{context}) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}$$

The loss function is again the negative log-likelihood:

$$J = -\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t+m)})$$

3.2.2 The Smoothing Effect of Averaging

Unlike Skip-Gram, which expands one context window into $2m$ distinct training examples (one for each context word), CBOW condenses the window into a single training example.

- **Regularization:** The averaging operation $\bar{\mathbf{v}}_o$ smooths out the noise. If one context word is an outlier (rare or noisy usage), its effect is dampened by the other words in the window.
- **Representation Quality:** This smoothing makes CBOW typically better at representing syntactic relationships (e.g., verb tenses) and frequent words. However, it can hurt the representation of rare words, as their specific contribution is "drowned out" by the average.
- **Efficiency:** Computationally, CBOW is faster than Skip-Gram because the potentially expensive Output Layer operation is performed once per window rather than $2m$ times.

3.2.3 Derivation of CBOW Gradients

In CBOW, we need to update the context vectors $\mathbf{v}_{o_k}$. The loss $l = -\log P(w_c \mid \mathcal{W}_o)$ depends on the average vector $\bar{\mathbf{v}}_o$.

Mathematical Derivation: CBOW Gradient w.r.t Context Vector

Using the chain rule, we first find the gradient w.r.t the average vector $\bar{\mathbf{v}}_o$:

$$\frac{\partial l}{\partial \bar{\mathbf{v}}_o} = -\mathbf{u}_c + \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j$$

Now, since $\bar{\mathbf{v}}_o = \frac{1}{2m} \sum_k \mathbf{v}_{o_k}$, the derivative w.r.t a specific context word $\mathbf{v}_{o_i}$ is:

$$\frac{\partial \bar{\mathbf{v}}_o}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \mathbf{I}$$

Combining these:

$$\frac{\partial l}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(-\mathbf{u}_c + \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j \right) \tag{8}$$

This implies that for every training step, the error signal is distributed equally among all context words in the window.

Insight: The Two Vectors Paradox

Why do we have two vectors $\mathbf{u}$ and $\mathbf{v}$ for each word? Mathematically, optimization is easier when the interactions are bilinear. If we used a single vector set $\mathbf{v}$ for both center and context, the dot product term would be $\mathbf{v}_o^\top \mathbf{v}_c$. The gradient would then involve a symmetric update that couples the variables more tightly, potentially making the optimization landscape more difficult (saddle points). However, in practice, after training, we simply discard $\mathbf{u}$ (context vectors) and use $\mathbf{v}$ (center vectors) as the final embedding, or sometimes average them $(\mathbf{v} + \mathbf{u})$.

Things to Ponder

1. **Computational Complexity:** Calculating the gradient involves a summation over the entire vocabulary $\mathcal{V}$ in the denominator. If $|\mathcal{V}| = 1,000,000$, what is the impact on training speed?
2. **Phrase Handling:** English contains fixed phrases like "New York" or "ice cream". If treated as separate words ("New", "York"), their individual meanings dominate. How might we modify the training process to learn a specific vector for "New York"? (Hint: Refer to Section 4 in the Mikolov 2013 paper).
3. **Geometric Interpretation:** In Skip-Gram, the objective maximizes $\mathbf{u}_o^\top \mathbf{v}_c$. How does this relate to cosine similarity? Why might the cosine similarity of synonyms be high after training?

4 Approximate Training Techniques

The primary bottleneck in standard word2vec is the normalization factor (the denominator) in the softmax function:

$$Z = \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)$$

Computing this sum for every single training step is prohibitively expensive when $|\mathcal{V}|$ is large. We use approximate methods to bypass this.

4.1 Negative Sampling

Negative Sampling (NEG) solves the computational problem by changing the objective. Instead of a multi-class classification problem (predicting word w out of $|\mathcal{V}|$ classes), it recasts the problem as binary classification.

4.1.1 The Logic

For a given center word w_c, we want the model to:

- Assign a high probability to the **true** context word w_o (Positive sample, label $D = 1$).
- Assign low probability to K random **noise** words w_k drawn from the vocabulary (Negative samples, label $D = 0$).

The probability of observing a positive pair is modeled by the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$:

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$$

The probability of a noise word being "negative" is:

$$P(D = 0 \mid w_c, w_k) = 1 - \sigma(\mathbf{u}_k^\top \mathbf{v}_c) = \sigma(-\mathbf{u}_k^\top \mathbf{v}_c)$$

4.1.2 Objective Function

We maximize the joint probability of the positive sample and the K negative samples:

$$\prod_{t=1}^{T}\Big(P(w^{(t+j)} \mid w^{(t)})\Big) \approx \prod_{t=1}^{T}\left(\sigma(\mathbf{u}_o^\top \mathbf{v}_c)\prod_{k=1}^{K}\sigma(-\mathbf{u}_{n_k}^\top \mathbf{v}_c)\right)$$

The logarithmic loss for a single step is:

$$J = -\log\sigma(\mathbf{u}_o^\top \mathbf{v}_c) - \sum_{k=1}^{K}\log\sigma(-\mathbf{u}_{n_k}^\top \mathbf{v}_c) \tag{9}$$

The computational cost is now proportional to K (typically 5 to 20), rather than $|\mathcal{V}|$.

4.1.3 Gradient Analysis of Negative Sampling

To understand the mechanics, consider the gradient for a noise word w_k:

$$\frac{\partial J}{\partial \mathbf{u}_k} = -\frac{\partial}{\partial \mathbf{u}_k}\log\sigma(-\mathbf{u}_k^\top \mathbf{v}_c)$$

Using $\sigma'(x) = \sigma(x)(1-\sigma(x))$, the gradient simplifies to:

$$\frac{\partial J}{\partial \mathbf{u}_k} = \sigma(\mathbf{u}_k^\top \mathbf{v}_c)\mathbf{v}_c$$

This simply means if the dot product $\mathbf{u}_k^\top \mathbf{v}_c$ is large (model falsely thinks noise is context), the sigmoid term is large, and we get a strong gradient pushing $\mathbf{u}_k$ away from $\mathbf{v}_c$.

4.1.4 Sampling Distribution

Noise words are not sampled uniformly. They are sampled from the unigram distribution raised to the power of 0.75: $P(w) \propto U(w)^{0.75}$. This heuristic increases the probability of sampling rare words as negatives compared to their raw frequency, which empirically improves embedding quality.

4.2 Hierarchical Softmax

Hierarchical Softmax replaces the flat softmax layer with a hierarchical structure (a binary tree), usually a Huffman tree where frequently used words are closer to the root.

4.2.1 Tree Structure

The leaves of the tree are the words in the vocabulary. There are $|\mathcal{V}| - 1$ internal nodes. Each internal node n has a vector $\mathbf{u}_n$. The probability of a word w is the product of probabilities of taking the correct left/right turn at every node on the path from the root to the leaf w.

Let $L(w)$ be the length of the path. Let $n(w, j)$ be the j-th node on the path. The probability is:

$$P(w \mid w_c) = \prod_{j=1}^{L(w)-1}\sigma\Big(\llbracket\text{turn is left}\rrbracket \cdot \mathbf{u}_{n(w,j)}^\top \mathbf{v}_c\Big)$$

where $\llbracket x \rrbracket$ is 1 if true and -1 if false.

This reduces complexity to $O(\log_2 |\mathcal{V}|)$.

Insight: Hardware Efficiency: NEG vs Hierarchical Softmax

While Hierarchical Softmax is theoretically elegant, Negative Sampling is often preferred in modern Deep Learning frameworks. Why? Hierarchical Softmax requires traversing a tree, which involves random memory access patterns (pointer chasing). This is inefficient on GPUs, which excel at dense matrix multiplications. Negative Sampling can be implemented as a large matrix multiplication of the center vectors against the noise vectors, utilizing GPU parallelism much more effectively.

Things to Ponder

1. **Noise Sampling:** Why raise the frequency to the power of 0.75? What happens if we sample uniformly? What happens if we sample exactly by frequency (power 1.0)?
2. **Sum of Probabilities:** In Hierarchical Softmax, prove that $\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1$. (Hint: Start from the root node where $\sigma(x) + \sigma(-x) = 1$ and propagate down).
3. **CBOW Implementation:** How would you adapt Negative Sampling for the CBOW model? Note that in CBOW, the input is an average of vectors rather than a single vector.

5 GloVe: Global Vectors for Word Representation

Word2vec relies on local context windows. It learns incrementally and does not explicitly store global statistics. In contrast, Matrix Factorization methods (like LSA) use global statistics but perform poorly on analogy tasks. **GloVe** (Pennington et al., 2014) combines these approaches.

5.1 Global Corpus Statistics

Let X be the word-word co-occurrence matrix, where X_{ij} denotes the number of times word j appears in the context of word i. Let $X_i = \sum_k X_{ik}$ be the total number of times any word appears in the context of i. The probability of word j appearing in the context of i is $P_{ij} = P(j \mid i) = X_{ij}/X_i$.

5.2 The Ratio of Probabilities

The core insight of GloVe is that the *ratio* of co-occurrence probabilities is better at distinguishing relevant words than raw probabilities. Consider words $i =$ "ice" and $j =$ "steam".

- For $k =$ "solid": $P(k|ice)$ is high, $P(k|steam)$ is low. Ratio is large.
- For $k =$ "gas": $P(k|ice)$ is low, $P(k|steam)$ is high. Ratio is small.
- For $k =$ "water": Related to both. Ratio is near 1.
- For $k =$ "fashion": Unrelated to both. Ratio is near 1.

5.2.1 Derivation of the Linear Constraint

We seek a function F that encodes the ratio of probabilities using vector differences:

$$F((\mathbf{w}_i - \mathbf{w}_j)^\top \tilde{\mathbf{w}}_k) = \frac{P_{ik}}{P_{jk}}$$

To satisfy the homomorphism property $F(a+b) = F(a)F(b)$ (necessary to preserve algebraic structure between vectors), we assume $F(x) = \exp(x)$. This gives:

$$\exp(\mathbf{w}_i^\top \tilde{\mathbf{w}}_k - \mathbf{w}_j^\top \tilde{\mathbf{w}}_k) = \frac{P_{ik}}{P_{jk}}$$

Simplifying for a single pair i, k:

$$\mathbf{w}_i^\top \tilde{\mathbf{w}}_k = \log P_{ik} = \log X_{ik} - \log X_i$$

Since the term $\log X_i$ is independent of k, we absorb it into a bias term b_i. We add another bias $\tilde{b}_k$ for symmetry.

$$\mathbf{w}_i^\top \tilde{\mathbf{w}}_k + b_i + \tilde{b}_k = \log X_{ik}$$

5.3 The GloVe Objective

To satisfy the linear relationship above for all pairs, GloVe minimizes a weighted least squares regression loss:

$$J = \sum_{i,j=1}^{N} f(X_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log X_{ij} \right)^2 \tag{10}$$

Here:

- $\mathbf{v}_i, \mathbf{u}_j$ are center and context vectors.
- b_i, c_j are bias terms that absorb the $\log(X_i)$ term.
- $\log X_{ij}$ is the target value.
- $f(X_{ij})$ is a weighting function.

5.3.1 Weighting Function

Rare co-occurrences are noisy and carry less information. Frequent co-occurrences are robust but shouldn't dominate. The function $f(x)$ is chosen as:

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

Commonly, $x_{max} = 100$ and $\alpha = 0.75$. This function is continuous, non-decreasing, and crucially $f(0) = 0$ (so we don't train on pairs that never co-occur).

Insight: Symmetry in GloVe

Note that the loss function is symmetric with respect to i and j (if X is symmetric). This implies that center and context vectors are mathematically interchangeable. In practice, GloVe trains two sets of vectors $\mathbf{W}$ and $\tilde{\mathbf{W}}$ and outputs their sum $\mathbf{W} + \tilde{\mathbf{W}}$ as the final embedding. This simple ensemble trick typically reduces overfitting and improves performance.

Things to Ponder

1. **Distance weighting:** The standard definition of co-occurrence X_{ij} counts occurrences in a window. Can we refine this? How would weighting co-occurrences by distance (e.g., $1/d$) affect the semantic resolution?
2. **Bias Equivalence:** Are the bias terms b_i and c_j mathematically equivalent? If the matrix X is symmetric, should b_i equal c_i?

6 Subword Embedding

Standard word embeddings (word2vec, GloVe, fastText) treat words as atomic units. They assign a distinct vector to "dog", "dogs", and "dogcatcher". They fail to capture the internal morphological structure (e.g., the root "dog"). Furthermore, they cannot generate vectors for Out-Of-Vocabulary (OOV) words.

6.1 fastText

Proposed by Bojanowski et al. (2017), **fastText** extends the Skip-Gram model to include subword information.

6.1.1 Character n-grams

A word is represented as a bag of character n-grams. We add special boundary symbols $<$ and $>$ and extract n-grams. Example: "where" with $n = 3$:

$$\mathcal{G}_{where} = \{\text{¡wh, whe, her, ere, re¿}\} \cup \{\text{¡where¿}\}$$

The original word is always included as a special sequence.

6.1.2 Vector Representation

Each n-gram g is assigned a vector $\mathbf{z}_g$. The word vector $\mathbf{v}_w$ is the sum of its n-gram vectors:

$$\mathbf{v}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g$$

The rest of the training (Skip-Gram with Negative Sampling) proceeds as usual, but replacing the static center vector with this sum.

6.1.3 Benefits

- **Morphology:** "eating" and "eats" share the n-gram "eat", so their vectors share parameters.
- **OOV Words:** If the model encounters a new word "orthodontist", it can construct a vector by summing the vectors for "orth", "thod", "odon", etc.

6.2 Byte Pair Encoding (BPE)

While fastText handles subwords via summation, modern Transformers (like BERT and GPT) use subword tokenization to create a fixed vocabulary of subword units. **Byte Pair Encoding (BPE)** is a compression algorithm adapted for this purpose.

6.2.1 The Algorithm

1. **Initialization:** Vocabulary includes all unique characters in the corpus.
2. **Statistics:** Count frequency of all adjacent symbol pairs.
3. **Merge:** Select the most frequent pair (e.g., 'e', 'r') and merge them into a new symbol ('er').
4. **Repeat:** Iterate until a desired vocabulary size is reached.

This allows the model to represent common words as single tokens (e.g., "apple") while splitting rare words into smaller known units (e.g., "unfriendly" → "un", "friend", "ly").

Insight: BPE vs. WordPiece

BPE is a greedy algorithm based on frequency. BERT uses a variant called **WordPiece**. WordPiece also iteratively merges symbols, but instead of choosing the most frequent pair, it chooses the pair that maximizes the likelihood of the training data (minimizes perplexity) when added to the language model. Conceptually similar, but the selection criteria differ slightly.

Things to Ponder

1. **Memory Footprint:** There are 3×10^8 possible 6-grams. How does fastText manage memory? (Hint: Review "Hashing Trick" in the fastText paper).
2. **CBOW Variant:** How would you design a subword model based on CBOW rather than Skip-Gram?
3. **Merging Cost:** To reach a vocabulary of size M starting from N characters, exactly how many merge operations are required?

7 Word Similarity and Analogy

Once trained, word vectors exhibit remarkable algebraic properties.

7.1 Cosine Similarity

Since the magnitude of vectors can vary (due to word frequency), we use Cosine Similarity to measure semantic closeness:

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}$$

7.2 Analogy Task

The famous "King - Man + Woman = Queen" relationship allows us to solve analogy tasks of the form "a is to b as c is to ?" by finding vector $\mathbf{d}$ such that:

$$\mathbf{d} \approx \mathbf{c} + (\mathbf{b} - \mathbf{a})$$

We search for the word w in the vocabulary that maximizes:

$$\text{argmax}_{w \in \mathcal{V}} \cos(\mathbf{w}, \mathbf{v}_c + \mathbf{v}_b - \mathbf{v}_a)$$

Things to Ponder

1. **fastText on Analogies:** Does fastText perform better on syntactic analogies (run-running) or semantic analogies (King-Queen)? Why?
2. **Search Speed:** Finding the nearest neighbor requires comparing against every word in $|\mathcal{V}|$. How can we speed this up for very large vocabularies? (Hint: Approximate Nearest Neighbor / Faiss).

8 BERT: Bidirectional Encoder Representations from Transformers

Static embeddings (Word2vec, GloVe, fastText) are **context-independent**. The vector for "bank" is the same in "bank deposit" and "river bank". This is a major limitation.

BERT (Devlin et al., 2018) introduces **context-sensitive** embeddings. It pretrains a deep neural network (Transformer Encoder) so that the representation of a word depends on the entire sentence it appears in.

8.1 Architecture

BERT uses the Transformer **Encoder** stack (unlike GPT which uses the Decoder). This allows it to look at context from both left and right directions simultaneously.

- **BERT-Base:** 12 Layers, 768 Hidden size, 12 Heads (110M parameters).
- **BERT-Large:** 24 Layers, 1024 Hidden size, 16 Heads (340M parameters).

8.2 Input Representation

The input to BERT is a sequence of tokens. To handle sentence pairs (e.g., Question-Answering), the input format is:

$$[\text{CLS}], \text{Tok}_1, \ldots, \text{Tok}_N, [\text{SEP}], \text{Tok}_{N+1}, \ldots, \text{Tok}_M, [\text{SEP}]$$

The input embedding is the sum of three embeddings:

1. **Token Embeddings:** WordPiece embeddings.
2. **Segment Embeddings:** Learned vector E_A for first sentence, E_B for second.
3. **Position Embeddings:** Learned vectors indicating position $0, 1, 2 \ldots$ (BERT uses learnable absolute positions, not sinusoidal).

8.3 Pretraining Tasks

BERT is pretrained on two self-supervised tasks.

8.3.1 Masked Language Modeling (MLM)

Standard conditional language models (like GPT) can only look to the left (unidirectional) to prevent "seeing the future." BERT wants bidirectional context. To solve this, it masks tokens.

- Randomly select 15% of tokens.
- Predict the original token based on the context.

The 80-10-10 Rule: If we always replace the chosen token with '[MASK]', the model never sees '[MASK]' during fine-tuning. To mitigate this mismatch:

- 80% of the time: Replace with '[MASK]'.
- 10% of the time: Replace with a random word (forces model to check context).
- 10% of the time: Keep original word (biases model toward correct representation).

8.3.2 Next Sentence Prediction (NSP)

To understand relationships between sentences (crucial for QA and NLI), BERT is trained to predict if Sentence B effectively follows Sentence A.

- 50% of pairs are actual consecutive sentences (Label: IsNext).
- 50% are random sentence pairs (Label: NotNext).

The prediction is made using the embedding of the special ‘[CLS]‘ token.

Insight: The Cost of Bidirectionality

Because MLM only predicts 15% of the tokens per batch, whereas a standard Left-to-Right Language Model predicts 100% of the tokens (every next word), BERT requires significantly more pretraining steps to converge. However, the resulting representations are far richer for understanding tasks.

Things to Ponder

1. **Convergence Speed:** Why does a Masked Language Model require more steps to converge than a Left-to-Right model?
2. **Activation Functions:** BERT uses GELU (Gaussian Error Linear Unit) instead of ReLU. Research the difference. Why might GELU be better for deep transformers?
3. **NSP Necessity:** Later models like RoBERTa removed the Next Sentence Prediction task. Why might NSP be unnecessary or even harmful in some contexts?