

# Natural Language Processing Specialization

## Formula Sheet

Fady Morris Ebeid  
(2020)

## Chapter 1 Classification and Vector Spaces

### 1 Logistic Regression

corpus: a language resource consisting of a large and structured set of texts.

#### 1.1 Notation

$V$ : Vocabulary size, the number of unique words in the entire set of sentences.

$\theta$ : Parameter vector,  $\theta = [\theta_0, \theta_1, \dots, \theta_n]$

$m$ : Number of examples (sentences)

$P(\text{class})$ : Probability that a sentence is in a given class.  
class  $\in \{\text{pos}, \text{neg}\}$ .

$\text{freq}(w_i, \text{class})$ : Frequency of a word  $w_i$  in a specific class.

#### 1.2 Preprocessing

1. Eliminate handles and URLs.
2. Tokenize the string  $\mathbf{w} = [w_1, w_2, \dots, w_n]$ .
3. Remove stop words(and, is, are, at, has, for, a, ...) and punctuation (, . : ! " ' ).
4. Stemming: Convert every word to its stem.(use Porter Stemmer [Por80]).
5. Convert words to lowercase.

#### 1.3 Feature Extraction with Frequencies

$\mathbf{X}^{(m)}$ : Features vector of a sentence  $m$ . It is a row vector.

$$\mathbf{X}^{(m)} = \left[ \underbrace{1}_{\text{bias}}, \sum_w \text{freq}(w, \text{pos}), \sum_w \text{freq}(w, \text{neg}) \right]$$

Then all the examples  $m$  can be represented as the matrix  $\mathbf{X}$ :

$$\mathbf{X} = \begin{bmatrix} 1 & X_1^{(1)} & X_2^{(1)} \\ 1 & X_1^{(2)} & X_2^{(2)} \\ \vdots & \vdots & \vdots \\ 1 & X_1^{(m)} & X_2^{(m)} \end{bmatrix} \quad (1.1)$$

#### 1.4 Logistic Regression: Regression and Sigmoid

The *logits*  $z^{(i)}$  for an example  $i$  can be calculated as:

$$z^{(i)} = \theta^\top \mathbf{x}^{(i)} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (1.2)$$

The hypothesis function  $h$  (sigmoid function  $\sigma$ ):

$$h(\mathbf{x}^{(i)}, \theta) = h(z^{(i)}) = \sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}} \quad (1.3)$$

Note: All the  $h$  values are between 0 and 1.

#### 1.5 Cost Function

The loss function for a single training example is:

$$\mathcal{L}(\theta) = - \left[ y^{(i)} \log(h(z^{(i)})) + (1 - y^{(i)}) \log(1 - h(z^{(i)})) \right]$$

The cost function used for logistic regression is the average of the log loss across all training examples:

$$\mathcal{J}(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(h(z^{(i)})) + (1 - y^{(i)}) \log(1 - h(z^{(i)})) \right] \quad (1.4)$$

Where:

- $m$  is the number of training examples.
- $y^{(i)}$ : is the actual label of the  $i^{th}$  training example.
- $h(z^{(i)})$  is the model prediction for the  $i^{th}$  training example.

#### 1.6 Gradient Descent

The gradient of the cost function  $\mathcal{J}$  with respect to one of the weights  $\theta_j$  is

$$\nabla_{\theta_j} \mathcal{J}(\theta) = \frac{1}{m} \sum_{i=1}^m (h(z^{(i)}) - y^{(i)}) x_j \quad (1.5)$$

To update the weight  $\theta_j$  using gradient descent:

$$\theta_j := \theta_j - \alpha \nabla_{\theta_j} \mathcal{J}(\theta) \quad (1.6)$$

Where  $\alpha$  is the *learning rate*, a value to control how big a single update will be.

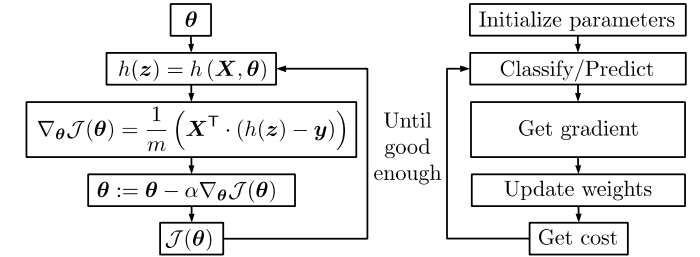
#### 1.7 Vectorized Implementation

Putting all the examples in a matrix  $\mathbf{X}$  (Equation 1.1), then the previous equations become:

$$\begin{aligned} \mathbf{z} &\stackrel{(1.2)}{=} \mathbf{X} \theta \\ h(\mathbf{X}, \theta) &\stackrel{(1.3)}{=} h(\mathbf{z}) = \sigma(\mathbf{z}) = \frac{1}{1 + e^{-\mathbf{z}}} \\ \mathcal{J}(\theta) &\stackrel{(1.4)}{=} -\frac{1}{m} \left[ \mathbf{y}^\top \cdot \log(h(\mathbf{z})) + (1 - \mathbf{y})^\top \cdot \log(1 - h(\mathbf{z})) \right] \\ \nabla_{\theta} \mathcal{J}(\theta) &\stackrel{(1.5)}{=} \frac{1}{m} \left( \mathbf{X}^\top \cdot (h(\mathbf{z}) - \mathbf{y}) \right) \end{aligned}$$

$$\theta \stackrel{(1.6)}{=} \theta - \alpha \nabla_{\theta} \mathcal{J}(\theta)$$

Figure 1.1: Training Logistic Regression



#### 1.8 Testing Logistic Regression

$m_{(\text{val})}$ : Total number of examples (sentences) in validation set.

$y_i^{(\text{val})}$ : Ground truth label for an example  $i \in \{1, \dots, m_{(\text{val})}\}$  in the validation set. 1 for positive sentiment, 0 for negative sentiment.

$\hat{y}_i^{(\text{val})}$ : Predicted label (sentiment) for the  $i^{th}$  example in the validation set.

1. Perform testing on unseen validation data  $\mathbf{X}^{(\text{val})}, \mathbf{y}^{(\text{val})}$  using trained weights  $\theta$ .
2. Calculate  $h(\mathbf{X}^{(\text{val})}, \theta) = h(\mathbf{z})$
3. Predict  $\hat{y}_i^{(\text{val})}$  for each example as follows

$$\hat{y}_i^{(\text{val})} = \begin{cases} 1, & \text{If } h(\mathbf{z})_i \geq 0.5 \\ 0, & \text{otherwise} \end{cases}$$

4. Calculate the *accuracy score* for all examples in the validation set:

$$\begin{aligned} \text{accuracy} &= \frac{1}{m_{(\text{val})}} \sum_{i=1}^{m_{(\text{val})}} (\hat{y}_i^{(\text{val})} == y_i^{(\text{val})}) \\ &= 1 - \underbrace{\frac{1}{m_{(\text{val})}} \sum_{i=1}^{m_{(\text{val})}} |\hat{y}_i^{(\text{val})} - y_i^{(\text{val})}|}_{\text{error}} \end{aligned}$$

## 2 Naïve Bayes

### 2.1 Conditional Probability and Bayes Rule

Conditional Probability:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad (1.7)$$

Bayes Rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad (1.8)$$

## 2.2 Naïve Bayes Assumptions

- Independence of events  $P(A \cap B) = P(A)P(B)$ . It assumes that the words in a piece of text are independent of one another, which is not true in reality, but it works well.
- Relative frequency in corpus: It relies on the distribution of the training data sets. A good data set will contain the same proportion of positive and negative tweets as a random sample would. However, most of available annotated corpora are artificially balanced. In reality positive sentences occur more frequently than negative.

## 2.3 Notation

$\text{class} \in \{\text{pos}, \text{neg}\}$ .

$w$ : A unique word in the vocabulary.

$\text{ratio}(w_i)$ : Ratio of the probability that the word  $w_i$  being positive to being negative.

$N_{\text{class}}$ : The total number of words in a class.

$N$ : total number of words in the corpus.

## 2.4 Naïve Bayes Introduction

$$N_{\text{class}} = \sum_{i=1}^V \text{freq}(w_i, \text{class}) \quad (1.9)$$

$$P(\text{class}) = \frac{N_{\text{class}}}{N} \quad (1.10)$$

$$N = N_{\text{pos}} + N_{\text{neg}}$$

$$P(\text{neg}) = 1 - P(\text{pos})$$

$$P(w|\text{class}) = \frac{\text{freq}(w, \text{class})}{N_{\text{class}}} \approx \frac{\text{freq}(w, \text{class}) + 1}{N_{\text{class}} + V} \quad (\text{Laplacian smoothing}) \quad (1.11)$$

$$\sum_{i=1}^V P(w_i|\text{class}) = 1$$

The Naive Bayes inference condition rule for binary classification (of a sentence):

$$\prod_{i=1}^n \frac{P(w_i|\text{pos})}{P(w_i|\text{neg})}$$

Where  $n$ : number of words in a sentence.

### Likelihood

$$\text{ratio}(w) = \frac{P(w|\text{pos})}{P(w|\text{neg})} \stackrel{(1.11)}{\approx} \frac{P(w|\text{pos}) + 1}{P(w|\text{neg}) + 1} \quad (\text{Laplacian smoothing}) \quad (1.12)$$

$$\text{ratio}(w) = \begin{cases} 0 : 1 & \text{Negative sentiment.} \\ 1 & \text{Neutral Sentiment.} \\ 1 : \infty & \text{Positive sentiment.} \end{cases}$$

$$P(\text{class}|w_i) \stackrel{(1.8)}{=} \frac{P(\text{class})P(w_i|\text{class})}{P(w_i)} \quad (1.13)$$

$$\frac{P(\text{pos}|w_i)}{P(\text{neg}|w_i)} \stackrel{(1.13)}{=} \frac{P(\text{pos})P(w_i|\text{pos})}{P(\text{neg})P(w_i|\text{neg})} \quad (1.14)$$

$$\frac{P(\text{pos}|\text{sentence})}{P(\text{neg}|\text{sentence})} \stackrel{(1.14)}{=} \frac{P(\text{pos})}{P(\text{neg})} \prod_{i=1}^n \frac{P(\text{pos})P(w_i|\text{pos})}{P(\text{neg})P(w_i|\text{neg})} \quad (1.15)$$

$$= \frac{P(\text{pos})}{P(\text{neg})} \prod_{i=1}^n \text{ratio}(w_i) \stackrel{(1.12)}{\approx} \underbrace{\frac{P(\text{pos})}{P(\text{neg})}}_{\text{prior ratio}} \prod_{i=1}^n \underbrace{\frac{P(w_i|\text{pos}) + 1}{P(w_i|\text{neg}) + 1}}_{\text{likelihood}} \quad (1.16)$$

Where  $n$ : number of words in a sentence.

### Log Likelihood Score

Carrying repeated multiplications in 1.16 can result in numerical underflow. This problem is solved by taking log of both sides of the equation to calculate the *log likelihood score* of a sentence using the following equation:

$$\begin{aligned} \log \frac{P(\text{pos}|\text{sentence})}{P(\text{neg}|\text{sentence})} &\stackrel{(1.16)}{=} \log \left[ \frac{P(\text{pos})}{P(\text{neg})} \prod_{i=1}^n \text{ratio}(w_i) \right] \\ &= \log \frac{P(\text{pos})}{P(\text{neg})} + \sum_{i=1}^n \log(\text{ratio}(w_i)) \\ &= \underbrace{\log \frac{P(\text{pos})}{P(\text{neg})}}_{\text{logprior}} + \underbrace{\sum_{i=1}^n \log \frac{P(w_i|\text{pos}) + 1}{P(w_i|\text{neg}) + 1}}_{\text{log likelihood}} \\ &= \log \frac{P(\text{pos})}{P(\text{neg})} + \underbrace{\sum_{i=1}^n \lambda(w_i)}_{\text{log likelihood}} \quad (1.17) \end{aligned}$$

Where

$$\begin{aligned} \lambda(w_i) &= \log(\text{ratio}(w_i)) \\ &\stackrel{(1.12)}{=} \log \frac{P(w_i|\text{pos}) + 1}{P(w_i|\text{neg}) + 1} \quad (1.18) \\ \lambda(w_i) &\begin{cases} < 0 & \text{Negative word.} \\ = 0 & \text{Neutral word.} \\ > 0 & \text{Positive word.} \end{cases} \end{aligned}$$

If *log likelihood score* is  $> 0$ , the sentence is positive. If it is  $< 0$ , the sentence is negative.

## 2.5 Training Naïve Bayes

- Collect and annotate corpus.

Preprocess text:

- Lowercase.
- Remove punctuation, URLs, names.
- Remove stop words.
- Stemming [Por80].
- Tokenize sentences  $\mathbf{w} = [w_1, w_2, \dots, w_n]$

- Word count.

- Compute  $\text{freq}(w, \text{class})$  for every word in the vocabulary.
- Compute  $N_{\text{class}}$  [equation 1.9]

- Compute conditional probabilities  $P(w|\text{pos})$ ,  $P(w|\text{neg})$  [equation 1.11]

- Calculate the *lambda score* ( $\lambda(w)$ ) for each word [equation 1.18]

- Get the *logprior*:

$$\log \frac{P(\text{pos})}{P(\text{neg})} \stackrel{(1.10)}{=} \log \frac{N_{\text{pos}}}{N_{\text{neg}}}$$

If you are working with a balanced dataset ( $N_{\text{pos}} = N_{\text{neg}}$ ), then  $\text{logprior} = 0$

## 2.6 Testing Naïve Bayes

$m_{(\text{val})}$ : Total number of examples (sentences) in validation set.

$y_i^{(\text{val})}$ : Ground truth label for an example  $i \in \{1, \dots, m_{(\text{val})}\}$  in the validation set. 1 for positive sentiment, 0 for negative sentiment.

$\hat{y}_i^{(\text{val})}$ : Predicted label (sentiment) for the  $i^{\text{th}}$  example in the validation set.

- Perform testing on unseen validation data  $\mathbf{X}^{(\text{val})}$ ,  $\mathbf{y}^{(\text{val})}$
- first, calculate *log likelihood score* for each sentence in the examples [equation 1.17]
- Predict  $\hat{y}_i^{(\text{val})}$  for each example as follows

$$\hat{y}_i^{(\text{val})} = \begin{cases} 1, & \text{If } \log \text{likelihood score} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Calculate the *accuracy score* for all examples in the validation set:

$$\begin{aligned} \text{accuracy} &= \frac{1}{m_{(\text{val})}} \sum_{i=1}^{m_{(\text{val})}} \left( \hat{y}_i^{(\text{val})} == y_i^{(\text{val})} \right) \\ &= 1 - \underbrace{\frac{1}{m_{(\text{val})}} \sum_{i=1}^{m_{(\text{val})}} \left| \hat{y}_i^{(\text{val})} - y_i^{(\text{val})} \right|}_{\text{error}} \end{aligned}$$

For a word not in the corpus, it is treated as neutral ( $\lambda(w) = 0$ )

### 3 Vector Space Models

- Represent words and documents as *vectors*.
- Representation that *captures* relative *meaning*.

#### 3.1 Word by Word and Word by Doc.

##### Word by Word Design (W/W)

Counts the *co-occurrence* of two different words, which is the *number of times* they occur together within a certain distance  $k$ . With *word by word* design you get a representation matrix with  $n \times n$  entries, where  $n$  equals to vocabulary size  $V$ .

##### Word by Document Design (W/D)

Counts the *Number of times a word* occurs within a certain category. Represented by a matrix with  $n \times c$  entries, where  $c$  is the number of categories.

#### 3.2 Euclidean Distance

The *euclidean distance* between two  $n$ -dimensional vectors:

$$\begin{aligned} d(\vec{v}, \vec{w}) &= d(\vec{w}, \vec{v}) \\ &= \|\vec{v} - \vec{w}\| \\ &= \sqrt{(v_1 - w_1)^2 + (v_2 - w_2)^2 + \dots + (v_n - w_n)^2} \\ &= \sqrt{\sum_{i=1}^n (v_i - w_i)^2} \end{aligned}$$

Where

- $n$  is the number of elements in the vector.
- The more similar the words, the more likely the Euclidean distance will be close to 0.

#### 3.3 Cosine Similarity

The main advantage of this metric over the *euclidean distance* is that it isn't biased by the size difference between the representations.

Vector norm:

$$\|\vec{v}\| = \sqrt{\sum_{i=1}^n v_i^2}$$

Dot product:

$$\vec{v} \cdot \vec{w} = \sum_{i=1}^n v_i \cdot w_i$$

Cosine similarity:

$$\cos(\theta) = \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\| \|\vec{w}\|}$$

*Cosine similarity* gives values between -1 and 1.

$$\cos(\theta) = \begin{cases} 1 & \text{Parallel and in the same direction.} \\ 0 & \text{Orthogonal(perpendicular).} \\ -1 & \text{Point exactly in opposite directions.} \end{cases}$$

- Numbers in the range  $[0, 1]$  indicate a *similarity score*.
- Numbers in the range  $[-1, 0]$  indicate a *dissimilarity score*.

#### 3.4 Manipulating Words in Vector Spaces

[Mik+13a]

#### 3.5 Visualization and PCA

PCA is used to visualize the embeddings on a  $k$ -dimensional subspace of the original  $n$ -dimensional subspace of the word embeddings.

*Eigenvector*: Uncorrelated features for your data.

*Eigenvalue*: The amount of information retained by each feature.

Perform PCA on a data matrix  $\mathbf{X} = [\mathbf{x}_1 | \mathbf{x}_2 | \dots | \mathbf{x}_n]^\top \in \mathbb{R}^{m \times n}$ , where  $m$  is the number of examples,  $n$  is the dimension (length) of a word embedding.

Steps of PCA:

1. Mean normalize data and obtain the normalized data matrix  $\bar{\mathbf{X}}$

$$\mu = \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i, \quad \sigma = \sqrt{\frac{1}{m} \sum_{i=1}^m \mathbf{x}_i^2 - \mu^2}$$

$$\bar{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu}{\sigma}$$

$$x_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}}$$

$$\bar{\mathbf{X}} = [\bar{\mathbf{x}}_1 | \bar{\mathbf{x}}_2 | \dots | \bar{\mathbf{x}}_n]^\top$$

2. Get the  $n \times n$  *covariance matrix*  $\Sigma$

$$\Sigma = \frac{1}{m} \bar{\mathbf{X}}^\top \bar{\mathbf{X}}$$

3. Perform a *singular value decomposition* to get the *eigenvectors*  $\mathbf{U} \in \mathbb{R}^{n \times n}$  and *eigenvalues* diagonal matrix  $\mathbf{S} \in \mathbb{R}^{n \times n}$ .

$$\mathbf{U}, \mathbf{S} = \text{SVD}(\Sigma)$$

4. Project data onto the  $k$ -dimensional principal subspace: Multiply your normalized data by the first  $k$  *eigenvectors* associated with the  $k$  largest *eigenvalues* to compute the projection  $\mathbf{X}' \in \mathbb{R}^{m \times k}$ .

$$\mathbf{B} = (\mathbf{U}_{ij})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq k}}$$

$$\mathbf{X}' = \bar{\mathbf{X}} \mathbf{B}$$

The percentage of *retained variance* can be calculated from

$$\frac{\sum_{i=0}^1 S_{ii}}{\sum_{j=0}^d S_{jj}}$$

### 4 Machine Translation and Document Search

#### 4.1 Machine Translation

##### Transforming Word Vectors

Assume that we have a subset of a *source language* dataset of word embeddings  $\mathbf{X} = [\mathbf{x}_1 | \mathbf{x}_2 | \dots | \mathbf{x}_m]^\top$  and a translation subset of *destination language* dataset  $\mathbf{Y} = [\mathbf{y}_1 | \mathbf{y}_2 | \dots | \mathbf{y}_m]^\top$ . We want to find a transformation matrix  $\mathbf{R}$  such that:

$$\mathbf{X} \mathbf{R} \approx \mathbf{Y}$$

Cost function:

$$\mathcal{J} = \frac{1}{m} \|\mathbf{X} \mathbf{R} - \mathbf{Y}\|_F^2$$

where:

- $m$  is the number of examples.
- $\|\mathbf{A}\|_F$  is the *Frobenius norm*,

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}$$

- The reason for taking the square is that it's easier to compute the gradient of the squared Frobenius.

The gradient of the cost function with respect to the *transformation matrix*:

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial \mathbf{R}} &= \frac{\partial}{\partial \mathbf{R}} \frac{1}{m} \|\mathbf{X} \mathbf{R} - \mathbf{Y}\|_F^2 \\ &= \frac{2}{m} (\mathbf{X} \mathbf{R} - \mathbf{Y})^\top \mathbf{X} \\ &= \frac{2}{m} \mathbf{X}^\top (\mathbf{X} \mathbf{R} - \mathbf{Y}) \end{aligned}$$

Then we use *gradient descent* to optimize the transformation matrix:

$$\mathbf{R} := \mathbf{R} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{R}}$$

The predictions can be obtained using the trained  $\mathbf{R}$  matrix:

$$\hat{\mathbf{Y}} = \mathbf{X} \mathbf{R}$$

The translation of a word  $i$  can be found using  $k$ -nearest neighbor of  $\hat{\mathbf{y}}_i$  from  $\mathbf{Y}$  with  $k = 1$ .

#### 4.2 Document Search

##### Document Representation

1. **Bag-of-words (BOW) document models**

Text documents are sequences of words. The ordering of words makes a difference.

2. **Document embeddings**

A document can be represented as a *document vector* by summing up the word embeddings of every word in the document. If we don't know the embedding of a word, we can ignore that word.

## Locality Sensitive Hashing

A more efficient version of  $k$ -nearest neighbors can be implemented using locality sensitive hashing. Instead of searching the vector space we can only search in a subspace for the nearest neighboring vectors.

Assume we have a plane(hyperplane)  $\pi$  that divides the vector space that has a normal vector  $\mathbf{p}$ , then for any point with a position vector  $\mathbf{v}$ :

$$\mathbf{p} \cdot \mathbf{v} \begin{cases} > 0, & \text{the point is above the plane.} \\ = 0, & \text{the point is on the plane.} \\ < 0, & \text{the point is below the plane.} \end{cases}$$

## Multiplanes Hash Functions

- Multiplanes hash functions* are based on the idea of numbering every single region that is formed by the intersection of  $n$  planes.
- We can divide the vector space into  $2^n$  parts(*hash buckets*).

The hash value for a position of a vector  $\mathbf{v}$  with respect to a plane  $\mathbf{p}_i$  is:

$$h_i = \begin{cases} 1, & \text{If } \text{sign}(\mathbf{p}_i \cdot \mathbf{v}) \geq 0. \\ 0, & \text{If } \text{sign}(\mathbf{p}_i \cdot \mathbf{v}) < 0. \end{cases}$$

Where  $i = \{1, \dots, n\}$

The combined hash bucket number for a vector (for all planes):

$$\text{hash} = \sum_{i=1}^n 2^{i-1} \times h_i$$

# Chapter 2

## Probabilistic Models

### 1 Autocorrect and Minimum Edit Distance

#### 1.1 Autocorrect

Reference: [Nor07]

How it works

- Identify a misspelled word.  
Words not in the dictionary are misspelled words.
- Find strings  $n$  edit distance away.  
*Edit*: an operation performed on a string to change it.  
Examples (for a string with  $n$  letters):

Operation	Description	Output Count
Insert	Add a letter	$26(n+1)$
Delete	Remove a letter	$n$
Replace	Change 1 letter to another	$25n$
Switch	Swap 2 adjacent letters	$n-1$

- Filter candidates.

Given a *vocabulary*, filter the *edit* list for *candidate words* found in the vocabulary.

- Calculate word probabilities.

$$P(w) = \frac{\text{count}(w)}{M}$$

Where:

- $P(w)$ : Probability of a word.
- $\text{count}(w)$ : Number of times the word appears.
- $M$ : Total number of words in the *corpus*.

Then select the word with the highest probability as your *autocorrect* replacement.

#### Algorithm 1: Autocorrect

```

1 def autocorrect(word, n):
    Data:
    probs: a dictionary that maps each word to its
           probability in the corpus.

           
$$\text{probs}[w \rightarrow P(w)](x) = \begin{cases} P(w) = \frac{\text{count}(w)}{M} & \text{for } x = w \\ 0 & \text{otherwise} \end{cases}$$


    vocab: a set containing all the vocabulary.
    Result: n-best: a set of tuples with the most
             probable n corrected words and their
             probabilities.

2   suggestions =  $\phi$ 
3   n-best =  $\phi$ 
4   if word  $\in$  vocab:
5       | suggestions = suggestions  $\cup$  {word}
6   else:
7       one-edit-set = one-edit-distance(word)  $\cap$  vocab
8       if one-edit-set  $\neq \phi$ :
9           | suggestions = suggestions  $\cup$  one-edit-set
10      else:
11          two-edit-set = two-edit-distance(word)  $\cap$  vocab
12          if two-edit-set  $\neq \phi$ :
13              | suggestions = suggestions  $\cup$  two-edit-set
14          else:
15              | suggestions = suggestions  $\cup$  {word}

16  best-words[ $w \rightarrow \text{probs}(w)$ ](x) =
    {  $x = w | w \in \text{suggestions}$  }
17  n-best = {The set of top n words from best-words
             sorted by probabilities}

```

### 1.2 Minimum Edit Distance

Reference: [Jur12]

*Minimum edit distance* is the sum of costs of edits needed to transform one string into the other. It evaluates the similarity between two strings.

It is used in spelling correction, document similarity, machine translation, DNA sequencing and more.

Edits (operations) are:

Operation	Description	Cost
Insert	Add a letter	1
Delete	Remove a letter	1
Replace	Change 1 letter to another	2

### Minimum Edit Distance Algorithm

Minimum edit distance can be calculated using *dynamic programming*. It breaks a problem down into subproblems which can be combined to form the final solution. To do this efficiently, we will use a table (see Figure 2.1) to maintain the previously computed substrings and use those to calculate larger substrings.

Initialization:

$$D[0, 0] = 0 \quad (2.1)$$

$$D[i, 0] = D[i - 1, 0] + \text{del\_cost}(\text{source}[i]) \quad (2.2)$$

$$D[0, j] = D[0, j - 1] + \text{ins\_cost}(\text{target}[j]) \quad (2.3)$$

Per cell operations:

$$D[i, j] =$$

$$\min \begin{cases} D[i - 1, j] & + \text{del\_cost} \\ D[i, j - 1] & + \text{ins\_cost} \\ D[i - 1, j - 1] & + \begin{cases} \text{rep\_cost}; & \text{if } \text{source}[i] \neq \text{target}[j] \\ 0; & \text{if } \text{source}[i] = \text{target}[j] \end{cases} \end{cases} \quad (2.4)$$

$$\text{Minimum edit distance} = D[m, n] \quad (2.5)$$

	$j$	0	1	$\dots$	$n$
$i$		#	$T_0$	$\dots$	$T_{n-1}$
0	#	$D[0, 0]$	$D[0, 1]$	$\dots$	$D[0, n]$
1	$S_0$	$D[1, 0]$	$D[1, 1]$	$\dots$	$D[1, n]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$D[2, n]$
$m$	$S_{m-1}$	$D[m, 0]$	$D[m, 1]$	$\dots$	$D[m, n]$

Figure 2.1: Minimum Edit Distance Table

Example:

source  $\rightarrow$  target  
 “play”  $\rightarrow$  “stay”

$$D[i, j] = \text{source}[i] \rightarrow \text{target}[j]$$

$$D[2, 3] = \text{“pl”} \rightarrow \text{“sta”}$$

$$D[0, 0] = \# \rightarrow \#$$

$$D[m, n] = \text{source} \rightarrow \text{target}$$

where #: empty string.

$j$		0	1	2	3	4
$i$		#	s	t	a	y
0	#	0	1	2	3	4
1	p	1	2	3	4	5
2	l	2	3	4	5	6
3	a	3	4	5	4	5
4	y	4	5	6	5	4

Figure 2.2: Minimum Edit Distance of “play”  $\rightarrow$  “stay”

## 2 Part of Speech Tagging and Hidden Markov Models

Reference: [JM19, Chapter 8]

### 2.1 Part of Speech Tagging

Part of speech (POS) tagging is the process of assigning tags that represent categories of parts of speech to words of a corpus.

Applications of POS tagging:

- Identifying named entities.  
Eiffel tower is located in Paris.
- Co-reference resolution.  
The Eiffel tower is located in Paris, it is 324 meters high.
- Speech recognition.

lexical term	tag	example
noun	NN	something, nothing
verb	VB	learn, study
determiner	DT	the, a
w-adverb	WRB	why, where
...	...	...

No.	Tag	Description
1.	CC	Coordinating conjunction
2.	CD	Cardinal number
3.	DT	Determiner
4.	EX	Existential there
5.	FW	Foreign word
6.	IN	Preposition or subordinating conjunction
7.	JJ	Adjective
8.	JJR	Adjective, comparative
9.	JJS	Adjective, superlative
10.	LS	List item marker
11.	MD	Modal
12.	NN	Noun, singular or mass
13.	NNS	Noun, plural
14.	NNP	Proper noun, singular
15.	NNPS	Proper noun, plural
16.	PDT	Predeterminer
17.	POS	Possessive ending
18.	PRP	Personal pronoun
19.	PRP\$	Possessive pronoun
20.	RB	Adverb
21.	RBR	Adverb, comparative
22.	RBS	Adverb, superlative
23.	RP	Particle
24.	SYM	Symbol
25.	TO	to
26.	UH	Interjection
27.	VB	Verb, base form
28.	VBD	Verb, past tense
29.	VBG	Verb, gerund or present participle
30.	VBN	Verb, past participle
31.	VBP	Verb, non-3rd person singular present
32.	VBZ	Verb, 3rd person singular present
33.	WDT	Wh-determiner
34.	WP	Wh-pronoun
35.	WP\$	Possessive wh-pronoun
36.	WRB	Wh-adverb

Table 2.1: Part-of-Speech Tags

Source: [San90] and [LJP03]

### 2.2 Markov Chains

States:

$$S = \{s_1, s_2, \dots, s_N\}$$

Markov property: The probability of the next event only depends on the current event.

#### Initial Probability Vector

$$\pi = [\pi_1, \pi_2, \dots, \pi_N]$$

Example:

	NN	VB	O	
$\pi =$	$\pi$ (initial)	0.4	0.1	0.5

#### Algorithm 2: Minimum Edit Distance

```

1 def min-edit-distance(source, target):
    Data:
    source: a string corresponding to the string you are
    starting with.
    target: a string corresponding to the string you want
    to end with.
    Result:
    D: a matrix of size  $(m+1) \times (n+1)$  containing
    minimum edit distances (see Figure 2.1)
    med: the minimum edit distance required to convert
    the source string to the target

2  $D[0, 0] \stackrel{(2.1)}{=} 0$ 
3 for  $i \in \{1, 2, \dots, m\}$ :
4      $D[i, 0] \stackrel{(2.2)}{=} [i - 1, 0] + \text{del\_cost}$ 
5     for  $j \in \{1, 2, \dots, n\}$ :
6          $D[0, j] \stackrel{(2.3)}{=} [0, j - 1] + \text{ins\_cost}$ 
7         if  $\text{source}[i - 1] = \text{target}[j - 1]$ :
8             r_cost = 0
9         else:
10            r_cost = 2
11          $D[i, j] \stackrel{(2.4)}{=} \min \begin{cases} D[i - 1, j] & + \text{del\_cost} \\ D[i, j - 1] & + \text{ins\_cost} \\ D[i - 1, j - 1] & + \text{r\_cost} \end{cases}$ 
12 med  $\stackrel{(2.5)}{=} D[m, n]$ 

```

## The Transition Matrix

The *transition matrix* has a dimension  $(N \times N)$ .

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,N} \\ a_{2,1} & a_{2,2} & \dots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \dots & a_{N,N} \end{pmatrix} \quad (2.6)$$

$$= \begin{pmatrix} P(s_1|s_1) & P(s_2|s_1) & \dots & P(s_N|s_1) \\ P(s_1|s_2) & P(s_2|s_2) & \dots & P(s_N|s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(s_1|s_N) & P(s_2|s_N) & \dots & P(s_N|s_N) \end{pmatrix}$$

For all the outgoing transition probabilities:

$$\sum_{j=1}^N a_{ij} = 1$$

Example:

$$\mathbf{A} = \begin{array}{c|ccc} & \text{NN} & \text{VB} & \text{O} \\ \hline \text{NN (noun)} & 0.2 & 0.2 & 0.6 \\ \text{VB (verb)} & 0.4 & 0.3 & 0.3 \\ \text{O (other)} & 0.2 & 0.3 & 0.5 \end{array}$$

## 2.3 Hidden Markov Models

*Hidden states:* parts of speech. States that are hidden and not directly observable from the text data.

*Emission probabilities:* The probability of a visible observation when we are in a particular state. Emission probabilities describe the transition probabilities between the hidden states  $S = \{s_1, s_2, \dots, s_N\}$  (parts of speech) of hidden Markov model to the *observables* or *emissions* (words of corpus)  $O = \{o_1, o_2, \dots, o_V\}$ .

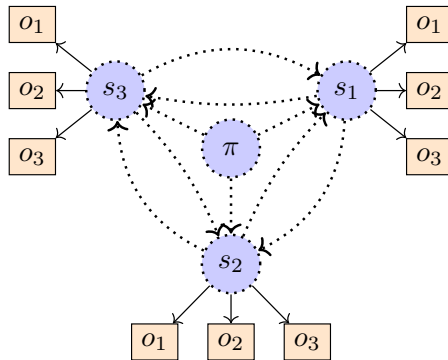


Figure 2.3: Hidden Markov Model

## Emission Matrix

$$\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1V} \\ b_{21} & b_{22} & \dots & b_{2V} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & \dots & b_{NV} \end{pmatrix} \quad (2.7)$$

$$= \begin{pmatrix} P(o_1|s_1) & P(o_2|s_1) & \dots & P(o_V|s_1) \\ P(o_1|s_2) & P(o_2|s_2) & \dots & P(o_V|s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(o_1|s_N) & P(o_2|s_N) & \dots & P(o_V|s_N) \end{pmatrix}$$

Emission matrix  $\mathbf{B}$  has a dimension  $(N \times V)$

Where  $N$  is the number hidden states (parts of speech tags),  $V$  is the number of observables (words in corpus).

$$\sum_{j=1}^M b_{ij} = 1$$

Example:

$$\mathbf{B} = \begin{array}{c|cccc} & \text{going} & \text{to} & \text{eat} & \dots \\ \hline \text{NN (noun)} & 0.5 & 0.1 & 0.02 & \\ \text{VB (verb)} & 0.3 & 0.1 & 0.5 & \\ \text{O (other)} & 0.3 & 0.5 & 0.68 & \end{array}$$

A word can have different parts of speech assigned depending on context in which they appear:

- “He lay <sup>NN</sup> on his back”
- “I will be <sup>RB</sup> back”

## 2.4 Calculating Transition Probabilities

1. Count the occurrences of tag pairs (the number of times each tag  $(t_i \in S)$  at time step  $i$  happened next to another tag  $(t_{i-1} \in S)$  at time step  $i - 1$ ).

$$C(t_i, t_{i-1})$$

2. Calculate probabilities: divide the counts by row sum to normalize the counts: The probability of a tag at position  $i$  given the tag at position  $i - 1$  becomes:

$$P(t_i|t_{i-1}) \stackrel{(2.6)}{=} a_{\text{rindex}(t_{i-1}), \text{cindex}(t_i)} = \frac{C(t_{i-1}, t_i)}{\sum_{j=1}^N C(t_{i-1}, t_j)} \quad (2.8)$$

Where

- $N$  is the total number of tags.

## Smoothing

To avoid division by zero and zero probabilities apply smoothing to the equation (2.8)

$$P(t_i|t_{i-1}) \stackrel{(2.8)}{=} \frac{C(t_{i-1}, t_i) + \epsilon}{\sum_{j=1}^N C(t_{i-1}, t_j) + N \cdot \epsilon} = \frac{C(t_{i-1}, t_i) + \epsilon}{C(t_{i-1}) + N \cdot \epsilon}$$

Where

- $C(t_{i-1})$  is the count of the previous POS tag occurrence in the corpus.
- $\epsilon$  is a smoothing parameter.

## 2.5 Calculating Emission Probabilities

Calculate the number of time a (tag, word) pair showed in the training set:

$$C(t_i, w_i)$$

Compute the probability of a word given its tag:

$$P(w_i|t_i) \stackrel{(2.7)}{=} b_{\text{rindex}(t_i), \text{cindex}(w_i)} = \frac{C(t_i, w_i) + \epsilon}{\sum_{j=1}^V C(t_i, w_j) + V \cdot \epsilon} = \frac{C(t_i, w_i) + \epsilon}{C(t_i) + V \cdot \epsilon}$$

Where

- $w$  is a word (observable) in the corpus.
- $C(t_i)$  is the number of times a tag has occurred in the corpus.
- $V$  is the number of words in the vocabulary.

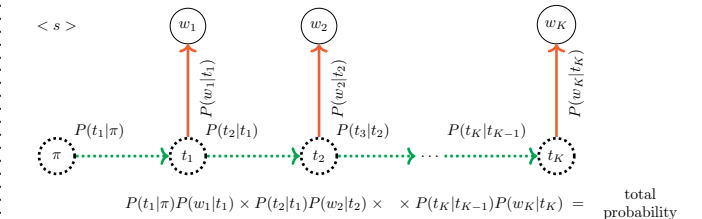
## 2.6 The Viterbi Algorithm

The *Viterbi algorithm* computes the most likely sequence of parts of speech tags for a given sentence (sequence of observations)

$\mathbf{w} = [w_1, w_2, \dots, w_K]$

The *joint probability* (combined probability) of the observing a word is calculated by multiplying the transition probability with the emission probability.

The *total probability* is calculated by multiplying all joint probabilities of steps of the sequence.





## Auxiliary Matrices

Given your transition and emission probabilities, you first populate and then use the auxiliary matrices  $\mathbf{C}$  and  $\mathbf{D}$ .

The matrix  $\mathbf{C} \in \mathbb{R}^{N \times K}$  holds the intermediate optimal probabilities.

The matrix  $\mathbf{D} \in \mathbb{R}^{N \times K}$  holds the indices of the visited states (or *best paths*, the different states you're traversing when finding the most likely sequence of parts of speech tags for the given sequence of words).

$$\mathbf{C} = \begin{array}{c|cccc} & w_1 & w_2 & \dots & w_K \\ \hline t_1 & & & & \\ \vdots & & & & \\ t_N & & & & \end{array}$$

$$\mathbf{D} = \begin{array}{c|cccc} & w_1 & w_2 & \dots & w_K \\ \hline t_1 & & & & \\ \vdots & & & & \\ t_N & & & & \end{array}$$

## Steps

### 1. Initialization:

The first column of each of the matrices  $\mathbf{C}$  and  $\mathbf{D}$  is populated.

$$\begin{aligned} c_{i,1} &= P(s_i|\pi)P(w_1|s_i) \\ &= \pi_i \cdot b_{i,\text{cindex}(w_1)} \end{aligned} \quad (2.9)$$

For matrix  $\mathbf{D}$ , in the first column, set all entries to zero, as there are no proceeding parts of speech tags we have traversed.

$$d_{i,1} = 0$$

### Vectorized:

$$\mathbf{c}_1 = \boldsymbol{\pi} \odot \mathbf{b}_{\text{cindex}(w_1)} \quad (2.10)$$

$$\mathbf{d}_1 = \vec{0}$$

### 2. Forward pass:

$$c_{i,j} = \max_k c_{k,j-1} \cdot a_{k,i} \cdot b_{i,\text{cindex}(w_j)} \quad (2.11)$$

$$d_{i,j} = \arg \max_k c_{k,j-1} \cdot a_{k,i} \cdot b_{i,\text{cindex}(w_j)} \quad (2.12)$$

where

- $a_{k,i}$  is the transition probability from the parts of speech tag  $t_k$  to the current tag  $t_i$ .
- $c_{k,j-1}$  represents the probability for the preceding path you've traversed.

### Vectorized:

$$\mathbf{c}_j = \max (\mathbf{c}_{j-1} \odot \mathbf{a}'_i) \cdot b_{i,\text{cindex}(w_j)} \quad (2.13)$$

$$\mathbf{d}_j = \arg \max (\mathbf{c}_{j-1} \odot \mathbf{a}'_i) \cdot b_{i,\text{cindex}(w_j)} \quad (2.14)$$

$$\text{where } \mathbf{A}' = [\mathbf{a}'_1 | \mathbf{a}'_2 | \dots | \mathbf{a}'_N]$$

### 3. Backward pass:

The probability at  $c_{i,K}$  is the probability of the most likely sequence of hidden states, generating the given sequence of words.

Get the index of  $c_{i,K}$

$$z_K = \underset{i}{\operatorname{argmax}} c_{i,K}$$

Use this index  $z_K$  to traverse backwards through the matrix  $\mathbf{D}$ , to reconstruct the sequence of parts of speech tags.

### Implementation note:

For numerical stability use log probabilities:

$$\log(c_{i,1}) \stackrel{(2.9)}{=} \log(\pi_i) + \log(b_{i,\text{cindex}(w_1)})$$

$$\log(c_{i,j}) \stackrel{(2.11)}{=} \max_k \log(c_{k,j-1}) + \log(a_{k,i}) + \log(b_{i,\text{cindex}(w_j)})$$

$$d_{i,j} \stackrel{(2.12)}{=} \arg \max_k \log(c_{k,j-1}) + \log(a_{k,i}) + \log(b_{i,\text{cindex}(w_j)})$$

### Vectorized:

$$\log(\mathbf{c}_1) \stackrel{(2.10)}{=} \log(\boldsymbol{\pi}) + \log(\mathbf{b}_{\text{cindex}(w_1)})$$

$$\log(\mathbf{c}_j) \stackrel{(2.13)}{=} \max [\log(\mathbf{c}_{j-1}) + \log(\mathbf{a}'_i)] + \log(b_{i,\text{cindex}(w_j)})$$

$$\mathbf{d}_j \stackrel{(2.14)}{=} \arg \max [\log(\mathbf{c}_{j-1}) + \log(\mathbf{a}'_i) + \log(b_{i,\text{cindex}(w_j)})]$$

## Algorithm 3: Viterbi Algorithm

### Data:

- $\mathcal{O} = \{o_1, o_2, \dots, o_V\}$ , the *observation space*
- $\mathcal{S} = \{s_1, s_2, \dots, s_N\}$ , the *state space*, where  $s_n$  is a tag.
- $\boldsymbol{\pi} = [\pi_1, \pi_2, \dots, \pi_N]$ . An array of *initial probabilities*, where  $\pi_i = P(x_1 = s_i)$
- $\mathbf{w} = [w_1, w_2, \dots, w_K]$ , A sequence of observations,  $w_i \in \mathcal{O}$
- $\mathbf{A} \in \mathbb{R}^{N \times N}$ . Transition matrix
- $\mathbf{B} \in \mathbb{R}^{N \times V}$ . Emission matrix, where  $B_{ij} = P(o_j | s_i)$

### Result:

$\mathbf{t} = [t_1, t_2, \dots, t_K]$ , the most likely hidden state sequence of parts of speech tags.

```

1 def VITERBI( $\mathcal{O}, \mathcal{S}, \boldsymbol{\pi}, \mathbf{y}, \mathbf{A}, \mathbf{B}$ ):
    /* Initialization */
2     for each state  $i = 1, 2, \dots, N$ :
3          $C_{i,1} \leftarrow \log(\pi_i) + \log(B_{i,\text{cindex}(w_1)})$ 
4          $D_{i,1} \leftarrow 0$ 
    /* Forward pass */
5     for each observation  $j = 2, 3, \dots, K$ :
6         for each state  $i = 1, 2, \dots, N$ :
7              $C_{ij} \leftarrow \max_k C_{k,j-1} + \log(A_{k,i}) + \log(B_{i,\text{cindex}(w_j)})$ 
8              $D_{ij} \leftarrow \arg \max_k C_{k,j-1} + \log(A_{k,i}) + \log(B_{i,\text{cindex}(w_j)})$ 
    /* Backward pass */
9      $z_K \leftarrow \arg \max_i C_{i,K}$ 
10     $t_K \leftarrow s_{z_K}$ 
11    for  $j = K, K-1, \dots, 2$ :
12         $z_{j-1} \leftarrow D_{z_j, j}$ 
13         $t_{j-1} \leftarrow s_{z_{j-1}}$ 
14    return  $\mathbf{t}$ 
```

### 3 Autocomplete and Language Models

#### Definitions:

- *Text corpus*: a large database of text documents.
- *Language model*: a tool that calculates the probabilities of sentences; it can also estimate the probability of an upcoming word given a history of previous words.
- *Sentence*: a sequence of words.

#### Applications:

- Speech recognition.  
Example :  $P(\text{"I saw a van"}) > P(\text{"eyes awe of an"})$
- Spelling correction.  
Example: "He entered the **ship** to buy some groceries"  
 $P(\text{"entered the **shop** to buy"}) > P(\text{"entered the **ship** to buy"})$
- Augmentative communication.

#### 3.1 N-grams and Probabilities

An *N-gram* is a sequence of  $N$  elements which can be words, characters, or other elements.

#### Example

Corpus: "I am happy because I am learning"

- Unigrams: {"I", "am", "happy", "because", "learning" }
- Bigrams: { "I am", "am happy", "happy because", ... }
- Trigrams: { "I am happy", "am happy because", ... }

#### Sequence Notation

Corpus : "This is great ... teacher drinks tea ."  
 $w_1 \quad w_2 \quad w_3 \quad \dots \quad w_{498} \quad w_{499} \quad w_{500}$

$m = 500$

$w_1^m = w_2 \quad w_2 \quad \dots \quad w_m$

$w_1^3 = w_1 \quad w_2 \quad w_3$

$w_{m-2}^m = w_{m-2} \quad w_{m-1} \quad w_m$

#### N-gram Probability

Probability of  $N$ -gram:

$$\begin{aligned} P(w_n | w_{n-N+1}^{n-1}) &= \frac{C(w_{n-N+1}^{n-1} w_n)}{\sum_{w \in V} C(w_{n-N+1}^{n-1} w)} \\ &= \frac{C(w_{n-N+1}^{n-1} w_n)}{C(w_{n-N+1}^{n-1})} \\ C(w_{n-N+1}^{n-1} w_n) &= C(w_{n-N+1}^n) \end{aligned} \quad (2.15)$$

#### Example:

Corpus: "I am happy because I am learning"

- **Unigram probability**: Probability of unigram:

$$P(w) = \frac{C(w)}{m}$$

Size of corpus  $m = 7$

$$P(\text{"I"}) = \frac{2}{7} \quad P(\text{"happy"}) = \frac{1}{7}$$

- **Bigram probability**:

Probability of bigram:

$$P(w_i | w_{i-1}) = \frac{C(w_{i-1} w_i)}{\sum_{w \in V} C(w_{i-1} w)} = \frac{C(w_{i-1} w_i)}{C(w_{i-1})} \quad (2.16)$$

$$P(\text{"am"} | \text{"I"}) = \frac{C(\text{"I am"})}{C(\text{"I"})} = \frac{2}{2} = 1$$

$$P(\text{"learning"} | \text{"am"}) = \frac{C(\text{"am learning"})}{C(\text{"am"})} = \frac{1}{2}$$

- **Trigram probability**:

$$P(\text{"happy"} | \text{"I am"}) = \frac{C(\text{"I am happy"})}{C(\text{"I am"})} = \frac{1}{2}$$

#### 3.2 Sequence Probabilities

Conditional probability and chain rule:

$$P(B|A) \stackrel{(1.7)}{=} \frac{P(A, B)}{P(A)}$$

$$\Rightarrow P(A, B) = P(A)P(B|A)$$

$$P(A, B, C, D) = P(A)P(B|A)P(C|A, B)P(D|A, B, C)$$

$$P(\text{"the teacher drinks tea"}) = P(\text{"the"})$$

$$P(\text{"teacher"} | \text{"the"})$$

$$P(\text{"drinks"} | \text{"the teacher"})$$

$$P(\text{"tea"} | \text{"the teacher drinks"}) \quad (2.17)$$

The problem is that the corpus almost never contains the exact sentence we're interested in, so:

$$\begin{aligned} P(\text{"tea"} | \text{"the teacher drinks"}) &= \frac{C(\text{"the teacher drinks tea"})}{C(\text{"the teacher drinks"})} \\ &= 0 \end{aligned}$$

#### Approximation of Sequence Probability

$$P(\text{"tea"} | \text{"the teacher drinks"}) \approx P(\text{"tea"} | \text{"drinks"})$$

$$P(\text{"the teacher drinks tea"}) \stackrel{(2.17)}{\approx} P(\text{"the"})$$

$$P(\text{"teacher"} | \text{"the"})$$

$$P(\text{"drinks"} | \text{"teacher"})$$

$$P(\text{"tea"} | \text{"drinks"})$$

Markov assumption: only last  $N$  words matter:

- Bigram  $P(w_n | w_1^{n-1}) \approx P(w_n | w_{n-1})$
- $N$ -gram  $P(w_n | w_1^{n-1}) \approx P(w_n | w_{n-N+1}^{n-1})$
- Entire sequence modeled with bigram:

$$\begin{aligned} P(w_1^n) &\approx \prod_{i=1}^n P(w_i | w_{i-1}) \\ &\approx P(w_1)P(w_2 | w_1) \dots P(w_n | w_{n-1}) \end{aligned} \quad (2.18)$$

#### 3.3 Starting and Ending Sentences

Sentence : "the teacher drinks tea"

#### Start and End of Sentence Tokens for N-grams

Add  $N - 1$  start tokens  $\langle s \rangle$  and one end token  $\langle /s \rangle$

#### Example:

- Bigram:

$$\text{"the teacher drinks tea"} \Rightarrow \langle s \rangle \text{ the teacher drinks tea } \langle /s \rangle$$

$$P(\text{"the teacher drinks tea"}) = P(\text{"the"} | \langle s \rangle)$$

$$P(\text{"teacher"} | \text{"the"})$$

$$P(\text{"drinks"} | \text{"teacher"})$$

$$P(\text{"tea"} | \text{"drinks"})$$

$$P(\langle /s \rangle | \text{"tea"})$$

- Trigram:

$$\text{"the teacher drinks tea"} \Rightarrow$$

$$\langle s \rangle \langle s \rangle \text{ the teacher drinks tea } \langle /s \rangle$$

$$w_1 \quad w_2 \quad \dots \quad w_{n-1} \quad w_n$$

$$P(w_1^n) \approx P(w_1 | \langle s \rangle \langle s \rangle) P(w_2 | \langle s \rangle w_1) \dots P(\langle /s \rangle | w_n w_{n-1})$$

#### 3.4 The N-gram Language Model

#### Count Matrix

Represents the numerator of Equation 2.15 where:

- Rows: unique corpus  $(N - 1)$ -grams.
- Columns: unique corpus words.

The count matrix could be made in a single pass through the corpus.

#### Example:

Corpus: " $\langle s \rangle$  I study I learn  $\langle /s \rangle$ "

Bigram count matrix:

	$\langle s \rangle$	$\langle /s \rangle$	"I"	"study"	"learn"
$\langle s \rangle$	0	0	1	0	0
$\langle /s \rangle$	0	0	0	0	0
"I"	0	0	0	1	1
"study"	0	0	1	0	0
"learn"	0	1	0	0	0



## Probability Matrix

$$\text{sum}(\text{row}) = C \left( w_{n-N+1}^{n-1} \right) = \sum_{w \in V} C \left( w_{n-N+1}^{n-1}, w \right)$$

Divide each cell of the *bigram count matrix* by its row sum:

	$\langle s \rangle$	$\langle /s \rangle$	"I"	"study"	"learn"
$\langle s \rangle$	0	0	1	0	0
$\langle /s \rangle$	0	0	0	0	0
"I"	0	0	0	0.5	0.5
"study"	0	0	1	0	0
"learn"	0	1	0	0	0

## Log Probability

To avoid numerical underflow of multiplying numbers  $\leq 1$  use log probabilities:

$$\begin{aligned} \log(P(w_1^n)) &\stackrel{(2.18)}{\approx} \log \left( \prod_{i=1}^n P(w_i | w_{i-1}) \right) \\ &\approx \sum_{i=1}^n \log[P(w_i | w_{i-1})] \end{aligned}$$

## Sentence Probability

$$\begin{aligned} P(\langle s \rangle \text{ I learn } \langle /s \rangle) &= P(\text{"I"} | \langle s \rangle) P(\text{"learn"} | \text{"I"}) P(\langle /s \rangle | \text{"learn"}) \\ &= 1 \times 0.5 \times 1 \\ &= 0 \end{aligned}$$

## Next Word Prediction (Generative Language Model)

Algorithm:

- Choose sentence start.
- Choose next bigram starting with previous word.
- Continue until  $\langle s \rangle$  is picked.

## 3.5 Language Model Evaluation

### Test Data

- Train: Used to train the model.
- Validation: Used for tuning hyperparameters.
- Test: Test using a metric to reflect how well your model performs on unseen data.

For smaller corpora: 80% train, 10% validation and 10% test.  
For large corpora (typical for text): 98% train, 1% validation and 1% test.

## Perplexity Metric

*Perplexity* is defined as the state of confusion or uncertainty. You can think of it as a measure of complexity in a sample of text. It is used to tell us whether a set of sentences look like they were written by humans rather than by a simple program choosing words at random.

## Perplexity for Bigram Models:

$$\begin{aligned} PP(W) &= P(s_1, s_2, \dots, s_m)^{-\frac{1}{m}} \\ &= \sqrt[m]{\prod_{i=1}^m \prod_{j=1}^{|s_i|} \frac{1}{P(w_j^{(i)} | w_{j-1}^{(i)})}} \end{aligned} \quad (2.19)$$

Where:

- $W$  → test set containing  $m$  sentences  $s$ .
- $s_i$  →  $i$ -th sentence in the test set, each ending with  $\langle /s \rangle$ .
- $m$  → Count of all sentences in entire test set  $W$
- $|s_i|$  → Number of words in a sentence  $s_i$  including  $\langle /s \rangle$  but not  $\langle s \rangle$ .
- $w_j^{(i)}$  →  $j$ -th word in  $i$ -th sentence.

Concatenate all sentences in  $W$ :

$$PP(W) \stackrel{(2.19)}{=} \sqrt[M]{\prod_{i=1}^M \frac{1}{P(w_i | w_{i-1})}} \quad (2.20)$$

Where:

- $M$  → Number of words in entire test set  $W$  including  $\langle /s \rangle$  but not  $\langle s \rangle$ .
- $w_i$  →  $i$ -th word in test set.

## Log perplexity:

$$\log_2(PP(W)) \stackrel{(2.20)}{=} -\frac{1}{M} \sum_{i=1}^M \log_2[P(w_i | w_{i-1})]$$

## Properties of Perplexity Score

- A text written by a human is more likely to have a lower perplexity score.
- smaller perplexity = better model.
- Good language models have *perplexity score* between 60 and 20 and sometimes even lower for English.
- Character level models  $PP <$  word-based models  $PP$ .
- In a good model with perplexity between 20 and 60, log perplexity would be between 4.3 and 5.9.

## 3.6 Out of Vocabulary Words

### Using $\langle \text{UNK} \rangle$ in Corpus

- Create vocabulary  $V$ .
- Replace any word in corpus not in  $V$  by  $\langle \text{UNK} \rangle$ .
- Count probabilities with  $\langle \text{UNK} \rangle$  as with any other word.

## Example

Min frequency  $f = 2$

Corpus:

“ $\langle s \rangle$  Lyn drinks chocolate  $\langle /s \rangle$ ”  
“ $\langle s \rangle$  John drinks tea  $\langle /s \rangle$ ”  
“ $\langle s \rangle$  Lyn eats chocolate  $\langle /s \rangle$ ”

Corpus:

“ $\langle s \rangle$  Lyn drinks chocolate  $\langle /s \rangle$ ”  
“ $\langle s \rangle$   $\langle \text{UNK} \rangle$  drinks  $\langle \text{UNK} \rangle$   $\langle /s \rangle$ ”  
“ $\langle s \rangle$  Lyn  $\langle \text{UNK} \rangle$  chocolate  $\langle /s \rangle$ ”

Vocabulary : { “Lyn”, “drinks”, “chocolate” }

Input query:

“ $\langle s \rangle$  Adam drinks chocolate  $\langle /s \rangle$ ”

↓

“ $\langle s \rangle$   $\langle \text{UNK} \rangle$  drinks chocolate  $\langle /s \rangle$ ”

## How to Create Vocabulary $V$

- Criteria:
  - Min word frequency  $f$ .
  - Max  $|V|$ , include words by frequency (most common words).
- Use  $\langle \text{UNK} \rangle$  sparingly.
- Perplexity: only compare Language models with same  $V$ .

## 3.7 Smoothing

Problem:  $N$ -grams made of known words still might be missing in the training corpus. Their counts cannot be used for probability estimation and will evaluate to 0.

## Smoothing

- Add-one smoothing (Laplacian smoothing)

For bigrams:

$$P(w_n | w_{n-1}) \stackrel{(2.16)}{=} \frac{C(w_{n-1}, w_n) + 1}{\sum_{w \in V} [C(w_{n-1}, w) + 1]} = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + |V|}$$

- Add- $k$  smoothing

$$\begin{aligned} P(w_n | w_{n-N+1}^{n-1}) &\stackrel{(2.15)}{=} \frac{C(w_{n-N+1}^{n-1} w_n) + k}{\sum_{w \in V} [C(w_{n-N+1}^{n-1} w) + k]} \\ &= \frac{C(w_{n-N+1}^{n-1} w_n) + k}{C(w_{n-N+1}^{n-1}) + k|V|} \end{aligned} \quad (2.21)$$

For  $N$ -grams that have a zero count, the probability in

**Equation 2.21** becomes  $\frac{1}{|V|}$

For bigrams:

$$P(w_n | w_{n-1}) \stackrel{(2.16)}{=} \frac{C(w_{n-1}, w_n) + k}{\sum_{w \in V} [C(w_{n-1}, w) + k]} = \frac{C(w_{n-1}, w_n) + k}{C(w_{n-1}) + k|V|}$$

- Advanced methods:
  - Kneser-Ney smoothing.
  - Good-Turing smoothing.

**Back-off**

- If an  $N$ -gram is missing, use  $(N - 1)$ -gram. If  $(N - 1)$ -gram is missing, use  $(N - 2)$ -gram, ...
  - Probability discounting e.g. Katz backoff.
  - “Stupid” back-off: Use lower-order  $N$ -grams and multiply by a constant. A constant of about 0.4 was experimentally shown to work well.

**Example:**

Using corpus in [section 3.6](#)

$$P(\text{“chocolate”}|\text{“john drinks”}) = 0.4 \times P(\text{“chocolate”}|\text{“drinks”})$$

**Interpolation**

Example for trigram:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 \times P(w_n|w_{n-2}w_{n-1}) \\ &+ \lambda_2 \times P(w_n|w_{n-1}) \\ &+ \lambda_3 \times P(w_n)\end{aligned}$$

where  $\sum_i \lambda_i = 1$

$\lambda_i$  weights come from optimization on a validation set.

**Example:**

$$\begin{aligned}\hat{P}(\text{“chocolate”}|\text{“John drinks”}) &= 0.7 \times P(\text{“chocolate”}|\text{“John drinks”}) \\ &+ 0.2 \times P(\text{“chocolate”}|\text{“drinks”}) \\ &+ 0.1 \times P(\text{“chocolate”})\end{aligned}$$

## 4 Word Embeddings with Neural Networks

Learning objectives:

- Identify key concepts of word representations.
- Generate word embeddings
- Prepare text for machine learning.
- Implement the continuous bag-of-words model

### 4.1 Word Representations

#### • Integers

Simple but ordering has little semantic sense.

#### • Vectors

##### – One-hot vectors:

Representing each word by a vector of a length as the vocabulary size, with a one in the row representing the word and zero in all of the other rows.

Advantages: Simple, no implied ordering.

Disadvantages: Huge vectors, no embedded meaning.

#### – Word embedding vectors

Advantages:

- \* Low dimension
- \* Embed meaning:
  - eg. semantic distance.  
 $\text{forest} \approx \text{tree}$        $\text{forest} \not\approx \text{ticket}$
  - eg. analogies.  
 $\text{Paris} \rightarrow \text{France}$        $\text{Rome} \rightarrow ?$

### 4.2 Word Embedding Methods

#### Basic word embedding methods

- word2vec (Google, 2013)
  - Continuous bag-of-words (CBOW) [Mik+13b]
  - Continuous skip-gram/Skip-gram with negative sampling (SGNS) [Mik+13a]
- Global Vectors (GloVe) (Stanford, 2014) [PSM14]
- fastText (Facebook, 2016) [Mik+17]
  - Supports out-of-vocabulary (OOV) words.

#### Advanced word embedding methods:

Deep learning, contextual embeddings

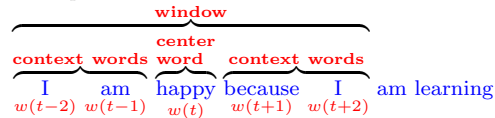
- BERT (Google, 2018) [Dev+19]
- ELMo (Allen Institute for AI, 2018) [Pet+18]
- GPT-2 (OpenAI, 2018) [Rad+19]

### 4.3 Continuous Bag of Words Model (CBOW)

Reference: [Mik+13b]

#### Creating a Training Example

Example:



Context half-size  $C = 2$

Window size = 5

A neural network takes *context words* as input and outputs the *center word*.

#### Cleaning and Tokenization

- Letter case  
“The” == “the” == “THE” → lowercase/upper case
- Punctuation  
“, ! . ? → .      “ ’ , ” →  $\Phi$       “...!! ??? → .
- Numbers  
 $1 \ 2 \ 3 \ 5 \ 8 \rightarrow \Phi$        $3.14159 \ 9021 \text{ as is } / \langle \text{NUMBER} \rangle$
- Special characters  
 $\Delta \ \$ \ \in \ \$ \ \P \ * \rightarrow \Phi$
- Special words  
emojis - hashtags

### Transforming Words into Vectors

Example:

Corpus :  $\overbrace{\text{I am}}^{\text{context words}} \overbrace{\text{happy}}^{\text{word}} \overbrace{\text{because I am learning}}^{\text{context words}}$   
 Vocabulary: am, because, happy, I, learning

Average of individual context words one-hot vectors:

$$\bar{\mathbf{x}} = \begin{bmatrix} \text{am} & \text{because} & \text{I} & \text{learning} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} / 4 = \begin{bmatrix} 0.25 \\ 0.25 \\ 0 \\ 0 \end{bmatrix}$$

Center word:

$$\mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

#### Architecture of the CBOW model

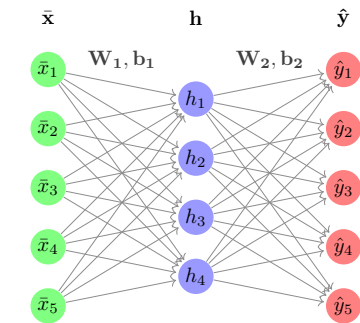


Figure 2.4: Architecture of the CBOW model

$N$ : Word embedding size (a hyperparameter)

$V$ : Vocabulary size

The neural network has three layers:

- Input layer  $\mathbf{x}$ , has  $V$  neurons.
- Hidden layer  $\mathbf{h}$ , has  $N$  neurons.
- Output layer  $\mathbf{y}$ , has  $V$  neurons.

**Forward Propagation**

For a single training example:

$$\begin{aligned}\mathbf{z}_1 &= \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1 \\ \mathbf{h} &= \text{ReLU}(\mathbf{z}_1) \\ \mathbf{z}_2 &= \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2 \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}_2)\end{aligned}$$

For a batch of training examples:

$$\begin{aligned}\mathbf{Z}_1 &= \mathbf{W}_1 \mathbf{X} + \mathbf{b}_1 \\ \mathbf{H} &= \text{ReLU}(\mathbf{Z}_1) \\ \mathbf{Z}_2 &= \mathbf{W}_2 \mathbf{H} + \mathbf{b}_2 \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{Z}_2)\end{aligned}$$

Dimensions:

$$\begin{aligned}\mathbf{x} &\in \mathbb{R}^{V \times 1} & \mathbf{h} &\in \mathbb{R}^{N \times 1} & \hat{\mathbf{y}} &\in \mathbb{R}^{V \times 1} \\ \mathbf{X} &\in \mathbb{R}^{V \times m} & \mathbf{H} &\in \mathbb{R}^{N \times m} & \hat{\mathbf{Y}} &\in \mathbb{R}^{V \times m} \\ \mathbf{W}_1 &\in \mathbb{R}^{N \times V} & \mathbf{b}_1 &\in \mathbb{R}^{N \times 1} & \mathbf{W}_2 &\in \mathbb{R}^{V \times N} & \mathbf{b}_2 &\in \mathbb{R}^{V \times 1}\end{aligned}$$

**Activation Functions**

$$\text{ReLU}(x) = \max(0, x)$$

$$\text{softmax}(\mathbf{z}) = \hat{\mathbf{y}} = \frac{e^{\mathbf{z}}}{\sum_{j=1}^V e^{z_j}}$$

$$\text{softmax}(z_i) = \hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^V e^{z_j}}$$

**Cost Function**

Cross-entropy loss function for a single example:

$$\begin{aligned}\mathcal{L} &= - \sum_{j=1}^V y_k \log \hat{y}_j \\ &= -\mathbf{y}^\top \odot \log(\hat{\mathbf{y}})\end{aligned}\tag{2.22}$$

Cross-entropy cost function of a batch of  $m$  examples:

$$\begin{aligned}\mathcal{J}_{\text{batch}} &= -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^V y_j^{(i)} \log \hat{y}_j^{(i)} \\ &\stackrel{(2.22)}{=} -\frac{1}{m} \sum_{i=1}^m \mathcal{L}^{(i)}\end{aligned}$$

**Backpropagation**

$$\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{W}_1} = \frac{1}{m} \text{ReLU} \left[ \mathbf{W}_2^\top (\hat{\mathbf{Y}} - \mathbf{Y}) \right] \mathbf{X}^\top$$

$$\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{W}_2} = \frac{1}{m} (\hat{\mathbf{Y}} - \mathbf{Y}) \mathbf{H}^\top$$

$$\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{b}_1} = \frac{1}{m} \text{ReLU} \left[ \mathbf{W}_2^\top (\hat{\mathbf{Y}} - \mathbf{Y}) \right] \mathbf{1}_m^\top$$

$$\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{b}_2} = \frac{1}{m} (\hat{\mathbf{Y}} - \mathbf{Y}) \mathbf{1}_m^\top$$

Where  $\mathbf{1}_m^\top$  is a row vector with  $m$  elements that are all 1s.

**Gradient Descent**

$$\mathbf{W}_1 := \mathbf{W}_1 - \alpha \frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{W}_1}$$

$$\mathbf{W}_2 := \mathbf{W}_2 - \alpha \frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{W}_2}$$

$$\mathbf{b}_1 := \mathbf{b}_1 - \alpha \frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{b}_1}$$

$$\mathbf{b}_2 := \mathbf{b}_2 - \alpha \frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{b}_2}$$

**Extracting Word Embedding Vectors**

To get the embedding vector  $\mathbf{w}^{(i)}$  of the word  $i$  in the vocabulary:

- Option 1: The columns of  $\mathbf{W}_1$  as the embedding vectors.

$$\mathbf{W}_1 = \left[ \mathbf{w}^{(1)} | \mathbf{w}^{(2)} | \dots | \mathbf{w}^{(V)} \right]$$

- Option 2: The columns of  $\mathbf{W}_2^\top$  as the embedding vectors.

$$\mathbf{W}_2 = \begin{bmatrix} \mathbf{w}^{(1)\top} \\ \mathbf{w}^{(2)\top} \\ \vdots \\ \mathbf{w}^{(V)\top} \end{bmatrix}$$

- Option 3: The columns of the average of  $\mathbf{W}_1$  and  $\mathbf{W}_2^\top$

$$\mathbf{W}_3 = \frac{1}{2} (\mathbf{W}_1 + \mathbf{W}_2^\top)$$

$$\mathbf{W}_3 = \left[ \mathbf{w}^{(1)} | \mathbf{w}^{(2)} | \dots | \mathbf{w}^{(V)} \right]$$

**Evaluating Word Embeddings**

- Intrinsic evaluation:** Test relationships between words

– Analogies

\* Semantic analogies

“France” is to “Paris” as “Italy” is to {?}

\* Syntactic analogies

“seen” is to “saw” as “been” is to {?}

Ambiguity:

“wolf” is to “pack” as “bee” is to {?}

→ swarm? colony?

– Clustering [ZTC16]

– Visualization

- Extrinsic evaluation:**

Test word embeddings on external task:

e.g. named entity recognition, parts-of-speech tagging.

Pros: Evaluates actual usefulness of embeddings.

Cons: Time-consuming - More difficult to troubleshoot.

## Chapter 3

### Sequence Models

#### 1 Neural Networks for Sentiment Analysis

##### 1.1 Notation

$n_v \rightarrow$  Vocabulary size.

$n_e \rightarrow$  Embedding size, where  $n_e \ll n_v$

$\mathbf{E} \rightarrow$  Embedding matrix.  $\mathbf{E} \in \mathbb{R}^{n_e \times n_v}$

$\mathbf{a}^{[i]} \rightarrow$  Activation of the  $i$ th layer.

$\mathbf{x} \rightarrow$  Input vector (one-hot vocabulary vector).  $\mathbf{x} \in \mathbb{R}^{n_v}$

##### 1.2 Feed forward pass

$$\mathbf{a}^{[0]} = \mathbf{x}$$

Dense Layer:

$$\mathbf{z}^{[i]} = \mathbf{W}^{[i]} \mathbf{a}^{[i-1]}\tag{3.1}$$

ReLU Layer:

$$\mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]}) = \text{ReLU}(\mathbf{z}^{[i]}) = \max(0, \mathbf{z}^{[i]})$$

An embedding layer takes an index assigned to each word (represented as a one-hot vector with size  $n_v$ ) from your vocabulary and maps it to a representation of that word with a determined embedding dimension ( $n_e$ ).

$$\begin{aligned}\mathbf{z}^{[1]} &\stackrel{(3.1)}{=} \mathbf{W}^{[1]} \mathbf{x} \\ &= \mathbf{E} \mathbf{x}\end{aligned}$$

For the embedding layer in your model, you'd have to learn a matrix of weights (embedding matrix)  $\mathbf{E}$ , of size  $(n_e \times n_v)$ . The size of the embedding could be treated as a hyperparameter in your model.

For sentiment analysis the output layer has two units corresponding to positive sentiment and negative sentiment.

#### 2 Recurrent Neural Networks for Language Modeling

##### 2.1 Vanilla RNN

$$\begin{aligned}\mathbf{h}^{(t)} &= g \left( \mathbf{W}_h \left[ \mathbf{h}^{(t-1)}, \mathbf{x}^{(t)} \right] + \mathbf{b}_h \right) \\ &= g \left( \mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{W}_{hx} \mathbf{x}^{(t)} + \mathbf{b}_h \right) \\ &= g \left( [\mathbf{W}_{hh} \mid \mathbf{W}_{hx}] \begin{bmatrix} \mathbf{h}^{(t-1)} \\ \mathbf{x}^{(t)} \end{bmatrix} + \mathbf{b}_h \right) \\ \hat{\mathbf{y}}^{(t)} &= g \left( \mathbf{W}_{yh} \mathbf{h}^{(t)} + \mathbf{b}_y \right)\end{aligned}$$

## Cross Entropy Loss

$$\mathcal{L}^{(t)}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)}) = - \sum_{j=1}^K y_j^{(t)} \log \hat{y}_j^{(t)}$$

The cost function is calculated by taking the average of loss function with respect to time

$$\begin{aligned} \mathcal{J} &= \frac{1}{T} \sum_{t=1}^T \mathcal{L}^{(t)}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)}) \\ &= -\frac{1}{T} \sum_{t=1}^T \sum_{j=1}^K y_j^{(t)} \log \hat{y}_j^{(t)} \end{aligned}$$

## 2.2 Gated Recurrent Unit (GRU)

Relevance gate:

$$\Gamma_r^{(t)} = \sigma(\mathbf{W}_r [\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_r)$$

Update gate:

$$\Gamma_u^{(t)} = \sigma(\mathbf{W}_u [\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_u)$$

Hidden state candidate:

$$\mathbf{h}'^{(t)} = \tanh(\mathbf{W}_h [\Gamma_r^{(t)} \odot \mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_h)$$

Output hidden state:

$$\mathbf{h}^{(t)} = \Gamma_u^{(t)} \odot \mathbf{h}'^{(t)} + (1 - \Gamma_u^{(t)}) \odot \mathbf{h}^{(t-1)}$$

Output prediction:

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{W}_y \mathbf{h}^{(t)} + \mathbf{b}_y)$$

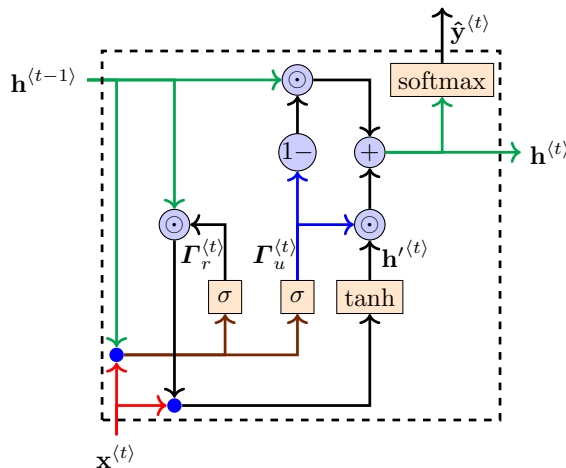


Figure 3.1: GRU Cell

## 2.3 Bidirectional RNN

$$\hat{\mathbf{y}}^{(t)} = g(\mathbf{W}_y [\vec{\mathbf{h}}^{(t)}, \overleftarrow{\mathbf{h}}^{(t)}] + \mathbf{b}_y)$$

## 2.4 Deep RNNs

Steps:

1. Get hidden states for current layer:

$$\mathbf{h}^{[l]\langle t \rangle} = f^{[l]}(\mathbf{W}_h^{[l]} [\mathbf{h}^{[l]\langle t-1 \rangle}, \mathbf{a}^{[l-1]\langle t \rangle}] + \mathbf{b}_h^{[l]})$$

2. Pass the activations to the next layer:

$$\mathbf{a}^{[l]\langle t \rangle} = f^{[l]}(\mathbf{W}_a^{[l]} \mathbf{h}^{[l]\langle t \rangle} + \mathbf{b}_a^{[l]})$$

A simpler architecture:

$$\mathbf{h}^{[l]\langle t \rangle} = g(\mathbf{W}_h^{[l]} [\mathbf{h}^{[l]\langle t-1 \rangle}, \mathbf{h}^{[l-1]\langle t \rangle}] + \mathbf{b}_h^{[l]})$$

## 2.5 Perplexity Score

Use log softmax as an activation function to get the prediction as log probabilities  $\log(\hat{\mathbf{y}}^{(t)})$

The *perplexity score* can be calculated as:

$$PP(W) = \sqrt[N]{\prod_{i=1}^N \mathbf{y}_i^{(t)\top} \hat{\mathbf{y}}^{(t)}}$$

$$\log(PP(W)) = -\frac{1}{N} \sum_{i=1}^N \mathbf{y}_i^{(t)\top} \log(\hat{\mathbf{y}}^{(t)})$$

where:

$\mathbf{y}^{(t)} \rightarrow$  target(label), encoded as one-hot vector.

$\hat{\mathbf{y}}^{(t)} \rightarrow$  prediction, a vector of probabilities.

$W \rightarrow$  test set containing N words(or characters).

See [section 3.5 - Properties of Perplexity Score](#)

## 2.6 Advantages of Using Deep RNNs

- Captures dependencies within a short range  
Using a statistical method like the ones described in [chapter 2](#) for language models will not give results as good as RNNs. The model will not be able to encode information seen previously in the data, and so the perplexity will increase.
- Takes up less RAM than other  $n$ -gram models  
Statistical  $n$ -gram models take up too much space and memory, so they are inefficient and slow.

## 2.7 RNNs Disadvantages

- Struggle with longer sequences.
- Prone to vanishing or exploding gradients.

## 3 LSTMs and Named Entity Recognition

### 3.1 Solving for vanishing or exploding gradients

- Identity RNN with ReLU activation  $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} -1 \\ 0 \end{matrix} \rightarrow 0$
- Gradient clipping  $32 \rightarrow 25$
- Skip connections.

### 3.2 LSTMs

- Learns when to remember and when to forget
- Anatomy:
  - A cell state
  - A hidden state with three gates
    - \* Forget gate: Decides what to keep.
    - \* Input gate: Decides what to add.
    - \* Output gate: Decides what the next hidden state will be.
  - Loops back again at the end of each time step after updating the states.
- Gates allow gradients to flow unchanged (offer a solution to vanishing gradients).

### 3.3 Applications of LSTMs

- Next character prediction.
- Chatbots.
- Music composition.
- Image captioning.
- Speech recognition.

### 3.4 Architecture of LSTM

The tanh layer ensures the values in your network stay numerically stable, by squeezing all values between -1 and 1. This prevents any of the values from the current inputs from becoming so large that they make the other values insignificant. Its output is multiplied element-wise by the sigmoid layer's output in both the input and output gates, ensuring an even flow through the LSTM unit.

### 3.5 Named Entity Recognition

Named entity recognition locates and extracts predefined entities from text such as places, organizations, names, time and dates.

Example of a labeled sentence:

B-per Sharon I flew I to B-geo Miami I last B-tim Friday

## 4 Siamese Networks

Siamese networks identify the *similarity* between things.

### Applications of siamese networks:

- Authentication of handwritten checks.
- Identify question duplicates on platforms like Quora or Stack Overflow.  
 “How old are you?” = “What is your age?”  
 “Where are you from?”  $\neq$  “Where are you going?”
- Face recognition.

### 4.1 Model Architecture

### 4.2 Cosine Similarity

$$\begin{aligned}\hat{y} &= s(\mathbf{v}_1, \mathbf{v}_2) \\ &= \cos(\mathbf{v}_1, \mathbf{v}_2) \\ &= \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \|\mathbf{v}_2\|}\end{aligned}\quad (3.2)$$

$$-1 \leq \hat{y} \leq 1$$

Compare *similarity score* against a threshold  $\tau$ :

$$\hat{y} \begin{cases} \leq \tau & \rightarrow \text{different} \\ > \tau & \rightarrow \text{same} \end{cases}$$

### 4.3 Triplet Loss Function

$\mathbf{A} \rightarrow$  Anchor.

$\mathbf{P} \rightarrow$  Positive example.

$\mathbf{N} \rightarrow$  Negative example.

$s \rightarrow$  Similarity function.

#### Example:

For the triplet  $(\mathbf{A}, \mathbf{P}, \mathbf{N})$ :

$$\begin{aligned}\text{“How old are you?”} &\rightarrow \mathbf{A} \text{ (Anchor)} \\ \text{“What is your age?”} &\rightarrow \mathbf{P} \text{ (Positive)} & s(\mathbf{A}, \mathbf{P}) \approx 1 \\ \text{“Where are you from?”} &\rightarrow \mathbf{N} \text{ (Negative)} & s(\mathbf{A}, \mathbf{N}) \approx -1\end{aligned}$$

Simple loss:

$$\begin{aligned}\text{diff} &= s(\mathbf{A}, \mathbf{N}) - s(\mathbf{A}, \mathbf{P}) \\ -2 &\leq \text{diff} \leq 2\end{aligned}\quad (3.3)$$

Non Linearity:

$$\begin{aligned}\mathcal{L}(\mathbf{A}, \mathbf{P}, \mathbf{N}) &= \begin{cases} 0; & \text{if diff} \leq 0 \\ \text{diff}; & \text{if diff} > 0 \end{cases} \\ 0 &\leq \mathcal{L}(\mathbf{A}, \mathbf{P}, \mathbf{N}) \leq 2\end{aligned}$$

Alpha margin:

$$\begin{aligned}\mathcal{L}(\mathbf{A}, \mathbf{P}, \mathbf{N}) &= \begin{cases} 0; & \text{if diff} + \alpha \leq 0 \\ \text{diff} + \alpha; & \text{if diff} + \alpha > 0 \end{cases} \\ &= \begin{cases} 0; & \text{if diff} \leq -\alpha \\ \text{diff} + \alpha; & \text{if diff} > -\alpha \end{cases} \\ &= \max(\text{diff} + \alpha, 0) \\ &\stackrel{(3.3)}{=} \max(s(\mathbf{A}, \mathbf{N}) - s(\mathbf{A}, \mathbf{P}) + \alpha, 0) \quad (3.4) \\ 0 &\leq \mathcal{L}(\mathbf{A}, \mathbf{P}, \mathbf{N}) \leq 2 + \alpha\end{aligned}$$

The *triplet cost* for all examples can be calculated as:

$$\mathcal{J} = \sum_{i=1}^m \mathcal{L}(\mathbf{A}^{(i)}, \mathbf{P}^{(i)}, \mathbf{N}^{(i)})$$

### Triplet Selection

Hard triplets are better for training. select hard triplets that satisfy:

$$s(\mathbf{A}, \mathbf{N}) \approx s(\mathbf{A}, \mathbf{P})$$

Harder to train. More to learn.

### Hard Negative Mining

**Mean negative:** mean of  $s(\mathbf{A}, \mathbf{N}^{(j)})$  for all negative examples  $\mathbf{N}^{(j)}$ .

$$\text{mean}_{\text{neg}} = \frac{1}{b-1} \sum_i s(\mathbf{A}, \mathbf{N}^{(j)})$$

Where  $b \rightarrow$  batch size.

**Closest negative:** The value  $s(\mathbf{A}, \mathbf{N}^{(j)})$  closest to (but less than) the value  $s(\mathbf{A}, \mathbf{P})$ . It is considered a *hard triplet* for training.

$$\begin{aligned}\mathcal{L}_1 &\stackrel{(3.4)}{=} \max(\text{mean}_{\text{neg}} - s(\mathbf{A}, \mathbf{P}) + \alpha, 0) \\ \mathcal{L}_2 &\stackrel{(3.4)}{=} \max(\text{closest}_{\text{neg}} - s(\mathbf{A}, \mathbf{P}) + \alpha, 0)\end{aligned}$$

$$\mathcal{L}_{\text{Full}}(\mathbf{A}, \mathbf{P}, \mathbf{N}) = \mathcal{L}_1 + \mathcal{L}_2$$

$$\mathcal{J}_{\text{Full}} = \sum_{i=1}^m \mathcal{L}_{\text{Full}}(\mathbf{A}^{(i)}, \mathbf{P}^{(i)}, \mathbf{N}^{(i)})$$

### 4.4 One Shot Learning

The network learns a similarity function which measure the similarity between 2 classes. It is trained once using the dataset and doesn't need to be retrained for new examples.

### 4.5 Testing Siamese Networks

1. Convert each input into an array of numbers.
2. Feed arrays into your model.
3. Compare  $\mathbf{v}_1, \mathbf{v}_2$  using cosine similarity.
4. Test against a threshold  $\tau$

# References

- [Dev+19] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv:1810.04805 [cs]* (May 2019). arXiv: 1810.04805. URL: <http://arxiv.org/abs/1810.04805> (visited on 09/02/2020).
- [JM19] Daniel Jurafsky and James H Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2019. URL: <https://web.stanford.edu/%20jurafsky/slp3/>.
- [Jur12] Dan Jurafsky. *Minimum Edit Distance*. Stanford University, Jan. 2012. URL: <https://web.stanford.edu/class/cs124/lec/med.pdf> (visited on 08/07/2020).
- [LJP03] Mark Lieberman, Yoon-Kyoung Joh, and Marjorie Pak. *Alphabetical list of part-of-speech tags used in the Penn Treebank Project*. 2003. URL: [https://www.ling.upenn.edu/courses/Fall\\_2003/ling001/penn\\_treebank\\_pos.html](https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html) (visited on 08/08/2020).
- [Mik+13a] Tomas Mikolov et al. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Vol. 26. Curran Associates, Inc., Oct. 2013, pp. 3111–3119. URL: <http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf> (visited on 07/22/2020).
- [Mik+13b] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv:1301.3781 [cs]* (Sept. 2013). arXiv: 1301.3781. URL: <http://arxiv.org/abs/1301.3781> (visited on 07/23/2020).
- [Mik+17] Tomas Mikolov et al. “Advances in Pre-Training Distributed Word Representations”. In: *arXiv:1712.09405 [cs]* (Dec. 2017). arXiv: 1712.09405. URL: <http://arxiv.org/abs/1712.09405> (visited on 09/02/2020).
- [Nor07] Peter Norvig. *How to Write a Spelling Corrector*. Feb. 2007. URL: <https://norvig.com/spell-correct.html> (visited on 08/04/2020).
- [Pet+18] Matthew E. Peters et al. “Deep contextualized word representations”. In: *arXiv:1802.05365 [cs]* (Mar. 2018). arXiv: 1802.05365. URL: <http://arxiv.org/abs/1802.05365> (visited on 09/02/2020).
- [Por80] Martin F. Porter. “An algorithm for suffix stripping”. In: *Program* 14.3 (1980), pp. 130–137. DOI: [10.1108/eb046814](https://doi.org/10.1108/eb046814). URL: <https://tartarus.org/martin/PorterStemmer/>.
- [PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: [10.3115/v1/D14-1162](https://doi.org/10.3115/v1/D14-1162). URL: <https://www.aclweb.org/anthology/D14-1162> (visited on 07/23/2020).
- [Rad+19] A. Radford et al. “Language Models are Unsupervised Multitask Learners”. In: 2019. URL: <https://openai.com/blog/better-language-models/>.
- [San90] Beatrice Santorini. “Part-of-speech tagging guidelines for the penn treebank project (3rd revision)”. In: *Technical Reports (CIS)* (1990), p. 570. URL: <https://catalog.ldc.upenn.edu/docs/LDC99T42/tagguid1.pdf>.
- [ZTC16] Michael Zhai, Johnny Tan, and Jinho D. Choi. “Intrinsic and extrinsic evaluations of word embeddings”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, Feb. 2016, pp. 4282–4283. (Visited on 09/04/2020).