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 General Comments About Notation

Vectors are represented by bold small letters (example: $\mathbf{x}$) and matrices are represented by bold capital letters (example: $\boldsymbol{X}$).

1.2 Notation

V: Vocabulary size, the number of unique words in the entire set of sentences.
$\boldsymbol{\theta}$: Parameter vector, $\boldsymbol{\theta} = [\theta_0, \theta_1, \dots, \theta_n]$
m: Number of examples (sentences)
$P(\text{class})$: Probability that a sentence is in a given class.
$\text{class} \in \{\text{pos}, \text{neg}\}$.
$\text{freq}(w_i, \text{class})$: Frequency of a word w_i in a specific class.

1.3 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.

1.4 Feature Extraction with Frequencies

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

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

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

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

1.5 Logistic Regression: Regression and Sigmoid

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

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

The hypothesis function h (sigmoid function σ):

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

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

1.6 Cost Function

The loss function for a single training example is:

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

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

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

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.

1.7 Gradient Descent

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

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

To update the weight θ_j using gradient descent:

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

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

1.8 Vectorized Implementation

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

$$\mathbf{z} \overset{(1.2)}{=} \boldsymbol{X}\boldsymbol{\theta}$$

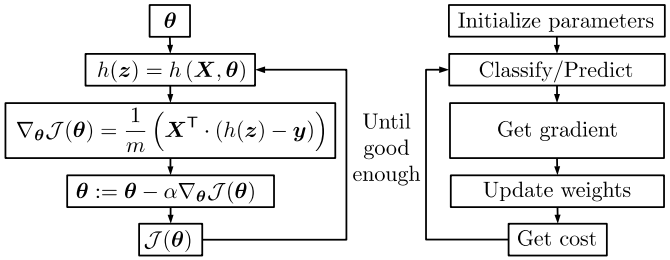
$$h\left(\boldsymbol{X}, \boldsymbol{\theta}\right) \overset{(1.3)}{=} h(\mathbf{z}) = \sigma\left(\mathbf{z}\right) = \frac{1}{1 + e^{-\mathbf{z}}}$$

$$\mathcal{J}(\boldsymbol{\theta}) \overset{(1.4)}{=} -\frac{1}{m}\left[\mathbf{y}^\intercal \cdot \log\left(h(\mathbf{z})\right) + (1 - \mathbf{y})^\intercal \cdot \log\left(1 - h(\mathbf{z})\right)\right]$$

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \overset{(1.5)}{=} \frac{1}{m}\left(\boldsymbol{X}^\intercal \cdot \left(h(\mathbf{z}) - \mathbf{y}\right)\right)$$

$$\boldsymbol{\theta} \overset{(1.6)}{:=} \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$$

Figure 1.1: Training Logistic Regression



1.9 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 $\boldsymbol{X}^{(\text{val})}, \mathbf{y}^{(val)}$ using trained weights $\boldsymbol{\theta}$.
2. Calculate $h(\boldsymbol{X}^{(\text{val})}, \boldsymbol{\theta}) = h(\mathbf{z})$
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:

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

2 Naïve Bayes

2.1 Conditional Probability and Bayes Rule

Conditional Probability:

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

Bayes Rule:

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

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

class $\in \{\text{pos}, \text{neg}\}$.
w: A unique word in the vocabulary.
ratio(w_i): Ratio of the probability that the word w_i being positive to being negative.
N_{class}: The total number of words in a class.
N: total number of words in the corpus.

2.4 Naïve Bayes Introduction

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

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

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

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

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

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

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

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

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

$$P(\text{class}|w_i) \overset{(1.8)}{=} \frac{P(\text{class})P(w_i|\text{class})}{P(w_i)} \tag{1.13}$$

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

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

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

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

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:

$$\log \frac{P(\text{pos}|\text{sentence})}{P(\text{neg}|\text{sentence})} \overset{(1.16)}{=} \log \left[\frac{P(\text{pos})}{P(\text{neg})} \prod_{i=1}^{n} \text{ratio}(w_i) \right]$$

$$= \log \frac{P(\text{pos})}{P(\text{neg})} + \sum_{i=1}^{n} \log(\text{ratio}(w_i))$$

$$= \underbrace{\log \frac{P(\text{pos})}{P(\text{neg})}}_{\text{logprior}} + \underbrace{\sum_{i=1}^{n} \log \frac{P(w_i|\text{pos}) + 1}{P(w_i|\text{neg}) + 1}}_{\text{log likelihood}}$$

$$= \log \frac{P(\text{pos})}{P(\text{neg})} + \underbrace{\sum_{i=1}^{n} \lambda(w_i)}_{\substack{\log \\ \text{likelihood}}} \tag{1.17}$$

Where

$$\lambda(w_i) = \log(\text{ratio}(w_i)) \overset{(1.12)}{=} \log \frac{P(w_i|\text{pos}) + 1}{P(w_i|\text{neg}) + 1} \tag{1.18}$$

$$\lambda(w_i) \begin{cases} < 0 & \text{Negative word.} \\ = 0 & \text{Neutral word.} \\ > 0 & \text{Positive word.} \end{cases}$$

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

2.5 Training Naïve Bayes

1. Collect and annotate corpus.
   Preprocess text:
   - Lowercase.
   - Remove punctuation, URLs, names.
   - Remove stop words.
   - Stemming [Por80].
   - Tokenize sentences $\boldsymbol{w} = [w_1, w_2, \ldots, w_n]$
2. Word count.
   (a) Compute freq$(w, class)$ for every word in the vocabulary.
   (b) Compute N_{class} [equation 1.9]
3. Compute conditional probabilities $P(w|\text{pos}), P(w|\text{neg})$ [equation 1.11]
4. Calculate the *lambda score* ($\lambda(w)$) for each word [equation 1.18]
5. Get the *logprior*:
   $$\log \frac{P(\text{pos})}{P(\text{neg})} \overset{(1.10)}{=} \log \frac{N_{\text{pos}}}{N_{\text{neg}}}$$
   If you are working with a balanced dataset ($N_{\text{pos}} = N_{\text{neg}}$), then 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 $i \in \{1, \ldots, m_{(\text{val})}\}$ 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 $\boldsymbol{X}^{(\text{val})}, \mathbf{y}^{(\text{val})}$
2. first, calculate *log likelihood score* for each sentence in the examples [equation 1.17]
3. Predict $\hat{y}_i^{(\text{val})}$ for each example as follows
   $$\hat{y}_i^{(\text{val})} = \begin{cases} 1, & \text{If } \textit{log likelihood score} > 0 \\ 0, & \text{otherwise} \end{cases}$$
4. Calculate the *accuracy score* for all examples in the validation set:
   $$\text{accuracy} = \frac{1}{m_{(\text{val})}} \sum_{i=1}^{m_{(\text{val})}} \left(\hat{y}_i^{(\text{val})} == y_i^{(\text{val})} \right)$$
   $$= 1 - \underbrace{\frac{1}{m_{(\text{val})}} \sum_{i=1}^{m_{(\text{val})}} \left| \hat{y}_i^{(\text{val})} - y_i^{(\text{val})} \right|}_{\text{error}}$$
   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:

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

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 $\boldsymbol{X} = [\mathbf{x}_1|\mathbf{x}_2|\ldots|\mathbf{x}_n]^\mathsf{T} \in \mathbb{R}^{m\times n}$, 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 $\bar{\boldsymbol{X}}$

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

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

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

$$\bar{\boldsymbol{X}} = [\bar{\mathbf{x}}_1|\bar{\mathbf{x}}_2|\ldots|\bar{\mathbf{x}}_n]^\mathsf{T}$$

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

$$\boldsymbol{\Sigma} = \frac{1}{m}\bar{\boldsymbol{X}}^\mathsf{T}\bar{\boldsymbol{X}}$$

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

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

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 $\boldsymbol{X'} \in \mathbb{R}^{m\times k}$.

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

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

The precentage 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 $\boldsymbol{X} = [\mathbf{x}_1|\mathbf{x}_2|\ldots|\mathbf{x}_m]^\mathsf{T}$ and a translation subset of *destination language* dataset $\boldsymbol{Y} = [\mathbf{y}_1|\mathbf{y}_2|\ldots|\mathbf{y}_m]^\mathsf{T}$ We want to find a transformation matrix $\boldsymbol{R}$ such that:

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

Cost function:

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

where:

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

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

- 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*:

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

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

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

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

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

The translation of a word i can be found using k-nearest neighbor of $\hat{\mathbf{y}}_i$ from $\boldsymbol{Y}$ 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 impelmented 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) π 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:

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

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

1. Identify a misspelled word.
   Words not in the dictionary are misspelled words.
2. 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

3. Filter candidates.
   Given a *vocabulary*, filter the *edit* list for *candidate words* found in the vocabulary.
4. Calculate word probabilities.

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

Where:

- $P(w)$: Probability of a word.
- count(w): Number of times the word apprears.
- M: Total number of words in the *corpus*.

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

Algorithm 1: Autocorrect

```
def autocorrect(word, n):
    Data:
    probs: a dictionary that maps each word to its
    probability in the corpus.
        probs[w → P(w)](x) = { P(w) = count(w)/M   for x = w
                             { 0                   otherwise
    vocab: a set containing all the vocabulary.
    Result: n-best: a set of tuples with the most
            probable n corrected words and their
            probabilities.
    suggestions = φ
    n-best= φ
    if word ∈ vocab:
        suggestions = suggestions ∪ {word}
    else:
        one-edit-set = one-edit-distance(word) ∩ vocab
        if one-edit-set ≠ φ:
            suggestions = suggestions ∪ one-edit-set
        else:
            two-edit-set = two-edit-distance(word) ∩ vocab
            if two-edit-set ≠ φ:
                suggestions = suggestions ∪ two-edit-set
            else:
                suggestions = suggestions ∪ {word}
    best-words[w → probs(w)](x) =
      {x = w|w ∈ suggestions}
    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:

$$D[0,0]=0 \tag{2.1}$$
$$D[i,0]=D[i-1,0]+del\_cost(source[i]) \tag{2.2}$$
$$D[0,j]=D[0,j-1]+ins\_cost(target[j]) \tag{2.3}$$

Per cell operations:

$$D[i,j]=$$

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

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

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

Figure 2.1: Minimum Edit Distance Table

Algorithm 2: Minimun Edit Distance

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:
  $\boldsymbol{D}$: 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
  $D[0,0] \overset{(2.1)}{=} 0$
  for $i \in \{1,2,\ldots,m\}$:
    $D[i,0] \overset{(2.2)}{=} [i-1,0] + del\_cost$
    for $j \in \{1,2,\ldots,n\}$:
      $D[0,j] \overset{(2.3)}{=} [0,j-1] + ins\_cost$
      if $source[i-1] = target[j-1]$:
        r_cost $= 0$
      else:
        r_cost $= 2$
      $$D[i,j] \overset{(2.4)}{=} \min \begin{cases} D[i-1,j] & + del\_cost \\ D[i,j-1] & + ins\_cost \\ D[i-1,j-1] & + r\_cost \end{cases}$$
  $med \overset{(2.5)}{=} D[m,n]$

Example:

$$source \longrightarrow target$$
$$\text{“play”} \longrightarrow \text{“stay”}$$

$$\begin{aligned} D[i,j] &= source[:i] \longrightarrow target[:j] \\ D[2,3] &= \text{“pl”} \longrightarrow \text{“sta”} \\ D[0,0] &= \# \longrightarrow \# \\ D[m,n] &= source \longrightarrow target \end{aligned}$$

where #: empty string.

i \ j		0	1	2	3	4
		#	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” $\longrightarrow$ “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

Sates:

$$\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$$

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

Initial Probability Vector

$$\boldsymbol{\pi} = [\pi_1, \pi_2, \ldots, \pi_N]$$

Example:

$\boldsymbol{\pi} =$

	NN	VB	O
π (initial)	0.4	0.1	0.5

The Transition Matrix

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

$$\boldsymbol{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,N} \\ a_{2,1} & a_{2,2} & \dots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \dots & a_{N,N} \end{pmatrix} \tag{2.6}$$

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

For all the outgoing transition probabilities:

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

Example:

$\boldsymbol{A} =$

	NN	VB	O
NN (noun)	0.2	0.2	0.6
VB (verb)	0.4	0.3	0.3
O (other)	0.2	0.3	0.5

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 $\mathcal{S} = \{s_1, s_2, \dots, s_N\}$ (parts of speech) of hidden Markov model to the *observables* or *emissions* (words of corpus) $\mathcal{O} = \{o_1, o_2, \dots, o_V\}$.

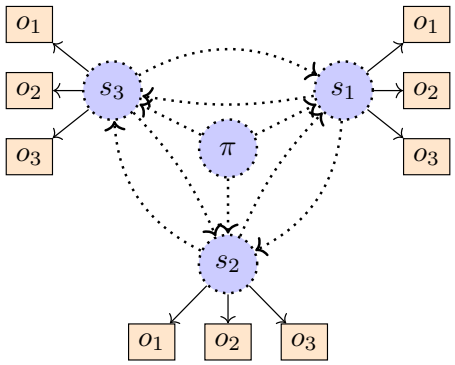


Figure 2.3: Hidden Markov Model

Emission Matrix

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

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

Emission matrix $\boldsymbol{B}$ 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:

$\boldsymbol{B} =$

	going	to	eat	...
NN (noun)	0.5	0.1	0.02	
VB (verb)	0.3	0.1	0.5	
O (other)	0.3	0.5	0.68	

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

- "He lay on his $\overbrace{\text{back}}^{\text{NN}}$"
- "I will be $\overbrace{\text{back}}^{\text{RB}}$"

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:

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

Where

- N is the total number of tags.

Smoothing

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

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

Where

- $C(t_{i-1})$ is the count of the previous POS tag occurrence in the corpus.
- ε 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:

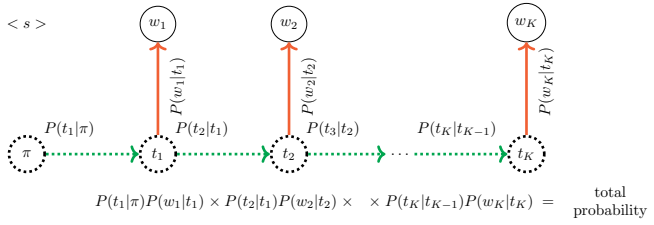
$$P(w_i|t_i) \overset{(2.7)}{=} b_{\text{rindex}(t_i),\text{cindex}(w_i)} = \frac{C(t_i, w_i) + \varepsilon}{\sum_{j=1}^{V} C(t_i, w_j) + V \cdot \varepsilon} = \frac{C(t_i, w_i) + \varepsilon}{C(t_i) + V \cdot \varepsilon}$$

Where

- w is a word (observable) in the corpus.
- $C(t_i)$ is the number of times a tag has occured 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 populates and then use the auxiliary matrices $\boldsymbol{C}$ and $\boldsymbol{D}$.
The matrix $\boldsymbol{C} \in \mathbb{R}^{N \times K}$ holds the intermediate optimal probabilities.
The matrix $\boldsymbol{D} \in \mathbb{R}^{N \times K}$ 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).

$\boldsymbol{C} =$

	w_1	w_2	$\dots$	w_K
t_1				
$\vdots$				
t_N				

$\boldsymbol{D} =$

	w_1	w_2	$\dots$	w_K
t_1				
$\vdots$				
t_N				

Steps

1. **Initialization:**

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

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

   For matrix $\boldsymbol{D}$, 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:

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

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

2. **Forward pass:**

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

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

   where

   (a) $a_{k,i}$ is the transition probability from the parts of speech tag t_k to the current tag t_i.

   (b) $c_{k,j-1}$ represents the probability for the preceding path you've traversed.

   Vectorized:

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

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

   where $\boldsymbol{A}^\mathsf{T} = \left[\mathbf{a}'_1|\mathbf{a}'_2|\dots|\mathbf{a}'_N\right]$

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 $c_{i,K}$

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

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

Implementation note:
For numerical stability use log probabilities:

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

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

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

Vectorized:

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

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

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

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, $w_i \in \mathcal{O}$
- $\boldsymbol{A} \in \mathbb{R}^{N \times N}$. Transition matrix
- $\boldsymbol{B} \in \mathbb{R}^{N \times V}$. 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.

def *VITERBI*$(\mathcal{O}, \mathcal{S}, \boldsymbol{\pi}, \mathbf{y}, \boldsymbol{A}, \boldsymbol{B})$**:**
  `/* Initialization */`
  for each *state* $i = 1, 2, \dots, N$**:**
    $C_{i,1} \leftarrow \log(\pi_i) + \log(B_{i,\text{cindex}(w_1)})$
    $D_{i,1} \leftarrow 0$
  `/* Forward pass */`
  for each *observation* $j = 2, 3, \dots, K$**:**
    for each *state* $i = 1, 2, \dots, N$**:**
      $C_{ij} \leftarrow \max_k C_{k,j-1} + \log(A_{k,i}) + \log(B_{i,\text{cindex}(w_j)})$
      $D_{ij} \leftarrow \arg\max_k C_{k,j-1} + \log(A_{k,i}) + \log(B_{i,\text{cindex}(w_j)})$
  `/* Backward pass */`
  $z_K \leftarrow \arg\max_i C_{i,K}$
  $t_K \leftarrow s_{z_K}$
  for $j = K, K-1, \dots, 2$**:**
    $z_{j-1} \leftarrow D_{z_j,\, j}$
    $t_{j-1} \leftarrow s_{z_{j-1}}$
  return 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 (w_1) is (w_2) great (w_3) . . . teacher (w_{498}) drinks (w_{499}) tea (w_{500}) .

$m = 500$

$w_1^m = w_2\ w_2\ \ldots\ w_m$

$w_1^3 = w_1\ w_2\ w_3$

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

N-gram Probability

Probability of N-gram:

$$P\left(w_n|w_{n-N+1}^{n-1}\right) = \frac{C\left(w_{n-N+1}^{n-1}\ w_n\right)}{\sum_{w\in V} C\left(w_{n-N+1}^{n-1}\ w\right)} = \frac{C\left(w_{n-N+1}^{n-1}\ w_n\right)}{C\left(w_{n-N+1}^{n-1}\right)} \tag{2.15}$$

$$C\left(w_{n-N+1}^{n-1} w_n\right) = C\left(w_{n-N+1}^{n}\right)$$

Example:
Corpus: “I am happy because I am learning”

- **Unigram probability:** Probability of unigram:
  $P(w) = \frac{C(w)}{m}$
  Size of corpus $m = 7$

$$P(\text{“I”}) = \frac{2}{7} \qquad P(\text{“happy”}) = \frac{1}{7}$$

- **Bigram probability:**
  Probability of bigram:

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

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

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

- **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:

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

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

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

$$\begin{aligned} P(\text{“the teacher drinks tea”}) = & P(\text{“the”}) \\ & P(\text{“teacher”}|\text{“the”}) \\ & P(\text{“drinks”}|\text{“the teacher”}) \\ & P(\text{“tea”}|\text{“the teacher drinks”}) \end{aligned} \tag{2.17}$$

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

$$P(\text{“tea”}|\text{“the teacher drinks”}) = \frac{C(\text{“the teacher drinks tea”})}{C(\text{“the teacher drinks”})} = 0$$

Approximation of Sequence Probability

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

$$\begin{aligned} P(\text{“the teacher drinks tea”}) \overset{(2.17)}{\approx} & P(\text{“the”}) \\ & P(\text{“teacher”}|\text{“the”}) \\ & P(\text{“drinks”}|\text{“teacher”}) \\ & P(\text{“tea”}|\text{“drinks”}) \end{aligned}$$

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:

$$P(w_1^n) \approx \prod_{i=1}^{n} P(w_i|w_{i-1}) \approx P(w_1)P(w_2|w_1)\ldots P(w_n|w_{n-1}) \tag{2.18}$$

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:
  “the teacher drinks tea” ⇒ “$\langle s\rangle$ the teacher drinks tea $\langle /s\rangle$”

$$\begin{aligned} P(\text{“the teacher drinks tea”}) = & P(\text{“the”}|\langle s\rangle) \\ & P(\text{“teacher”}|\text{“the”}) \\ & P(\text{“drinks”}|\text{“teacher”}) \\ & P(\text{“tea”}|\text{“drinks”}) \\ & P(\langle /s\rangle|\text{“tea”}) \end{aligned}$$

- Trigram:
  “the teacher drinks tea” ⇒
  “$\langle s\rangle$ $\langle s\rangle$ the (w_1) teacher (w_2) . . . drinks (w_{n-1}) tea (w_n) $\langle /s\rangle$”

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

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

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

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 <= 1 use log probabilities:

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

Sentence Probability

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

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:

$$PP(W) = P(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m)^{-\frac{1}{m}}$$
$$= \sqrt[m]{\prod_{i=1}^{m} \prod_{j=1}^{|\mathbf{s}_i|} \frac{1}{P(w_j^{(i)}|w_{j-1}^{(i)})}} \tag{2.19}$$

Where:

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

Concatenate all sentences in W:

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

Where:

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

Log perplexity:

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

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 $\langle$UNK$\rangle$ in Corpus

- Create vocabulary V.
- Replace any word in corpus not in V by $\langle$UNK$\rangle$.
- Count probabilities with $\langle$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 $\langle /s \rangle$"

$\Longrightarrow$

Corpus:
"$\langle s \rangle$ Lyn drinks chocolate $\langle /s \rangle$"
"$\langle s \rangle$ $\langle$UNK$\rangle$ drinks $\langle$UNK$\rangle$ $\langle /s \rangle$"
"$\langle s \rangle$ Lyn $\langle$UNK$\rangle$ chocolate $\langle /s \rangle$"

Vocabulary : {"Lyn", "drinks", "chocolate"}
Input query:
"$\langle s \rangle$ Adam drinks chocolate $\langle /s \rangle$"
$\Downarrow$
"$\langle s \rangle$ $\langle$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 $\langle$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.

Smooting

- Add-one smoothing (Laplacian smooting)
  For bigrams:
  $$P(w_n|w_{n-1}) \overset{(2.16)}{=} \frac{C(w_{n-1}, w_n) + 1}{\sum_{w \in V}[C(w_{n-1}, w) + 1]} = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + |V|}$$
- Add-k smoothing
  $$P\left(w_n|w_{n-N+1}^{n-1}\right) \overset{(2.15)}{=} \frac{C\left(w_{n-N+1}^{n-1}\, w_n\right) + k}{\sum_{w \in V}\left[C\left(w_{n-N+1}^{n-1}\, w\right) + k\right]}$$
  $$= \frac{C\left(w_{n-N+1}^{n-1}\, w_n\right) + k}{C\left(w_{n-N+1}^{n-1}\right) + k|V|} \tag{2.21}$$
  For N-grams that have a zero count, the probability in Equation 2.21 becomes $\frac{1}{|V|}$
  For bigrams:
  $$P(w_n|w_{n-1}) \overset{(2.16)}{=} \frac{C(w_{n-1}, w_n) + k}{\sum_{w \in V}[C(w_{n-1}, w) + k]} = \frac{C(w_{n-1}, w_n) + k}{C(w_{n-1}) + k|V|}$$
- Advanced methods:
  - Kneser-Ney smoothing.
  - Good-Turing smooting.

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 $\sum_i \lambda_i = 1$

λ_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.
      forest $\approx$ tree forest $\not\approx$ ticket
      · eg. analogies.
      Paris $\rightarrow$ France 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:

window
context words | center word | context words
I ($w(t-2)$) am ($w(t-1)$) | happy ($w(t)$) | because ($w(t+1)$) I ($w(t+2)$) am learning

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" $\rightarrow$ lowercase/upper case
- Punctuation
  , ! . ? $\rightarrow$. " ' , " $\rightarrow \Phi$ ... !! ??? $\rightarrow$.
- Numbers
  1 2 3 5 8 $\rightarrow \Phi$ 3.14159 9021 as is / ⟨NUMBER⟩
- Special characters
  Δ $ € § ¶ * $\rightarrow \Phi$
- Special words
  emojis - hashtags

Transforming Words into Vectors

Example:

context words | center word | context words
Corpus : I am | happy | because I am learning
Vocabulary: am, because, happy, I, learning

Average of individual context words one-hot vectors:

$$\bar{\mathbf{x}} = \begin{bmatrix} \text{am} \\ \text{because} \\ \text{happy} \\ \text{I} \\ \text{learning} \end{bmatrix} \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix}_{\text{am}} + \begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix}_{\text{because}} + \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix}_{\text{I}} + \begin{bmatrix}0\\0\\0\\0\\1\end{bmatrix}_{\text{learning}} /4 = \begin{bmatrix}0.25\\0.25\\0\\0.5\\0\end{bmatrix}_{\text{I am because I}}$$

Center word:

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

Architecture of the CBOW model

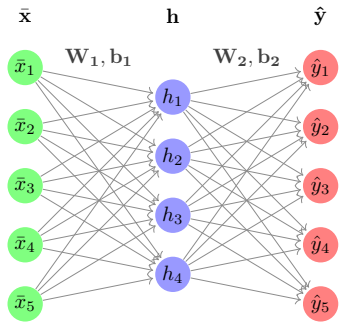


Figure 2.4: Architecture of the CBOW model

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.
- Ouptut layer $\hat{\mathbf{y}}$, has V neurons.

Forward Propagation

For a single training example:

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

For a batch of training examples:

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

Dimensions:

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

Activation Functions

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

$$\text{softmax}(\mathbf{z}) = \hat{\mathbf{y}} = \frac{e^{\mathbf{z}}}{\sum_{j=1}^{V} e^{z_j}} \tag{2.22}$$

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

Alternative implementation of softmax:

$$\begin{aligned}\log \text{softmax}(\mathbf{z}) &\overset{(2.22)}{=} \log \frac{e^{\mathbf{z}}}{\sum_{j=1}^{V} e^{z_j}} \\ &= \log e^{\mathbf{z}} - \log \sum_{j=1}^{V} e^{z_j} \\ &= \mathbf{z} - \log \sum_{j=1}^{V} e^{z_j} \\ \therefore \text{softmax}(\mathbf{z}) &= e^{\mathbf{z} - \log \sum_{j=1}^{V} e^{z_j}}\end{aligned}$$

Cost Function

Cross-entropy loss function for a single example:

$$\mathcal{L} = -\sum_{j=1}^{V} y_k \log \hat{y}_j \tag{2.23}$$
$$= -\mathbf{y}^{\mathsf{T}} \odot \log(\hat{\mathbf{y}})$$

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

$$\begin{aligned}\mathcal{J}_{\text{batch}} &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{V} y_j^{(i)} \log \hat{y}_j^{(i)} \\ &\overset{(2.23)}{=} -\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}^{(i)}\end{aligned}$$

Backpropagation

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

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

Gradient Descent

$$\boldsymbol{W}_1 := \boldsymbol{W}_1 - \alpha\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \boldsymbol{W}_1}$$
$$\boldsymbol{W}_2 := \boldsymbol{W}_2 - \alpha\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \boldsymbol{W}_2}$$
$$\mathbf{b}_1 := \mathbf{b}_1 - \alpha\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{b}_1}$$
$$\mathbf{b}_2 := \mathbf{b}_2 - \alpha\frac{\partial \mathcal{J}_{\text{batch}}}{\partial \mathbf{b}_2}$$

Extracting Word Embedding Vectors

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

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

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

- Option 2: The columns of $\boldsymbol{W}_2^{\mathsf{T}}$ as the embedding vectors.

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

- Option 3: The columns of the average of $\boldsymbol{W}_1$ and $\boldsymbol{W}_2^{\mathsf{T}}$

$$\boldsymbol{W}_3 = \frac{1}{2}\left(\boldsymbol{W}_1 + \boldsymbol{W}_2^{\mathsf{T}}\right)$$
$$\boldsymbol{W}_3 = \left[\mathbf{w}^{(1)}|\mathbf{w}^{(2)}|\dots|\mathbf{w}^{(V)}\right]$$

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 dize, where $n_e << n_v$
$\boldsymbol{E} \rightarrow$ Embedding matrix. $\boldsymbol{E} \in \mathbb{R}^{n_e \times n_v}$
$\mathbf{a}^{[i]} \rightarrow$ Activation of the ith 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:

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

ReLU Layer:

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

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 embeding dimension (n_e).

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

For the embedding layer in your model, you'd have to learn a matrix of weights (embedding matrix)$\boldsymbol{E}$, 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

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

$$\hat{\mathbf{y}}^{\langle t\rangle} = g\left(\boldsymbol{W}_{\mathbf{yh}}\mathbf{h}^{\langle t\rangle} + \mathbf{b}_{\mathbf{y}}\right)$$

Cross Entropy Loss

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

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

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

2.2 Gated Recurrent Unit (GRU)

Relevance gate: $\boldsymbol{\Gamma}_r^{\langle t\rangle} = \sigma\left(\boldsymbol{W}_{\mathbf{r}}\left[\mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b}_{\mathbf{r}}\right)$

Update gate: $\boldsymbol{\Gamma}_u^{\langle t\rangle} = \sigma\left(\boldsymbol{W}_{\mathbf{u}}\left[\mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b}_{\mathbf{u}}\right)$

Hidden state candidate:

$$\mathbf{h}'^{\langle t\rangle} = \tanh\left(\boldsymbol{W}_{\mathbf{h}}\left[\boldsymbol{\Gamma}_r^{\langle t\rangle} \odot \mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b}_{\mathbf{h}}\right)$$

Output hidden state:

$$\mathbf{h}^{\langle t\rangle} = \boldsymbol{\Gamma}_u^{\langle t\rangle} \odot \mathbf{h}'^{\langle t\rangle} + \left(1 - \boldsymbol{\Gamma}_u^{\langle t\rangle}\right) \odot \mathbf{h}^{\langle t-1\rangle}$$

Output prediction: $\hat{\mathbf{y}}^{\langle t\rangle} = \text{softmax}\left(\boldsymbol{W}_{\mathbf{y}}\mathbf{h}^{\langle t\rangle} + \mathbf{b}_{\mathbf{y}}\right)$

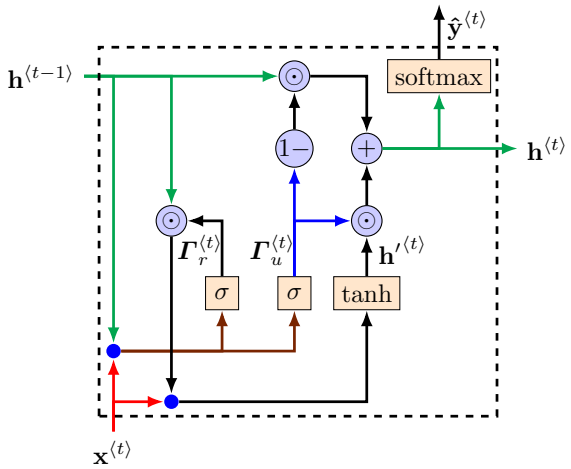


Figure 3.1: GRU Cell

2.3 Bidirectional RNN

$$\hat{\mathbf{y}}^{\langle t\rangle} = g\left(\boldsymbol{W}_{\mathbf{y}}\left[\vec{\mathbf{h}}^{\langle t\rangle}, \overleftarrow{\mathbf{h}}^{\langle t\rangle}\right] + \mathbf{b}_{\mathbf{y}}\right)$$

2.4 Deep RNNs

Steps:

1. Get hidden states for current layer:
$$\mathbf{h}^{[l]\langle t\rangle} = f^{[l]}\left(\boldsymbol{W}_{\mathbf{h}}^{[l]}\left[\mathbf{h}^{[l]\langle t-1\rangle}, \mathbf{a}^{[l-1]\langle t\rangle}\right] + \mathbf{b}_{\mathbf{h}}^{[l]}\right)$$
2. Pass the activations to the next layer:
$$\mathbf{a}^{[l]\langle t\rangle} = f^{[l]}\left(\boldsymbol{W}_{\mathbf{a}}^{[l]}\mathbf{h}^{[l]\langle t\rangle} + \mathbf{b}_{\mathbf{a}}^{[l]}\right)$$

A simpler architecture:

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

2.5 Perplexity Score

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

The *perplexity score* can be calculated as:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \boldsymbol{y}_i^{\langle t\rangle\intercal}\hat{\boldsymbol{y}}^{\langle t\rangle}}$$

$$\log(PP(W)) = -\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{y}_i^{\langle t\rangle\intercal}\log\left(\hat{\boldsymbol{y}}^{\langle t\rangle}\right)$$

where:
$\mathbf{y}^{\langle t\rangle}$ → target(label), encoded as one-hot vector.
$\hat{\mathbf{y}}^{\langle t\rangle}$ → prediction, a vector of probabilities.
W → 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 preplexity 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 $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$ -1 → 0
- Gradient clipping 32 → 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).

Calculations

Forget gate: $\boldsymbol{\Gamma}_f^{\langle t \rangle} = \sigma\left(\boldsymbol{W_f}\left[\mathbf{h}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}\right] + \mathbf{b_f}\right)$

Input gate: $\boldsymbol{\Gamma}_i^{\langle t \rangle} = \sigma\left(\boldsymbol{W_i}\left[\mathbf{h}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}\right] + \mathbf{b_i}\right)$

Cell state candidate: $\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\left(\boldsymbol{W_c}\left[\mathbf{h}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}\right] + \mathbf{b_c}\right)$

Ouptut gate: $\boldsymbol{\Gamma}_o^{\langle t \rangle} = \sigma\left(\boldsymbol{W_o}\left[\mathbf{h}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}\right] + \mathbf{b_o}\right)$

Cell state: $\mathbf{c}^{\langle t \rangle} = \boldsymbol{\Gamma}_i^{\langle t \rangle} \odot \tilde{\mathbf{c}}^{\langle t \rangle} + \boldsymbol{\Gamma}_f^{\langle t \rangle} \odot \mathbf{c}^{\langle t-1 \rangle}$

Output hidden state: $\mathbf{h}^{\langle t \rangle} = \boldsymbol{\Gamma}_o^{\langle t \rangle} \odot \tanh\left(\mathbf{c}^{\langle t \rangle}\right)$

Output prediction: $\hat{\mathbf{y}}^{\langle t \rangle} = \text{softmax}\left(\boldsymbol{W_y}\mathbf{h}^{\langle t \rangle} + \mathbf{b_y}\right)$

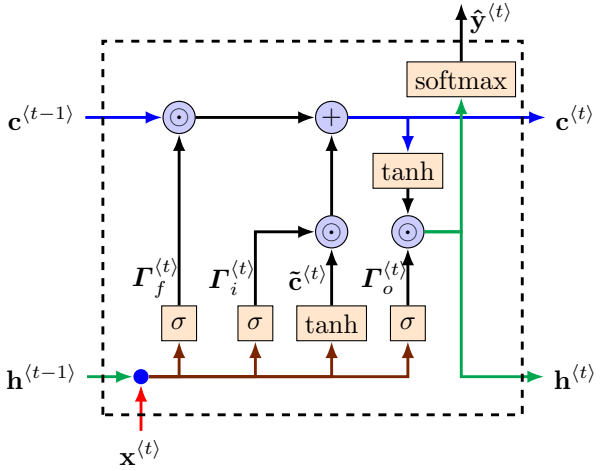


Figure 3.2: LSTM Cell

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	0	0	B-geo	0	B-tim
Sharon	flew	to	Miami	last	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?" ≠ "Where are you going?"
- Face recognition.

4.1 Model Architecture

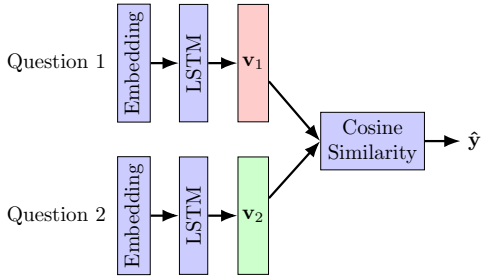


Figure 3.3: Siamese Network Architecture

4.2 Cosine Similarity

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

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

Compare *similarity score* against a threshold τ:

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

4.3 Triplet Loss Function

$\boldsymbol{A} \rightarrow$ Anchor.
$\boldsymbol{P} \rightarrow$ Positive example.
$\boldsymbol{N} \rightarrow$ Negative example.
$s \rightarrow$ Similarity function.

Example:

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

"How old are you?" $\rightarrow \boldsymbol{A}$ (Anchor)
"What is your age?" $\rightarrow \boldsymbol{P}$ (Positive) $s(\boldsymbol{A}, \boldsymbol{P}) \approx 1$
"Where are you from?" $\rightarrow \boldsymbol{N}$ (Negative) $s(\boldsymbol{A}, \boldsymbol{N}) \approx -1$

Simple loss:

$$\text{diff} = s(\boldsymbol{A}, \boldsymbol{N}) - s(\boldsymbol{A}, \boldsymbol{P}) \tag{3.3}$$

$$-2 \leq \text{diff} \leq 2$$

Non Linearity:

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

$$0 \leq \mathcal{L}(\boldsymbol{A}, \boldsymbol{P}, \boldsymbol{N}) \leq 2$$

Alpha margin:

$$\begin{aligned}\mathcal{L}(\boldsymbol{A}, \boldsymbol{P}, \boldsymbol{N}) &= \begin{cases} 0; & \text{if diff} + \alpha \leq 0 \\ \text{diff} + \alpha; & \text{if diff} + \alpha > 0 \end{cases} \\ &= \begin{cases} 0; & \text{if diff} \leq -\alpha \\ \text{diff} + \alpha; & \text{if diff} > -\alpha \end{cases} \\ &= \max(\text{diff} + \alpha, 0) \\ &\overset{(3.3)}{=} \max\left(s(\boldsymbol{A}, \boldsymbol{N}) - s(\boldsymbol{A}, \boldsymbol{P}) + \alpha, 0\right)\end{aligned} \tag{3.4}$$

$$0 \leq \mathcal{L}(\boldsymbol{A}, \boldsymbol{P}, \boldsymbol{N}) \leq 2 + \alpha$$

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

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

Triplet Selection

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

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

Harder to train. More to learn.

Hard Negative Mining

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

$$\text{mean}_{\text{neg}} = \frac{1}{b-1} \sum_{i} s\left(\boldsymbol{A}, \boldsymbol{N}^{(j)}\right)$$

Where $b \rightarrow$ batch size.
Closest negative: The value $s\left(\boldsymbol{A}, \boldsymbol{N}^{(j)}\right)$ closest to (but less than) the value $s\left(\boldsymbol{A}, \boldsymbol{P}\right)$. It is considered a *hard triplet* for training.

$$\mathcal{L}_1 \overset{(3.4)}{=} \max\left(\text{mean}_{\text{neg}} - s(\boldsymbol{A}, \boldsymbol{P}) + \alpha, 0\right)$$
$$\mathcal{L}_2 \overset{(3.4)}{=} \max\left(\text{closest}_{\text{neg}} - s(\boldsymbol{A}, \boldsymbol{P}) + \alpha, 0\right)$$

$$\mathcal{L}_{\text{Full}}(\boldsymbol{A}, \boldsymbol{P}, \boldsymbol{N}) = \mathcal{L}_1 + \mathcal{L}_2$$
$$\mathcal{J}_{\text{Full}} = \sum_{i=1}^{m} \mathcal{L}_{\text{Full}}\left(\boldsymbol{A}^{(i)}, \boldsymbol{P}^{(i)}, \boldsymbol{N}^{(i)}\right)$$

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 τ

Chapter 4
Attention Models

2 Text Summarization

2.1 Transformers vs RNNs

Problems with RNNs:

- Parallel computing is difficult to implement in RNNs.
- For long sequences there is loss of information.
- They suffer from the problem of vanishing gradients.

Transformers help with the problems of RNNs.
In transformers, to capture sequential information, *positional encoding* is used to retain the positional information of the input sequence. It outputs values to be added to the embeddings.

2.2 Transformer NLP applications

- Text summarization.
- Auto-Complete.
- Named entity recognition (NER).
- Question answering.
- Machine Translation.
- Chat-bots.

Examples of state of the art transformers:

- GPT-2: Generatie Pre-training for Transformer. Created by OpenAI. [Rad+19]
- BERT: Bidirectional Encoder Representation from Transformers. [Dev+19]
- T5: Text-to-text transfer transformer. [Raf+20]

T5 is a powerful multi-task transformer. It can perform tasks of translation, classification, question answering, regression and summarization.

2.3 Dot-Product Attention

Reference: [Vas+17]
Notation
L_Q : Length of query sentence (number of queries).
L_K : Length of key sentence (number of keys).
D : Embedding dimension.
$\mathbf{q}_i$ Embedding vector for the ith word of the query sentence.
$\mathbf{q}_i \in R^D$
$\mathbf{k}_i$ Embedding vector for the kth word of the key sentence.
$\mathbf{k}_j \in R^D$
$\boldsymbol{Q}$: Query matrix. $\boldsymbol{Q} \in \mathbb{R}^{L_Q \times D}$
$\boldsymbol{K}$: Key matrix. $\boldsymbol{K} \in \mathbb{R}^{L_K \times D}$
$\boldsymbol{V}$: Value matrix. $\boldsymbol{V} \in \mathbb{R}^{L_K \times D}$
$\boldsymbol{W}_A$: Attention weights (queries by keys matrix). Represents how much similar each key is to each query. $\boldsymbol{W}_A \in \mathbb{R}^{L_Q \times L_K}$
$\boldsymbol{Z}$: Attention output matrix. $\boldsymbol{Z} \in \mathbb{R}^{L_Q \times D}$
Example

- Query ($\boldsymbol{Q}$) : Embedding vectors for the words of the German sentence "Ich bin glücklich".
- Key ($\boldsymbol{K}$) : Embedding vectors for the words of the English sentence "I am happy"

$$\boldsymbol{W}_A = \text{softmax}\left(\boldsymbol{Q}\boldsymbol{K}^\intercal\right) \qquad [\text{similarity}\left(\mathbf{q}_i, \mathbf{k}_j\right)] \qquad (4.1)$$

Notes:

- $\mathbf{q}_i$ and $\mathbf{k}_j$ are similar if $\mathbf{q}_i \cdot \mathbf{k}_j$ is large.
- The softmax function is applied on the L_K dimension (per row) of the dot product to convert them to probabilities.
- Each query $\mathbf{q}_i$ picks the most similar $\mathbf{k}_j$

$$\begin{aligned} \boldsymbol{Z} &= \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) \\ &= \text{softmax}\left(\boldsymbol{Q}\boldsymbol{K}^\intercal\right)\boldsymbol{V} \\ &\overset{(4.1)}{=} \boldsymbol{W}_A\boldsymbol{V} \end{aligned}$$

See Figure 4.1a
Where Z_{ij} is the weighted sum. weighting each value $\mathbf{v}_i$ by the probability that the key $\mathbf{k}_i$ matches the query $\mathbf{q}_j$.
The attention mechanism calculates the dynamic or alignment weights $\boldsymbol{W}_A$ representing the relative importance of the inputs in the sequence $\{\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_{L_K}\}$, which are the keys, for that particular output $\mathbf{q}_j$, which is the query. Multiplying the dynamic weights or the alignments scores with the input sequence (the values $\boldsymbol{V}$) will then weight the sequence.

2.4 Types of Attention

- **Encoder/decoder attention:** One sentence (decoder) looks at another one (encoder).
- **Causal (self) attention** : Attention where in one sentence, words look at previous words (used for text generation).
- **Bi-directional self attention:** In one sentence, words look at both previous and future words.

2.5 Causal Attention

$\boldsymbol{M}$: Mask, a triangular matrix. It has 0s in the diagonal and below it, and $-\infty$ above it.

$$\boldsymbol{W}_A \overset{(4.1)}{=} \text{softmax}\left(\boldsymbol{Q}\boldsymbol{K}^\intercal + \boldsymbol{M}\right) \qquad [\text{similarity}\left(\mathbf{q}_i, \mathbf{k}_j, \text{mask out } j > i\right)]$$

Query $\mathbf{q}_i$ picks the most similar key $\mathbf{k}_j$, but only when $j < i$.
The mask prevents the model from attending to words in the future and restricts attention to the past.
Example of a mask:

$$\boldsymbol{M} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2.6 Scaled Dot-Product Attention

See Figure 4.1a

$$\boldsymbol{W}_A \overset{(4.1)}{=} \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\intercal}{\sqrt{d_k}}\right) \quad [\text{similarity}\,(\mathbf{q}_i, \mathbf{k}_j, \text{mask out } j > i)]$$

d_k is the dimension of queries and keys.

The scale factor $\frac{1}{\sqrt{d_k}}$ is used as normalization. It prevents the gradients from the function to be extremely small when large valuses of D are used.

2.7 Multi-Head Attention

See Figure 4.1b

When you use a multi-head attention, a head can learn different relationships between words from another head.

$$\begin{aligned}\boldsymbol{Z} &= \text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) \\ &= \text{Concat}(\boldsymbol{H}_1, \dots, \boldsymbol{H}_n)\boldsymbol{W}^O\end{aligned}$$

Where:

n : number of heads.

$\boldsymbol{H}_i = \text{Attention}\left(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V\right)$

Each head $\boldsymbol{H}_i$ is the output of the attention function of *Query*, *Key* and *Value* with trainable parameters $\left(\boldsymbol{W}_i^Q, \boldsymbol{W}_i^K, \boldsymbol{W}_i^V\right)$.

$\boldsymbol{H}_i \in \mathbb{R}^{L_Q \times D}$

$\boldsymbol{W}^O \in \mathbb{R}^{3D \times D}$

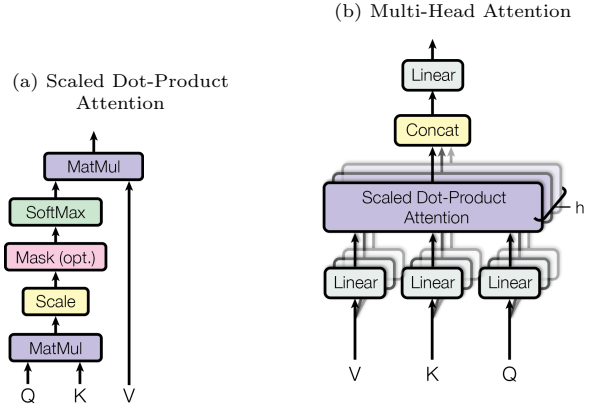


Figure 4.1: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.
Image source: [Vas+17]

2.8 The Transformer Model

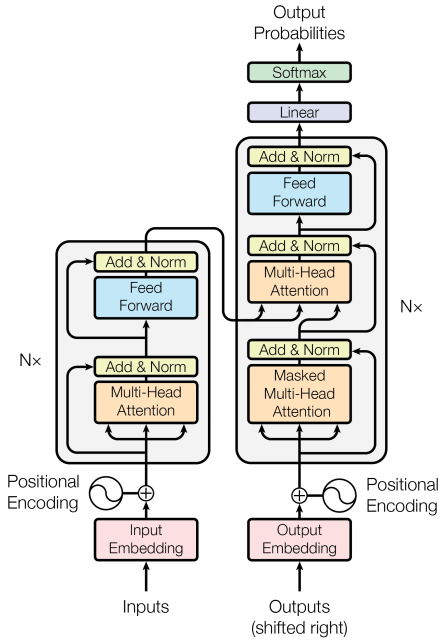


Figure 4.2: The Transformer - model architecture
Image source: [Vas+17]

4 Chatbot

4.1 Transformer issues

- Attention on sequence of length L takes L^2 time and memory as from (4.1) we can see that $\boldsymbol{Q}\boldsymbol{K}^\intercal \in \mathbb{R}^{L \times L}$.
- N layers take N times as much memory and activations need to be stored for backpropagation.

  GPT-3 has 96 layers and new models will have more (as of 2020).

4.2 Locality sensitive hashing (LSH) Attention

Reference: [And+15]

- Hashes $\boldsymbol{Q}$ and $\boldsymbol{K}$.
- Performs standard attention within the same hash bins.
- The process is repeated a few times to increase the probability of key in the same bin.

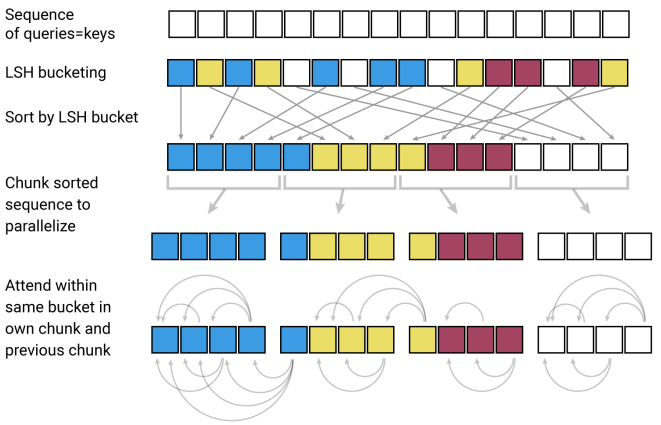


Figure 4.3: Simplified depiction of LSH Attention
Image source: [KKL20]

4.3 Reversible Residual Layers

Reference: [Gom+17]

The problem with running large deep models is that you will often run out of memory as each layer allocates memory to store activations for use in backpropagation. To save this resource, you need to be able to recompute these activations during the backward pass without storing them during the forward pass.This is where *reversible residual layers* come into play.

The standard approach requires one to store the outputs of each stage for use during backpropagation.

$$\mathbf{y}_a = \mathbf{x} + \text{Attention}(\mathbf{x}) \qquad \mathbf{y}_b = \mathbf{y}_a + \text{FFW}(\mathbf{y}_a)$$

By using the following setup, one need only store the outputs of the last stage, $\mathbf{y}_1$, $\mathbf{y}_2$.

$$\mathbf{y}_1 = \mathbf{x}_1 + \text{Attention}(\mathbf{x}_2) \qquad \mathbf{y}_2 = \mathbf{x}_2 + \text{FFW}(\mathbf{y}_1)$$

Using those values and running the algorithm in reverse, one can reproduce the values required for backpropagation.

$$\mathbf{x}_2 = \mathbf{y}_2 - \text{FFW}(\mathbf{y}_1)$$

$$\mathbf{x}_1 = \mathbf{y}_1 - \text{Attention}(\mathbf{x}_2)$$

This trades additional computation for memory space which is at a premium with the current generation of GPU's/TPU's.

4.4 Reformer

Reference: [KKL20]

The reformer is a transformer model designed to handle context windows of up to 1 million words. Reformer model solves the memory problems of transformer by implementing *locality sensitive hashing(LSH) Attention* and *reversible residual layers*.

https://github.com/FadyMorris/formula-sheets
DOI: 10.5281/zenodo.3987960

References

[And+15] Alexandr Andoni et al. "Practical and Optimal LSH for Angular Distance". In: *arXiv:1509.02897 [cs]* (Sept. 2015). arXiv: 1509.02897. URL: http://arxiv.org/abs/1509.02897 (visited on 12/05/2020).

[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).

[Gom+17] Aidan N. Gomez et al. "The Reversible Residual Network: Backpropagation Without Storing Activations". In: *arXiv:1707.04585 [cs]* (July 2017). arXiv: 1707.04585. URL: http://arxiv.org/abs/1707.04585 (visited on 12/06/2020).

[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).

[KKL20] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. "Reformer: The Efficient Transformer". In: *arXiv:2001.04451 [cs, stat]* (Feb. 2020). arXiv: 2001.04451. URL: http://arxiv.org/abs/2001.04451 (visited on 11/14/2020).

[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 (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. 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. 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/.

[Raf+20] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *arXiv:1910.10683 [cs, stat]* (July 2020). arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683 (visited on 11/14/2020).

[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.

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". In: *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762 (visited on 11/14/2020).

[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).