Lecture 3

What is FastText

What is FastText

- fastText is an open-source library, developed by the Facebook AI Research lab. Its main focus is on achieving scalable solutions for the tasks of text classification and representation while processing large datasets quickly and accurately.

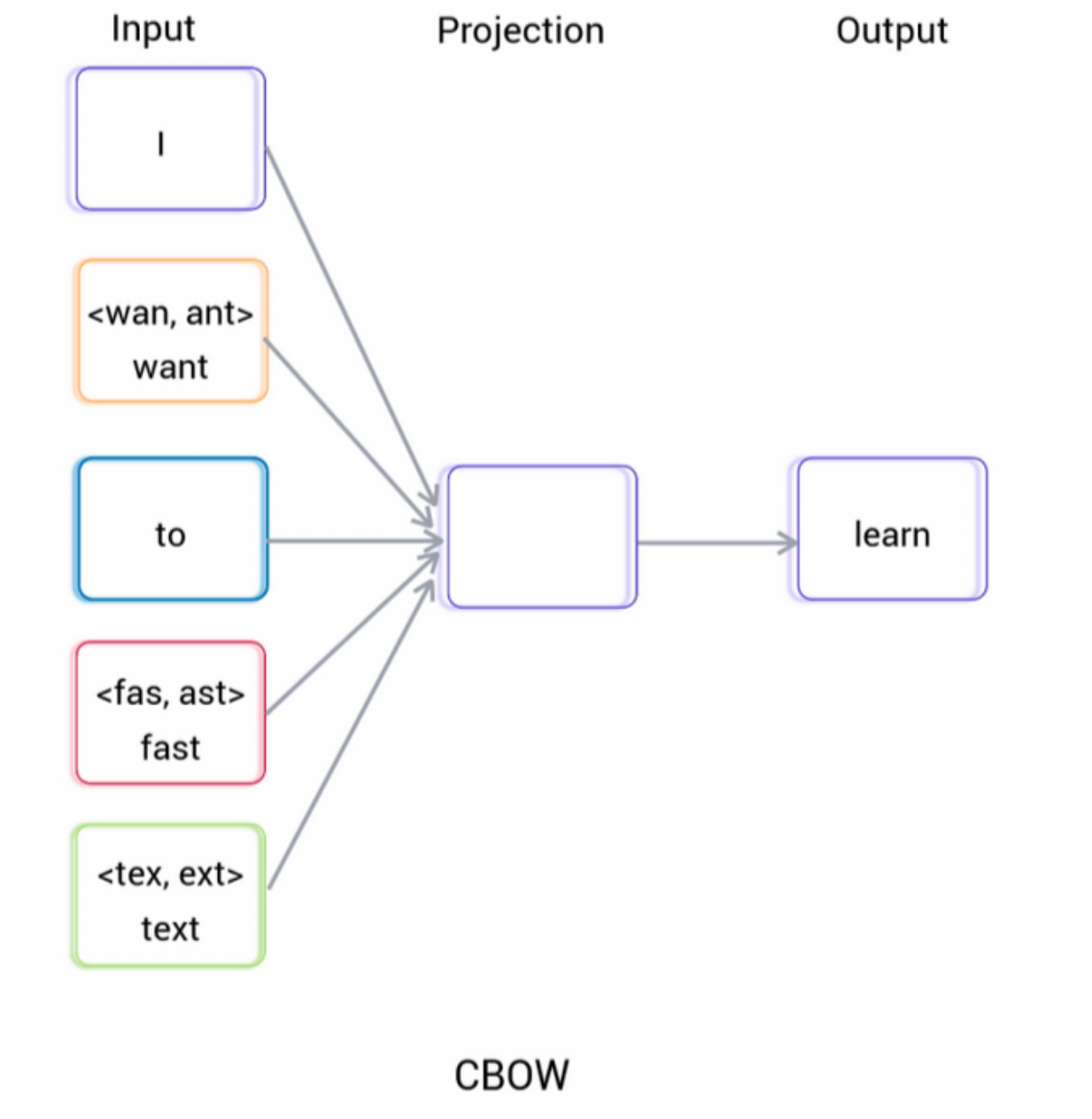
- FastText is a modified version of word2vec.

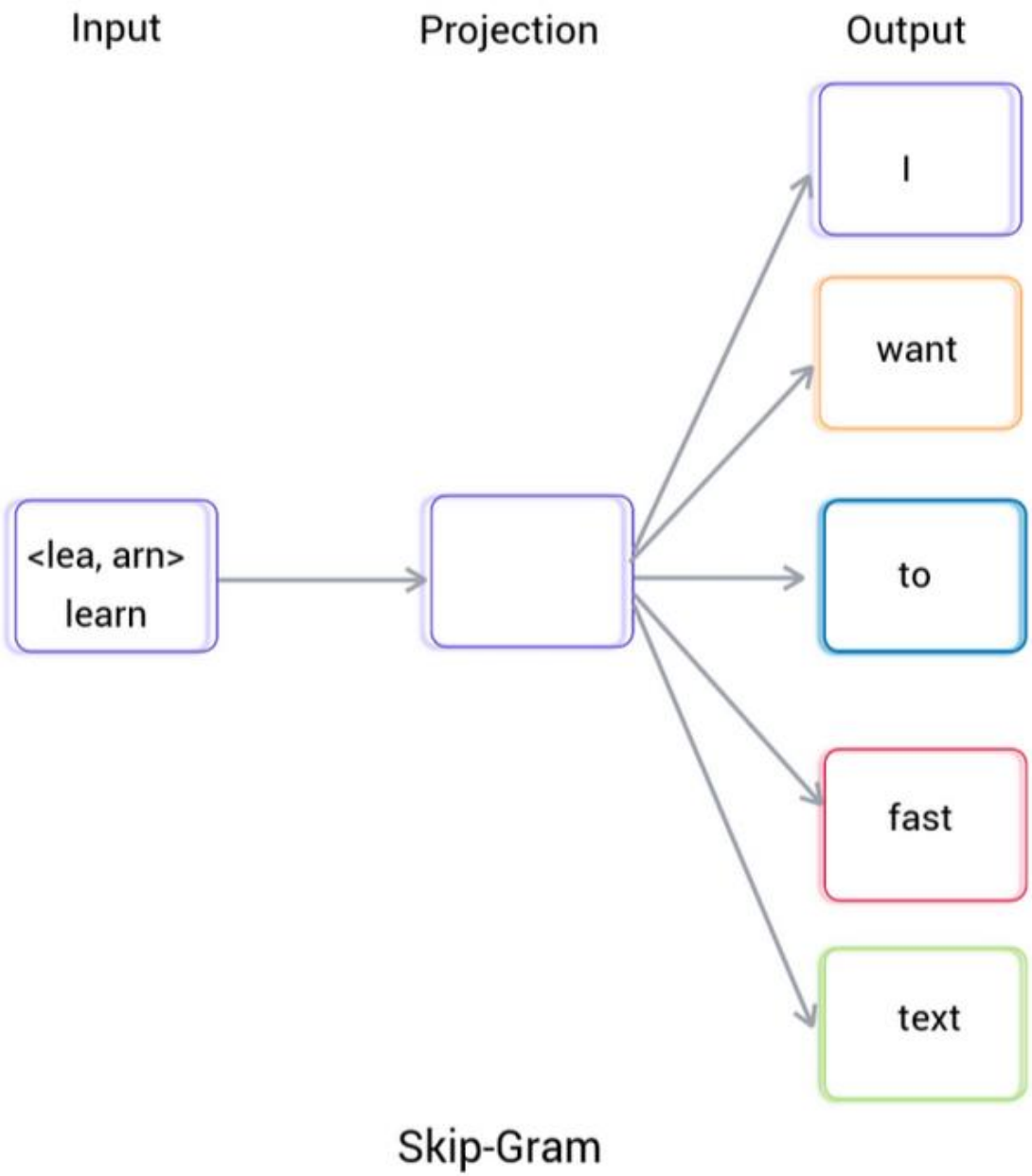
Why Should you Use FastText?

Why Should you Use FastText?

- To improve vector representation for morphologically rich language, FastText provides embeddings for character n-grams, representing words as the average of these embeddings.
- It is an extension of the word2vec model. Word2Vec model provides embedding to the words, whereas fastText provides embeddings to the character n-grams.

- Like the word2vec model, fastText uses CBOW and Skip-gram to compute the vectors.
- FastText can also handle out-of-vocabulary words, i.e., the fasttext can find the word embeddings that are not present at the time of training.


Input
Projection
Output
I
<wan, ant>
want
to
<fas, ast>
fast
<tex, ext>
text
learn
CBOW

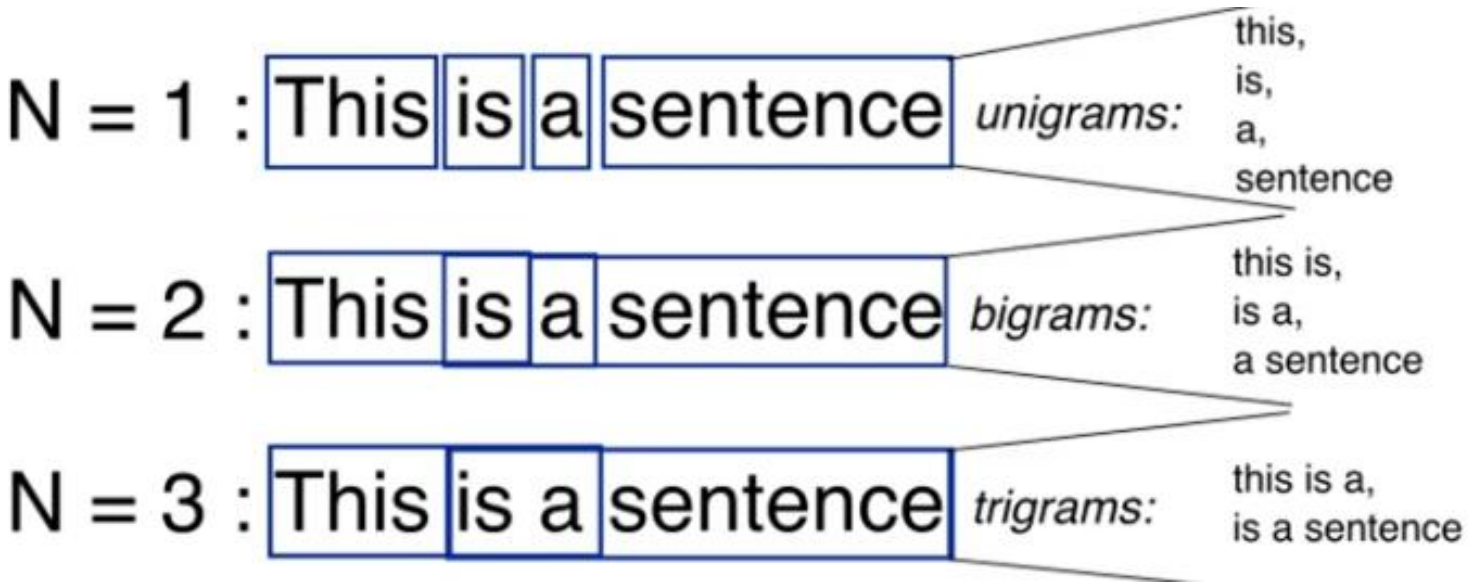


Skip-Gram

N-gram

N-gram

- Is a sequence of N words
- “Natural Language ” is a 2-gram “bigram”
- “Natural Language course ” is a 3-gram “trigram”
- “A Natural Language course ” is a 4-gram


N = 1 : This is a sentence
unigrams:
this,
is,
a,
sentence
N = 2 : This is a sentence
bigrams:
this is,
is a,
a sentence
N = 3 : This is a sentence
trigrams:
this is a,
is a sentence

How many N-grams in a sentence ?

If X=Num of words in a given sentence k, N is the number of required gram.
The number of n-grams for sentence K would be:

$$n - grams_k = X - (N - 1)$$

Why N-gram?

- auto completion of sentences
- Auto spell check
- Grammer

Problems with N-gram

- Too many features
- OOV words : These words that appear during testing but not in training.

Solution :

- We shouldn't increase N-grams or take one of them only.
- We should use more than one

(unigram , bigram , trigram)

Remove Some N-grams

- Remove some N-Grams from features based on their occurrence frequency in documents of our corpus.
- **High Frequency N-grams**
  - Articles, prepositions etc. (and, a, the, is)
  - They are called ***stop words***; they won't help us to discriminate texts.
  - ***Remove*** these stop words
- **Low frequency N-grams**
  - ***Typos***, rare n-grams
  - We don't need them either, otherwise we will likely ***overfit***
- **Medium Frequency N-grams**
  - Those are ***good*** n-grams

Character n-gram

Character n-gram

Character n-gram is the contiguous sequence of n items from a given sample of a character or word. It may be bigram, trigram, etc. For example character **trigram (n = 3) of the word "where"** will be:

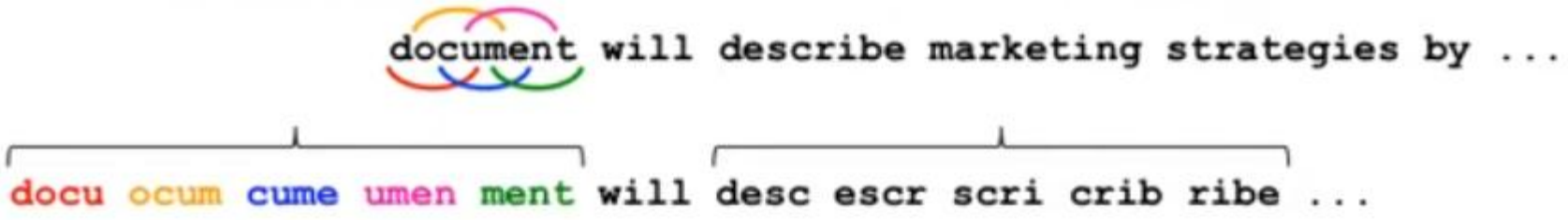
<wh, whe, her, ere, re>

- In FastText architecture, they have also included word itself with character n-gram. That means input data to the model for the word “where” will be:

  <wh, whe, her, ere, re> and **<where>**

Why character N-gram

- Building a good stemmer is hard
- Cheap alternative:
  - take every n-character substring of the word
  - related words ➔ many of the same n-grams
  - n=4,5 works well for European languages



Stemming vs. character N-grams

Language	# Docs	Words	Morfessor	Snowball	4-grams
Bulgarian	85,427	0.2195	0.2786 (+26.9%)		0.3163 (+44.1%)
Czech	81,735	0.2270	0.3215 (+41.6%)		0.3294 (+45.1%)
Dutch	190,605	0.4162	0.4274 (+2.7%)	0.4273 (+2.7%)	0.4378 (+4.9%)
English	166,754	0.4829	0.4265 (-11.7%)	0.5008 (+3.7%)	0.4411 (-8.7%)
Finnish	55,344	0.3191	0.3846 (+20.5%)	0.4173 (+30.7%)	0.4827 (+51.3%)
French	129,804	0.4267	0.4231 (-0.84%)	0.4558 (+6.8)%)	0.4442 (+4.1%)
German	294,805	0.3489	0.4122 (+18.1%)	0.3842 (+10.1%)	0.4281 (+22.7%)
Hungarian	49,530	0.1979	0.2932 (+48.2%)		0.3549 (+79.3%)
Italian	157,558	0.3950	0.3770 (-4.6%)	0.4350 (+10.1%)	0.3925 (-0.6%)
Portuguese	210,734	0.3232	0.3403 (+5.3%)		0.3316 (+2.6%)
Russian	16,715	0.2671	0.3307 (+23.8%)		0.3406 (+27.5%)
Spanish	454,041	0.4265	0.4230 (-0.82%)	0.4671 (+9.5%)	0.4465 (+4.7%)
Swedish	142,819	0.3387	0.3738 (+10.4%)		0.4236 (+25.1%)
Average		**0.3376**	**0.3701 (+9.6%)**	**0.3614 (+7.0%)**	**0.3976 (+17.7%)**

- **Words**: Using words as units.
- **Morfessor:** A morphological segmentation model (percentage increase over words is shown).
- **Snowball:** A stemming algorithm (percentage increase over words is shown).
- **4-grams:**Character-based 4-gram segmentation (percentage increase over words is shown).

4-grams consistently provide the highest improvement across all languages (+17.7% on average), making them the most effective approach in this comparison.

What is the Difference Between fastText and Word2Vec?

What is the Difference Between fastText and Word2Vec?

- Word2Vec works on the word level, while fastText works on the character n-grams.
- Word2Vec cannot provide embeddings for out-of-vocabulary words, while fastText can provide embeddings for OOV words.
- FastText can provide better embeddings for morphologically rich languages compared to word2vec.
- FastText uses the hierarchical classifier to train the model; hence it is faster than word2vec.

What is the Difference Between fastText and Word2Vec?

Aspect	Word2Vec	fastText
Handling of OOV Words	Struggles with OOV words	Handles OOV words efficiently using subword embeddings
Representation of Words	Generates embeddings solely based on words	Considers subword information for richer representations
Training Efficiency	Training speed is moderate	Exceptional speed and scalability, especially with large datasets
Use Cases	Well-suited for finding word relationships and semantic similarities	Preferred for handling OOV words, sentiment analysis, and understanding morphology
Word-Level vs. Subword-Level	Operates at the word level	Considers subword units for understanding word meanings and morphology

Fasttext is a solution for classification with a large number of labels

- It is very fast in training word vector models. You can train about 1 billion words in less than 10 minutes.
- The models built through deep neural networks can be slow to train and test. These methods use a linear classifier to train the model.

Linear classifier

- Text and labels are represented as vectors.
- Vector representations such that text and it's associated labels have similar vectors.
- In simple words, the vector corresponding to the text is closer to its corresponding label.

- To find the probability score of a correct label given it's associated text we use the **softmax function:**

$$softmax(w_{travel}) = \frac{e^{w_{car}^T * w_{travel}}}{\sum_{lelabels} e^{w_{car}^T * w_l}}$$

- Here travel is the label and car is the text associated to it.

Let's say we have a neural network that classifies images into three categories:

- **Cat**
- **Dog**
- **Rabbit**

The model's final layer produces three raw scores (logits), e.g.:

$$z=[2.0,1.0,0.1]$$

If we apply **sigmoid** independently to each class:

$$\sigma(z_i) = \frac{1}{1 + e^{-z_i}}$$

Applying sigmoid:

$$\sigma(2.0) = \frac{1}{1 + e^{-2.0}} \approx 0.88$$

$$\sigma(1.0) = \frac{1}{1 + e^{-1.0}} \approx 0.73$$

$$\sigma(0.1) = \frac{1}{1 + e^{-0.1}} \approx 0.52$$

So, the model outputs:

$$[0.88, 0.73, 0.52]$$

Problem:

- These are **not valid probabilities**, since their sum is **0.88 + 0.73 + 0.52 = 2.13** (which is greater than 1).

Using Softmax (Correct for Multi-class)

The softmax function:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

First, calculate exponentials:

$$e^{2.0} \approx 7.39, \quad e^{1.0} \approx 2.72, \quad e^{0.1} \approx 1.11$$

Sum of exponentials:

$$7.39 + 2.72 + 1.11 = 11.22$$

Now, apply softmax:

$$\text{softmax}(2.0) = \frac{7.39}{11.22} \approx 0.66$$

$$\text{softmax}(1.0) = \frac{2.72}{11.22} \approx 0.24$$

$$\text{softmax}(0.1) = \frac{1.11}{11.22} \approx 0.10$$

Final softmax output:

$$[0.66, 0.24, 0.10]$$

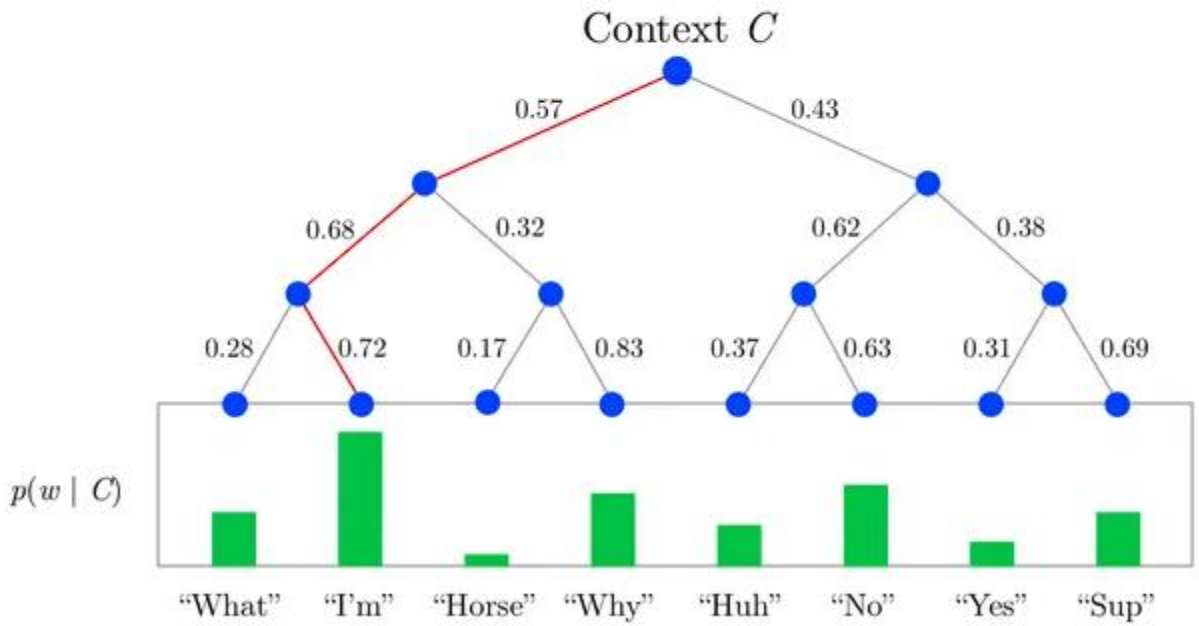
Why is Softmax Correct?

- The values **sum to 1**.
- The model assigns the highest probability to "Cat" (0.66), making a clear decision.
- It ensures a **valid probability distribution** over the classes.

- This is quite computationally expensive because for every piece of text not only we have to get the score associated with its correct label but we need to get the score for every other label in the training set. This limits the use of these models on very large datasets.

- FastText solves this problem by using a **hierarchical classifier** to train the model.
- The idea is to build a binary tree whose leaves correspond to the labels. Each intermediate node has a binary decision activation (e.g. sigmoid) that is trained, and predicts if we should go to the left or to the right.
- The probability of the output unit is then given by the product of the probabilities of intermediate nodes along the path from the root to the output unit leave.

Hierarchical softmax



- *FastText* uses the *Huffman* algorithm to build these trees.
- For multi-class classification with a large number of classes, this approach results in significant gain in efficiency compared to other models.

Example

frequency	word
50000	“the”
20000	“is”
5000	“fast”
1000	“text”
300	“learning”

First step : fasttext will compute the frequency for every word appeared.
The words that appeared with the largest frequency will have the shortest path but the words that appeared with the lowest frequency will have the shortest path.

- “the” will take the shortest path.
- “learning” will take the longest path.
- Second step:

  - Start with all words as leaf nodes.

  - Pick the two words with the lowest frequency and combine them into a new node.

  The new node’s frequency = sum of the two words’ frequencies.

  - **Repeat until all words are merged into a single tree.**

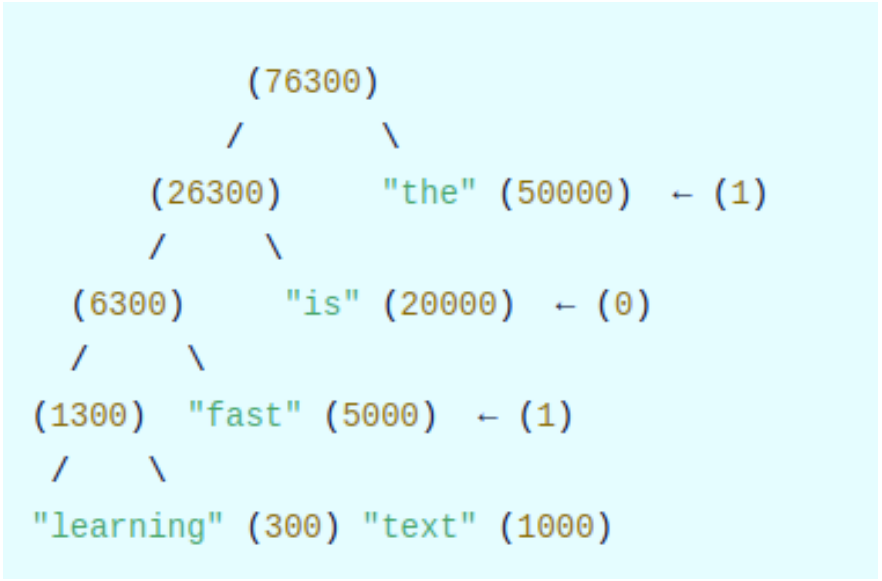

(76300)
(26300)
"the" (50000)
(6300)
"is" (20000)
(1300)
"fast" (5000)
"learning" (300)
"text" (1000)

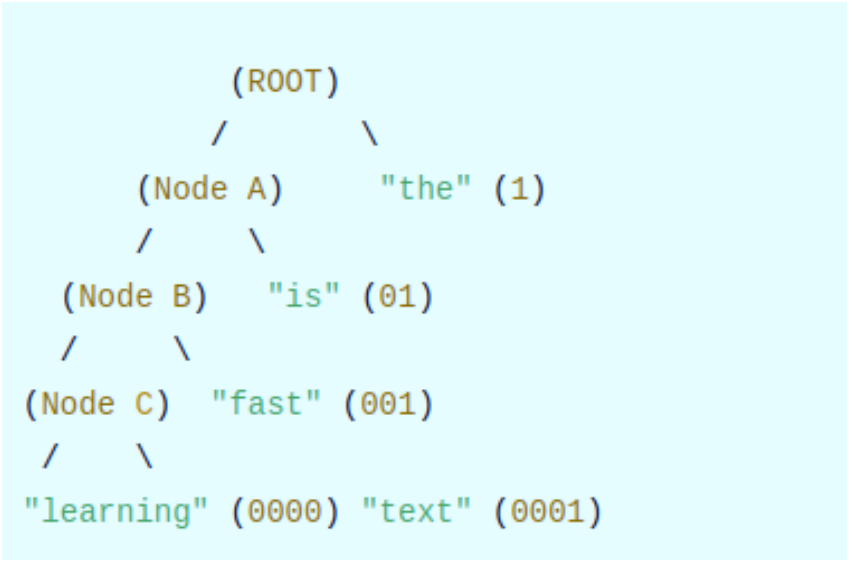
Example of Predicting a Word “fast“

- Start at the **root**
- Compute **sigmoid probability** at each node:

  If **probability > 0.5**, go **right (1)**

  If **probability < 0.5**, go **left (0)**
- Continue until reaching the leaf node for "fast"





➡ **Step 1:** Start at the root
Compute **sigmoid probability P at Node A**
Suppose P=0.3 (**less than 0.5**) → **Go left (0)**

➡ **Step 2:** Now at **Node B**
Compute **sigmoid probability P at Node B**
Suppose P=0.4 (**less than 0.5**) → **Go left (0)**

➡ **Step 3:** Now at **Node C**
Compute **sigmoid probability P at Node C**
Suppose P=0.7 (**greater than 0.5**) → **Go right (1)**

- **Each internal node has a binary classifier trained with sigmoid**
- **Sigmoid probability determines whether to go left (0) or right (1)**
- **FastText follows the Huffman path to reach the correct word**
- **This is much more efficient than computing probabilities for all words!**

softmax

Word	Logit (z)
"cat"	1.2
"dog"	0.8
"fish"	2.0
"bird"	-0.5

- **Step 1: Compute e^z for each word**

$$e^{1.2} = 3.3201, \quad e^{0.8} = 2.2255, \quad e^{2.0} = 7.3891, \quad e^{-0.5} = 0.6065$$

- **Step 2: Compute the sum of exponentials**

$$\sum e^z = 3.3201 + 2.2255 + 7.3891 + 0.6065 = 13.5412$$

- **Step 3: Compute final probabilities for each word**

$$P(\text{"cat"}) = \frac{3.3201}{13.5412} = 0.2452$$

$$P(\text{"dog"}) = \frac{2.2255}{13.5412} = 0.1644$$

$$P(\text{"fish"}) = \frac{7.3891}{13.5412} = 0.5458$$

$$P(\text{"bird"}) = \frac{0.6065}{13.5412} = 0.0447$$

- **P("fish") = 0.5458** (most likely word)
- **P("cat") = 0.2452**
- **P("dog") = 0.1644**
- **P("bird") = 0.0447** (least likely word)

Hierarchical softmax

Node	Logit (z)
Root → N1	1.2
Root → N2	-1.2
N1 → "cat"	0.8
N1 → "dog"	-0.8
N2 → "fish"	2.0
N2 → "bird"	-2.0

We calculate **Sigmoid** at each step.

To get **"fish"**, we follow the path:

1. **Root → N2**
2. **N2 → "fish"**

Using **Sigmoid**:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$P(N2) = \sigma(-1.2) = \frac{1}{1+e^{1.2}} = 0.2315$$

$$P(\text{"fish"}|N2) = \sigma(2.0) = \frac{1}{1+e^{-2.0}} = 0.8808$$

$$P(\text{"fish"}) = P(N2) \times P(\text{"fish"}|N2) = 0.2315 \times 0.8808 = 0.2039$$

- **Softmax:** $P(\text{"fish"}) = 0.5458$
- **Hierarchical Softmax:** $P(\text{"fish"}) = 0.2039$

🔹 **Hierarchical Softmax does not directly match Softmax values** because it depends on the **binary tree structure**, unlike Softmax, which computes probabilities for all words at once.

🔹 However, **Hierarchical Softmax is computationally much faster**, making it **more efficient** when dealing with a **large vocabulary**.

Uses of FastText:

- It is used for finding semantic similarities
- It can also be used for text classification(ex: spam filtering).
- It can train large datasets in minutes.

Thanks