What is Word Embedding?
Word Embedding is the technique to represent the word in vector space for capturing the syntactic and semantic similarities within the words.

Why Do We Need Them?
We need them to represent the words in vector space where the vector distance of similar words will be shorter than the dissimilar words. That means similar words will be closely grouped in vector space.

There is an interesting example of the necessity of word embedding in the following link.
https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

What is word2vec?
Word2vec is the most popular technique to learn word embedding which was first introduced by Mr. Tomas Mikolov in 2013.
Word2vec-

- Is a two-layer neural network to generate word embedding for a given text corpus.
- Is mapping of words in a vector space.
- Can operate addition, subtraction, and can calculate the distance within the vectors. For example, King - Man + Woman = Queen
- Can preserve the relationship with the vectors.

Now I'll shortly summarize the paper titled, "**Efficient Estimation of Word Representations in Vector Space"** by **Tomas Mikolov.**
Summarization:

In this paper, we shall learn the syntactic and semantic similarities of words and the embedding of words.
Example of syntactic similarities: Ex. Big and Bigger. Small and Smaller.
Example of semantic similarities: Man:King::Women: Queen

The main goal of the paper is to create word embedding efficiently. For creating the word embedding efficiently, the author proposed two models.

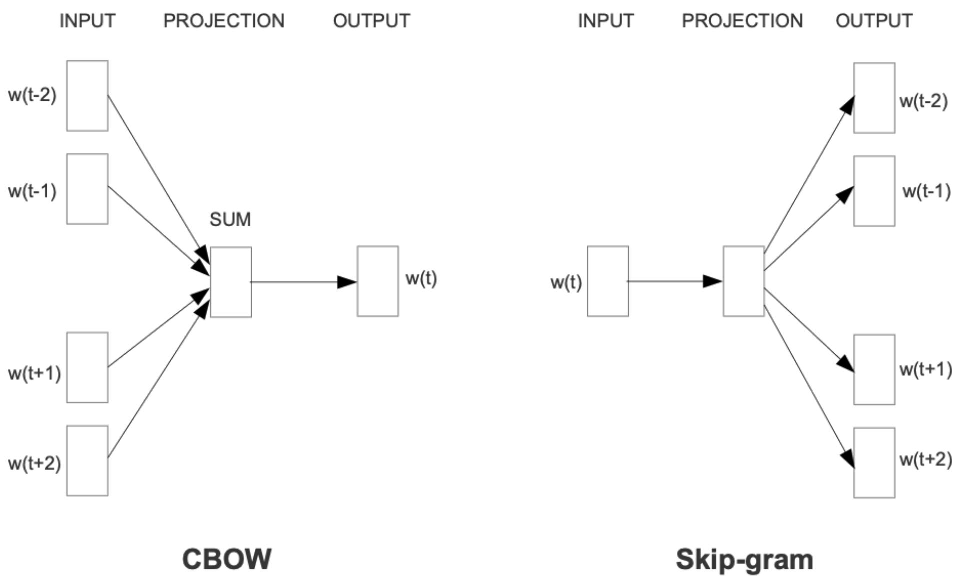


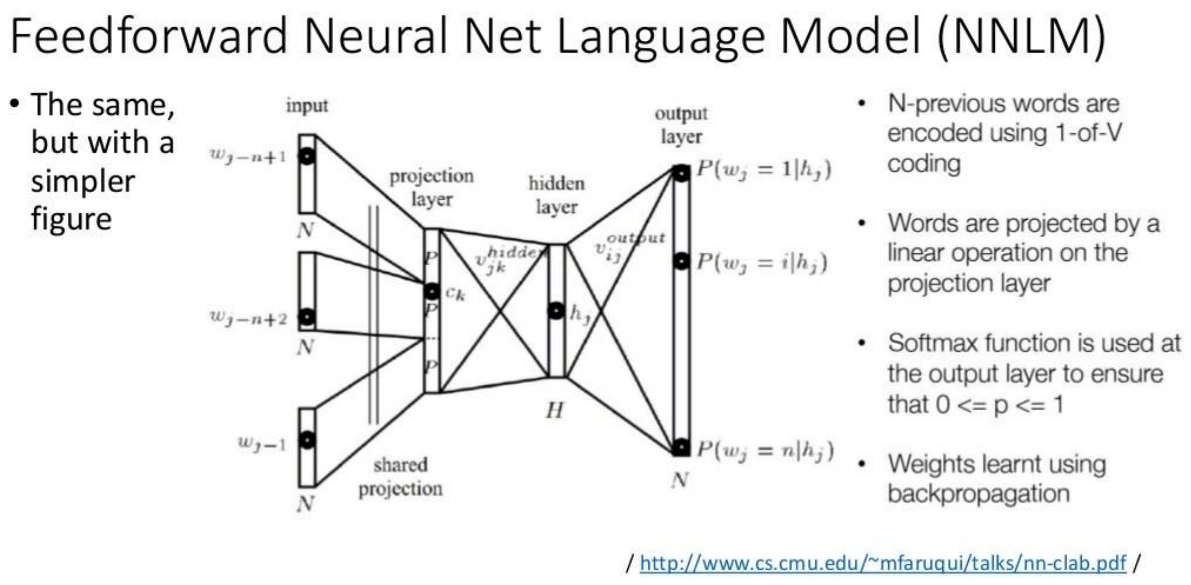
Fig: 1
Source: https://arxiv.org/pdf/1301.3781.pdf

Continuous Bag-of-Word Model (CBOW):

Predict the target word by looking at context words(surrounding words).

Continuous Skip-gram: To look at the target words for finding the context words.

Feedforward Neural Net Language Model (NNLM):



Feedforward Neural Net Language Model (NNLM)

- The same, but with a simpler figure

- N-previous words are encoded using 1-of-V coding
- Words are projected by a linear operation on the projection layer
- Softmax function is used at the output layer to ensure that 0 <= p <= 1
- Weights learnt using backpropagation

/ http://www.cs.cmu.edu/~mfaruqui/talks/nn-clab.pdf /

Fig-2: NNLM Model

The cost of the NNLM Model can be calculated by-
Q = NxD + NxDxH + HxV
Where,
P = NxD = Cost of creating projection layer
NxDxH = Cost of projecting to Hidden layer
HxV = Cos of calculating the output layer from the hidden layer

N -> number of word
V -> size vocabulary
D -> Dimetinos of project layer
H -> Hidden layer

The dominating term is the HxV. Because the size of the vocabulary is huge.

HxV can be minimized by using hierarchical softmax or by using a binary tree.
Thus the maximum complexity is caused by the term NxDxH.
This paper proposed an architecture that doesn't have a hidden layer. The efficiency of that model is dependent on the softmax normalization.

New Proposed Architectures:
They proposed two new model architectures for creating the word embedding. Which are computationally less expensive. According to the previous section, complexity is caused by the non-linear hidden layer and the neural network can be trained in two steps.
First, continuous word vectors can be learned from a simple model, and second, n-gram NNLM can be trained on that. That means if a model is trained to predict a word, again that model can be trained to predict another new word.

Details fo CBOW and Skip-gram Model:
Continuous Bag-of-Words Model (CBOW):
CBOW model is similar to feedforward **NNLM**. But in the **CBOW** model, instead of using the no-linear hidden layer, uses a projection layer that is shared by all of the words. **"That means the project layer is a single set of shared weights without any activation function".** Moreover, instead of taking the previous **window_size** words from the past to predict the next word, it uses the **window_size** words from the past and **window_size** words from the future to predict the **center word**. In this paper, the center word is called the target word, and surrounds **window_size*2** words are called context words.

The author is calling this architecture the Continuous Bag-of-Words Model because the order of the context words does not influence the projection.

Continuous Skip-Gram:

This is the exact opposite of the CBOW model. In Skip-gram we try to predict the context words for a target word. So, our input will be the target word and our output will be the context words.

Data processing:
Filtering the corpus:
Let's take an example for understanding data processing. We are assuming that there is just a single sentence in the text corpus.

Sentence: **"The 5 quick brown / fox jumps <b> over the lazy dog and lazy cat."**
Then I have

- Removed the **stop words**.
- Removed the word if it is not an **alpha**.
- Removed the word if the number of characters is less than **one**.
- Removed sentences if the word count is greater than **max_word_count(100)** or less than **min_word_count(5).**
- Converted the sentences to words.

After filtering and converting the sentences to words, we shall get the

[['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog', 'lazy', 'cat']]

Then created a vocabulary with the frequency count of the words.

Vocabulary with frequency count: **{'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'lazy': 2, 'dog': 1, 'cat': 1}**

Then sorted the vocabulary based on their frequency count in descending order to remove the words which are less frequent.

Sorted vocabulary: **{'lazy': 2, 'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'dog': 1, 'cat': 1}**

I have noticed that training accuracy heavily depends on the frequency count. If we remove the less frequent words from the vocabulary, the accuracy increases highly.

As I have taken a single sequence as an example, so let's assume that we are removing the words from the dictionary which has the frequency count is **0 (zero):D.** In the actual implementation, we won't find any word in the vocabulary which frequency count is zero **0 (zero).**

After filtering out the less frequent word, we will get the following dictionary.

{'lazy': 2, 'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'dog': 1, 'cat': 1}

As of now, we have removed some words from the dictionary, so we have to remove the word from our word corpus which is not available in the vocabulary.

In our example, we didn’t remove any word from the dictionary, so our word corpus will remain the same as before.

So, our word corpus is **[['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog', 'lazy', 'cat']].**

Then, I have assigned a unique id to each of the words in the vocabulary. And created two dictionary **word_to_id, id_to_word.** Dictionary, **word_to_id** was created to get word for a word id, and **id_to_word** was created to get the word for a word id.

Word_to_id: **[(0, 'PAD'), ('lazy', 1), ('quick', 2), ('brown', 3), ('fox', 4), ('jumps', 5), ('dog', 6), ('cat', 7)]**

Id_to_word: **[('PAD', 0), (1, 'lazy'), (2, 'quick'), (3, 'brown'), (4, 'fox'), (5, 'jumps'), (6, 'dog'), (7, 'cat')]**

Finally, it’s time to express the word corpus by the unique id. So, I have expressed the words by their **word_id**.

So, **[['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog', 'lazy', 'cat']]** will be converted to **[[2, 3, 4, 5, 1, 6, 1, 7]]**

Expressing the Word Id by a One-hot Encoded Vector:
In both models, input and output will be the one-hot encoded vector. For converting the word id to a one-hot encoded vector I have created a **V x V** dimensional matrix. Where V is the number of words in the vocabulary. Every value of this matrix will be zero except the value of diagonal. The value of the diagonal will be 1. So, if we want to convert the word id to a one-hot vector, we have to take only a row or a column of that matrix in which word id is equal to the row or column number.

One-hot matrix:

```
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
```

Then I have converted the target and context id into a one-hot encoded vector.

I have also summed up all of the one-hot encoded **context_ids** vectors to a single one-hot encoded vector of size **vocabulary_size**. So, there is one vector of size **vocabulary_size**, for containing all of the **context_ids** of a **target_id**. Because in the **CBOW** model, at the input layer, and in the **skip-gram** model at the output layer, I have to sum up the one-hot context vectors. Here I have just precalculated it.

Example after converting the word_id to a one-hot encoded vector and calculating the sum of the context vectors for a target vector.

```
------------------------------------
target-> word: quick ,id: 2
one-hot encoded target-> [0. 0. 1. 0. 0. 0. 0. 0.]
context-> word: brown ,id: 3
one-hot encoded context-> [0. 0. 0. 1. 0. 0. 0. 0.]
context-> word: fox ,id: 40
one-hot encoded context-> [0. 0. 0. 0. 1. 0. 0. 0.]
sum of context vector-> [0. 0. 0. 1. 1. 0. 0. 0.]
------------------------------------
target-> word: brown ,id: 3
one-hot encoded target-> [0. 0. 0. 1. 0. 0. 0. 0.]
context-> word: quick ,id: 2
one-hot encoded context-> [0. 0. 1. 0. 0. 0. 0. 0.]
context-> word: fox ,id: 4
one-hot encoded context-> [0. 0. 0. 0. 1. 0. 0. 0.]
context-> word: jumps ,id: 5
one-hot encoded context-> [0. 0. 0. 0. 0. 1. 0. 0.]
sum of context vector-> [0. 0. 1. 0. 1. 1. 0. 0.]
```

CBOW Model Architecture:
CBOW model tries to predict the target word for its surrounding context words. CBOW model architecture is shown in Fig-3: for multiple context vectors.

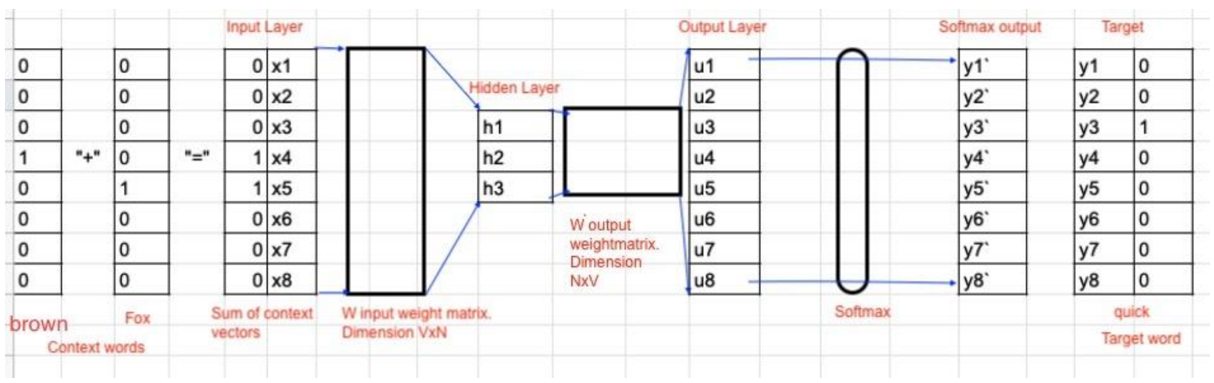


Fig-3: A simplified version of the CBOW model.

Where **{x1k, x2k,...xCk}** is the input vector of context words of dimension **1 x V,** where **V** is the vocabulary size. In our example, the size of **V** is **8**. As we have already calculated the sum of the input vector, so in our case, there will be just one input vector. So, let's assume that our input vector is **{x1, x2...x8}.** Moreover, in our example, the value of **C** is **4**. Because our **window_size** is **2**.

W is the weight matrix in the input and hidden layer with dimension **V x N**. Where **N** is the hidden layer size. Let's assume that **N** is **3**.
Let's, **W** =

w11	w12	w13
w21	w22	w23
w31	w32	w33
w41	w42	w43
w51	w52	w53
w61	w62	w63
w71	w72	w73
w81	w82	w83

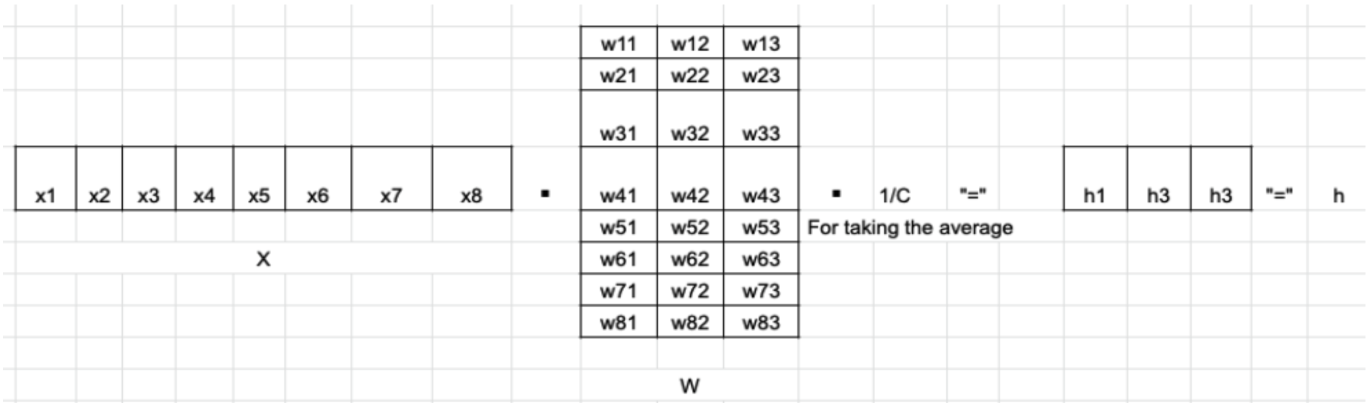
h is the output of the average of **X.W** with dimension **1xN.**

W`** is the weight matrix between the hidden layer and output layer with dimension **N x V.** Let's take **W` =

w\`11	w\`12	w\`13	w\`14	w\`15	w\`16	w\`17	w\`18
w\`21	w\`22	w\`23	w\`24	w\`25	w\`26	w\`27	w\`28
w\`31	w\`32	w\`33	w\`34	w\`35	w\`36	w\`37	w\`38

u is the output of **h. W`** with dimension **1xV**.
y` is the output of **Softmax(u).**
y is the actual target**.**

Training:
Forward propagation:
Calculating hidden layer h.
Hidden layer matrix**, h = X.W/C**. Here I have divided the **X.W** by **C** for taking the average.

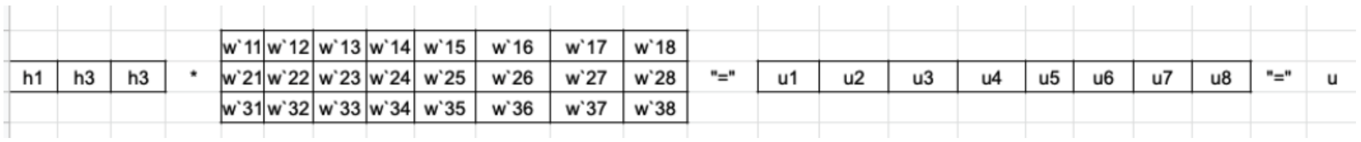


$$\underbrace{\begin{bmatrix} x1 & x2 & x3 & x4 & x5 & x6 & x7 & x8 \end{bmatrix}}_{X} \cdot \underbrace{\begin{bmatrix} w11 & w12 & w13 \\ w21 & w22 & w23 \\ w31 & w32 & w33 \\ w41 & w42 & w43 \\ w51 & w52 & w53 \\ w61 & w62 & w63 \\ w71 & w72 & w73 \\ w81 & w82 & w83 \end{bmatrix}}_{W} \cdot \text{1/C} \text{ "=" } \begin{bmatrix} h1 & h3 & h3 \end{bmatrix} \text{ "=" } h$$

For taking the average

So,
 h1 = w11x1 + w21x2 + w31x3 + w41x4 + w51x5 + w61x6 + w71x7 + w81x8
…...
 h3 = w13x1 + w23x2 + w33x3 + w43x4 + w53x5 + w63x6 + w73x7 + w83x8

Calculating output layer matrix (u):



$$\begin{bmatrix} h1 & h3 & h3 \end{bmatrix} * \begin{bmatrix} w`11 & w`12 & w`13 & w`14 & w`15 & w`16 & w`17 & w`18 \\ w`21 & w`22 & w`23 & w`24 & w`25 & w`26 & w`27 & w`28 \\ w`31 & w`32 & w`33 & w`34 & w`35 & w`36 & w`37 & w`38 \end{bmatrix} \text{ "=" } \begin{bmatrix} u1 & u2 & u3 & u4 & u5 & u6 & u7 & u8 \end{bmatrix} \text{ "=" } u$$

u1 = w`11h1 + w`21h2 + w`31h3
……..
u8 = w`15h1 + w`25h2 + w`35h3

Calculating softmax:
Softmax is used for calculating the probability of the target for the context word.
y1 = e^u1/(e^u1+..+e^u8)
y2 = e^u2/(e^u1+..+e^u8)
…..
y8 = e^u8/(e^u1+..+e^u8)

Error calculation:
Now we need to calculate the error for checking the performance of our model and how accurately updating the **W** and **W`**.

The following function is used for calculating the error/loss.

$$\begin{aligned} E &= -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C} | w_I) \\ &= -\log \prod_{c=1}^{C} \frac{\exp(u_{c,j_c^*})}{\sum_{j'=1}^{V} \exp(u_{j'})} \\ &= -\sum_{c=1}^{C} u_{j_c^*} + C \cdot \log \sum_{j'=1}^{V} \exp(u_{j'}) \end{aligned}$$

Source: https://arxiv.org/pdf/1411.2738v3.pdf
Where, j*c is the actual c-the output of the target word.

Back Propagation:
We need to update the **W** and **W`** through the backpropagation. The gradient descent technique is used to update the **W** and **W`**. For updating the W and W` we have to calculate the derivative of loss with respect to their weight and subtract this derived loss from the actual weight.

W` can be updated by the following function.

$$w'_{ij}{}^{(\text{new})} = w'_{ij}{}^{(\text{old})} - \eta \cdot e_j \cdot h_i.$$

Source: https://arxiv.org/pdf/1411.2738v3.pdf

Where, η(Eta) is the learning rate.
Similarly, we can update the **W** by the following equation.

$$w_{ij}{}^{(\text{new})} = w_{ij}{}^{(\text{old})} - \eta \cdot e_j \cdot h_i.$$

Word2Vec for CBOW Model:

After many iterations, **W** and **W`** will be tuned. Tuned **W** and **W`** is the vector of the vocabulary word. We can use both **W** and **W`** as the vector. Generally, **W** is used as the vector.

Continues Skip-Gram Model: Skip-gram is similar to the **CBOW** model but in a reverse way. In the skip-gram model, we try to predict the context words for a target word. So, our input will be the one-hot encoded target vector and output will be the encoded context vector. As we are taking the target word as input, we don’t need to take the average at the input layer.