# Machine Translate

## English to French

Goal is to convert sentences in English to French using Recurrent neural network Encoder-Decoder model.

-Team 1
Nimish Jindal (UF ID : 56553685)
Vivek Gade (UF ID : 98180101)

# Table of Contents

# Natural Language Processing

## 1. Introduction

Human language is specifically “constructed” to convey meaning, and is not produced by a physical manifestation(like image) of any kind. In that way, it is very different from vision or any other machine learning task. Most words are just symbols that convey some idea or thing. Processing natural language is challenging because natural language is ambiguous unlike programming languages where tokens have fixed meaning and syntax is also properly defined.

There are different levels of tasks in NLP, from speech processing to semantic interpretation and discourse processing. The goal of NLP is to be able to design algorithms to allow computers to "understand" natural language in order to perform some task. Tasks can be of varying level of difficulty:

Easy
• Spell Checking
• Keyword Search
• Finding Synonyms

Medium
• Parsing information from websites, documents, etc.

Hard
• Machine Translation (e.g. Translate French text to English)
• Semantic Analysis (What is the meaning of query statement?)
• Coreference (e.g. What does "he" or "it" refer to given a document?)
• Question Answering (e.g. Answering Jeopardy questions).

## 2. How to represent words?

The first and most important common denominator across all NLP tasks is how we represent words as input to any of our programming models. Much of the earlier NLP work treated each word as atomic symbol.

To perform well on most NLP tasks we first need to have some notion of similarity and difference between words. For example - in realm of real numbers 9 is more closer to 10 than 2. We know this and computer also “understands” this even though 1, 2, 3…. are also some kind of symbols in metric space. A lot mathematical models have been build upon this fact. But if I give you “hotel” and “motel” then you know that these two words/symbols are “close” in meaning

but computer does not understand the same. Moreover, we also know that same stem words can convey different meanings based on the context (like tense, count, gender).

So, first we know that we are looking for some kind of similarity between words. Second, we need more than one dimension to represent meaning of a word because different dimensions can encode different aspects of meaning(tense). With Vector spaces(Inner Product Spaces) we easily fulfill both of these requirements. Given two vectors you can scale them, add them (i.e., make any linear combination of them which would also be a vector) or find similarity between them(inner product) using distance measures such as Jaccard, Cosine, Euclidean, etc.

# Word Vectors

There are an estimated 13 million tokens for the English language all of which are not completely unrelated. For example, “Feline” or “cat” and “hotel” or “motel”. So there should exist some N-dimensional space that is sufficient to encode all semantics of our language. For instance, semantic dimensions might indicate -
>tense (past vs. present vs. future),
>count (singular vs. plural), and
>gender (masculine vs. feminine).

One simple word vector representation could be - the **one-hot vector**: represent every word as $R^{|V|}\times 1$ vector with all 0s and one 1 at the index of that word in the sorted english language. In this notation, |V| is the size of our vocabulary. Word vectors in this type of encoding would appear as the following:

$$w^{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

We represent each word as a completely independent entity. As we previously discussed, this word representation does not give us directly any notion of similarity. For instance,

$$(w^{hotel})^{T} w^{motel} = (w^{hotel})^{T} w^{cat} = 0$$

So maybe we can try to reduce the size of this space from $R^{|V|}$ to something smaller and thus find a subspace that encodes the relationships between words.

## 1. SVD Based Methods

For this class of methods to find word embeddings (another terminology word vectors), we first loop over a massive dataset and accumulate word co-occurrence counts in some form of a matrix X, and then perform Singular Value Decomposition on X to get a $USV^{T}$ decomposition. We then use the rows of U as the word embeddings for all words in our dictionary. For example, Let our corpus contain just three sentences and the window size be 1:
1. I enjoy flying.
2. I like NLP.
3. I like deep learning

The resulting counts matrix will then be:

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

We now perform SVD on X, observe the singular values (the diagonal entries in the resulting S matrix), and cut them off at some index k based on the desired percentage variance captured:

$$\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$$

We then take the submatrix of $U_{1:|v|,\ 1:k}$ to be our word embedding matrix. This would thus give us a k-dimensional representation of every word in the vocabulary. This word vector is more than sufficient to encode semantic and syntactic (part of speech) information but this method of computation is not efficient in space and time complexity and many other aspects.

SVD based methods do not scale well for big matrices and it is hard to incorporate new words or documents. Computational cost for a m × n matrix is $O(mn^2)$.

## 2. Iteration Based Methods - Word2vec

Instead of computing and storing global information about some huge dataset (which might be billions of sentences), we can try to create a model that will be able to learn word vectors one iteration at a time. **The idea is to design a model whose parameters that we learn are the word vectors.** Remember, we have already encountered this situation before. In neural networks, we learned weight vectors(normal to hyperplane) after seeing each data point. The objective then was to minimize loss (wrong classification of data point) in case of classification problem.

So our goal is essentially to define an objective and then allow backpropagation to penalize the model parameters that caused the error. This idea of learning the weights through backpropagation was first introduced way back in 1986. Given a problem you just need to come

up with a well defined objective. The simpler the model and the task(objective function), the faster it will be to train it.

Several approaches have been tested in this way that - we first convert words to vectors iteratively and then use them for special tasks like POS tagging etc. It was shown by [Collobert et al., 2011] that model parameters(word vectors) achieved great performance over various tasks. Here we discuss a simpler, more recent, probabilistic method by [Mikolov et al., 2013] - word2vec. Word2vec is a software package that actually includes :

- 2 algorithms: continuous bag-of-words (CBOW) and skip-gram. CBOW aims to predict a center word from the surrounding context in terms of word vectors. Skip-gram does the opposite, and predicts the distribution (probability) of context words from a center word.

- 2 training methods: negative sampling and hierarchical softmax. Negative sampling defines an objective by sampling negative examples, while hierarchical softmax defines an objective using an efficient tree structure to compute probabilities for all the vocabulary.

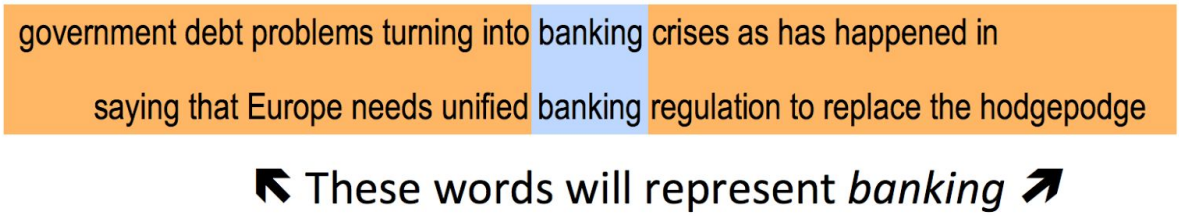
## 3. The Skip Gram Model

We want to train our model, parameterized by word vectors, one sentence at a time.

We know that dot product of vectors gives us some notion of similarity between them.

Now we need an idea that, given a sentence, could tell us how to improve the word vectors. Have you played ever played this game- “Fill in the Blanks”? Word2vec (Skipgram or CBOW) model is essentially based on this idea that - we can get a good sense about a word by looking at the neighboring words(context). Example:

- Find thousands of instances of the word banking in text/corpus.
- Check the neighbouring words - “debt problems”, “government”, “regulation”, “crises”...
- We know that given these words, the blank to be filled has high probability of being “banking” - This objective is refered as CBOW.
- Or given the word “banking”, there’s high probability that context words would be “debt problem”, “government”, “regulation” - This objective is refered as Skipgram.




More formally, the model is going to define a probability distribution - the probability of a word appearing in the context(neighborhood) of a given centre word. Learning goal is to find a distribution (**parameters of distribution - word vectors**) that is most is most likely to have generated this data set(context_word, center_word). We start from an initial random distribution, and then change the parameters(hence distribution) as we see more and more examples(context_word, center_word pairs). We change the parameters as to maximize the probability of seeing a context word given the center word.

**Objective function**:
For each word t = 1… T, predict surrounding words in a window of "radius" m of every word.

Maximize the probability of any context word($\mathbf{w}_{t+j}$) given the current center word($\mathbf{w}_t$).

$$J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} p(w_{t+j} | w_t ; \theta)$$

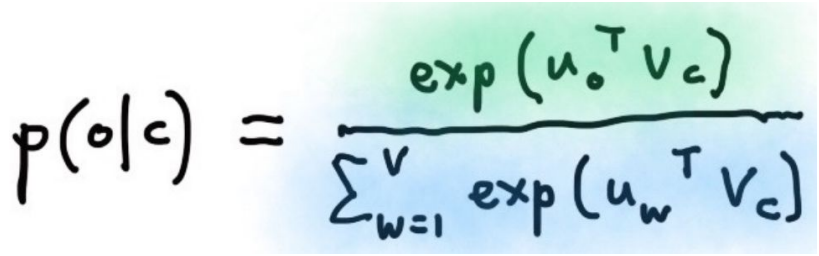
As we do this, word vectors for both "bank" and "banking" will modify overtime to maximize the probability of their context words. As both shall have same context words, "bank" and "banking" word vectors shall come closer.

Want to maximise the probability of these predictions.
-We take log so it's easier to work with derivatives(direct summation).
-put a negative: because ML people usually love to minimise things rather than maximise. And so we have **negative log likelihood** or negative log probability. So that's our objective function that we'll be formally minimizing. (Note: objective function/loss function/cost function it's all the same)

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log p(w_{t+j} | w_t)$$

Where θ represents model parameters that we'll optimize.

For $\mathbf{p(w_{t+j} \mid w_t)}$ the simplest formulation is:



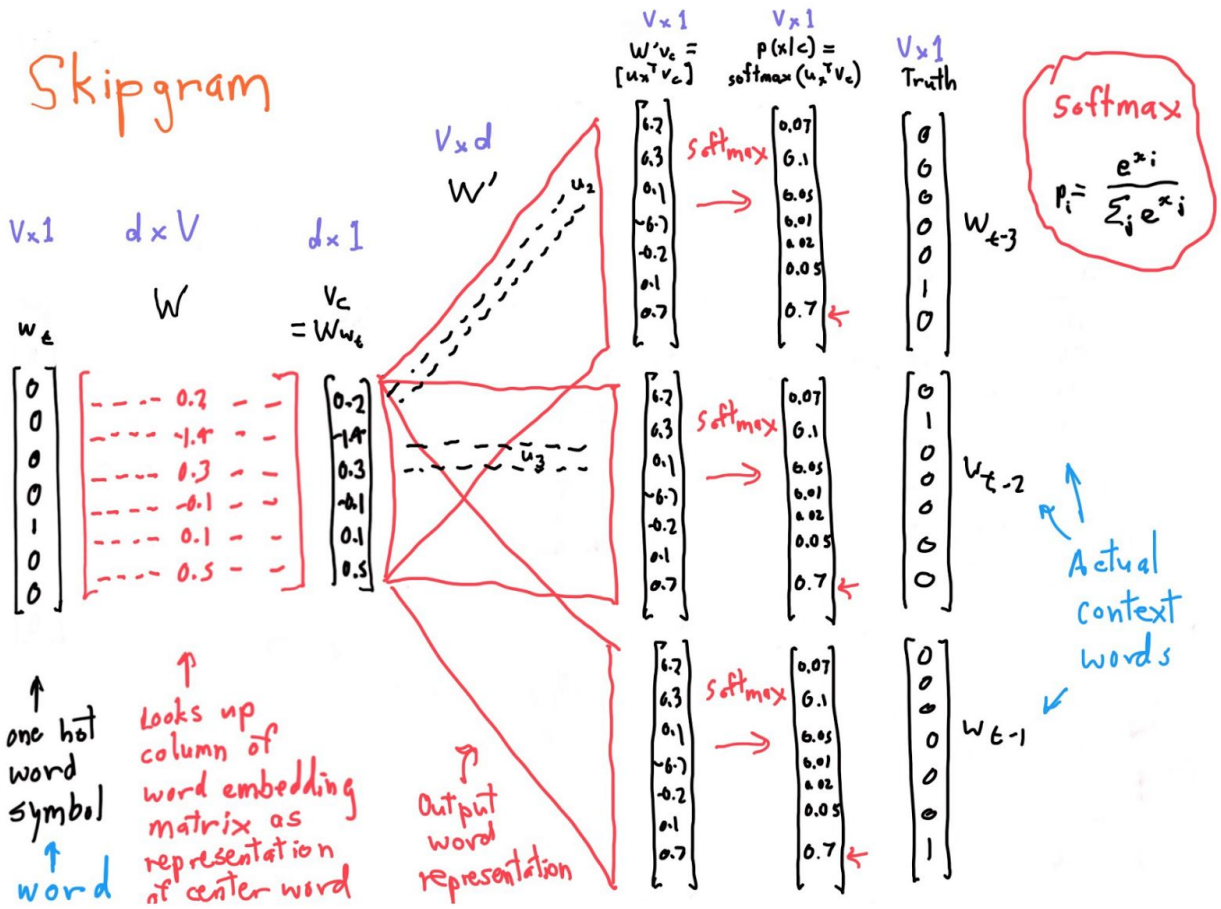
$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$$

where
-o is the output (or context) word index, c is the center word index.
-$v_c$ and $u_o$ are “center” and “outside” vectors

-**Why dot product?** We use dot product (of $v_c$ and $u_o$) in the SoftMax form because dot product gives sort of similarity b/w 2 vectors. So maximizing p(o|c) would imply maximizing the dot product. That is the 2 vectors will be placed close to each other in the vector space. So, as you can guess there are two vector representations of each word in the model- one as context vector and other as center vector. We return the center vector representation as word vectors after the model is trained.

-**Why softmax?** Softmax is a standard way to map from $R^V$ to a probability distribution. When we calculate dot products they are just real numbers 17,-23,4.. etc. so we can’t directly turn them into probability distribution. So an easy thing we can do is exponentiate them, so they are all positive and then normalize(divide by sum of each).
When we exponentiate, the big things get way bigger and so they dominate, and distribution blows out in the direction of a max element. So you might think that that's a bad thing to do. Doing things like this is "standard" approach, underlying a lot of math, including all those super common logistic regression functions. Researchers have certainly worked on whole bunch of other ways to fix this as well. However, this simple standard approach gives very good results. Probably that’s why we keep using it.

-**Are we paying attention to the location of context words w.r.t center word?** No. In this simple model we just care about which are the context words($u_{o1}$, $u_{o2}$ ,$u_{o2}$) in the window size m of center word($v_c$), and we train one example ($u_{o1,}$ $v_c$) at a time. It's not that this is a bad idea, there are other models which absolutely pay attention to position and distance. But if you're just interested in word meaning(which our word vector is supposed to represent) then it turns out that not paying attention to position actually helps you rather than hurting you.




**Intuition:**

And what does it mean for two words to have similar contexts(neighbours)? You can expect that synonyms like “intelligent” and “smart” would have very similar contexts. Or that words that are related, like “engine” and “transmission”, would probably have similar contexts as well. This can also handle stemming for us – the network will likely learn similar word vectors for the words “ant” and “ants” because these should have similar contexts.

## Limitations

Skip-gram neural network contains a huge number of weights. For vocab of 10000 words and 300 features, network would have 3M weights in the hidden layer and output layer each! And to make matters worse, you need a huge amount of training data in order to tune that many

weights and avoid over-fitting. Millions of weights times billions of training samples means that training this model is going to be a beast.

## Improvements

The authors of Word2Vec addressed these issues in their second paper.
There are three innovations in this second paper:

1. Treating common word pairs or phrases as single “words” in their model.
2. Subsampling frequent words to decrease the number of training examples.
3. Modifying the optimization objective with a technique they called “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.

# Neural Machine Translation

## 1. Introduction

Machine translation (MT) is the translation of text by a computer, with no(minimum) human involvement. In other words,

Given: source language text X = (word1 word2 ...)

Find: target language text Y = (word1' word2' ...)

Neural machine translation (SMT) approach is based on the following probabilistic model -

- Every sentence in one language is a possible translation of any sentence in the other language -

$$p(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T),$$

- The objective of the model is then to maximize this probability over all n training examples. Again, we take log to replace product( $\Pi$ ) with summation( $\Sigma$ ).

$$\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{y}_n \mid \mathbf{x}_n),$$

where θ is the set of the model parameters and each ($x_n$,$y_n$) is an input/output sentence pair from the training set.

The problem of translation could be approached in two ways -

- Directly convert source text(X) into target text(Y) or
- First convert/encode source text into some intermediate form(C) and then decode the target text from encoding.

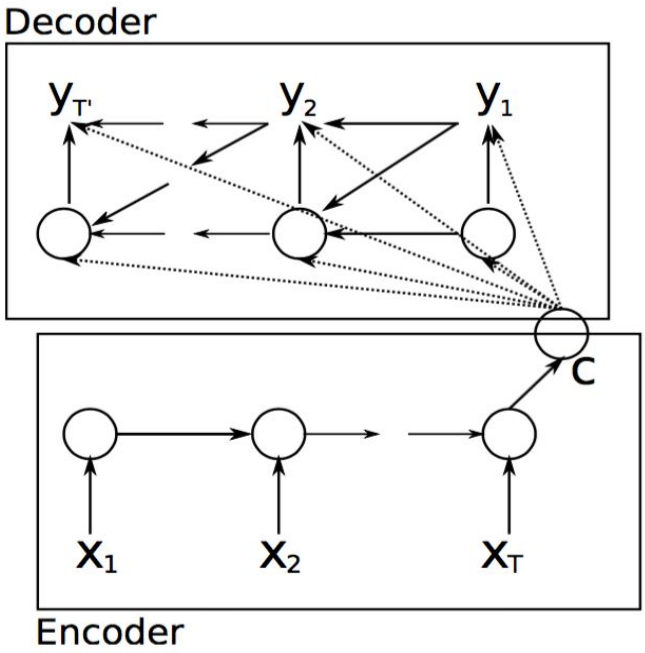
It has been observed that second approach gives better translation results.

We implemented a novel neural network architecture that

- First encodes a variable-length sequence into a fixed-length vector representation,
- And then decodes a given fixed-length vector representation back into a variable-length sequence.

The complete model then learns the parameters using backpropagation with objective function defined above, one sentence at a time.

## 2. Encoder-Decoder Architecture




**Encoder:**
An encoder reads the input sentence(X), a sequence of word vectors ($x_1, \cdots, x_T$), and converts it into a vector C. Idea is, hidden layers in RNN can be thought of as summarizing the input seen so far.

$$(\text{input}(x_1) + \text{empty\_hidden}) \text{ -> hidden -> output}$$
$$(\text{input}(x_2) + \text{prev\_hidden}) \text{ -> hidden -> output}$$
$$(\text{input}(x_3) + \text{prev\_hidden}) \text{ -> hidden -> output}$$
$$(\text{input}(x_4) + \text{prev\_hidden}) \text{ -> hidden -> output}$$

So we represent our encoding(context vector C) as output of hidden layer when final input $x_T$ is passed. We represent this RNN as -

$$h_t = f(x_t, h_{t-1})$$

where $h_t$ is a hidden state at time t, $x_t$ is input word vector at time t and f is non-linear multi-layered function. C = $h_T$ is the vector generated from the sequence of the hidden states.

**Decoder:**
The decoder defines a probability over the translation y by decomposing the joint probability into the ordered conditionals.

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c).$$

With an RNN, each conditional probability is modeled as - a unidirectional language model, which computes the conditional distribution over the next target word given all the previous target words and the encoding(context vector) -

$$p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c),$$

where g is a nonlinear, multi-layered function that outputs the probability of $y_t$, and $s_t$ is the hidden state of the RNN.

**Backpropagation:**
The two components of the proposed RNN Encoder–Decoder are jointly trained to maximize the conditional log-likelihood

$$\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{y}_n \mid \mathbf{x}_n).$$

This is done by stochastic gradient descent, where the gradient of the log-likelihood is efficiently computed by the backpropagation algorithm.

**In practice:**
For large datasets, each word of the source/target phrase can be embedded in a 500 dimensional vector space. The hidden state of an encoder/decoder can consist of 1000 hidden units.

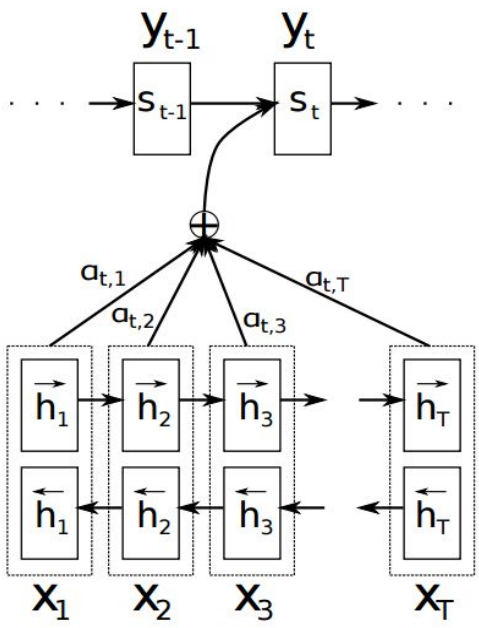
# 3. Limitations

1. We used sigmoid as activation function that is vulnerable to Vanishing gradient problem.
2. RNNs can theoretically store information over long sequence of words but practically they do not.
3. An encoding produced by the above model is often disproportionately influenced by words appearing later in the sentence.

# 4. Improvements

1. Bidirectional encoder: This will allow us to encode “phrase”-level or structural information well in source sentence.

2. Attention based mechanism: This is now a standard De-facto technique. Find annotation vector($h_t$) corresponding to each word in source sentence, in forward and backward direction and concatenate them.




3. The context vector $\mathbf{c}_i$ depends on a sequence of annotations ($h_1$,$h_2$ ...)

4. Each annotation $h_i$ contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence.

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

5. Let $\boldsymbol{\alpha}_{ij}$ be a probability that the target word $\mathbf{y}_i$ is aligned to, or translated from, a source word $\mathbf{x}_j$. The weight $\boldsymbol{\alpha}_{ij}$ of each annotation $h_j$ is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

where,

$$e_{ij} = a(s_{i-1}, h_j)$$

is an alignment model which scores how well the inputs around position j and the output at position i match.The alignment model is parametrized as a feedforward neural network which is jointly trained with all the other components of the proposed system.

# References

**Word2Vec:**
Following papers:
1. A neural probabilistic language model.
http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf
2.Natural language processing (almost) from scratch. [Collobert et al., 2011]
https://arxiv.org/pdf/1103.0398.pdf
3. Word2Vec: Efficient Estimation of Word Representations in Vector Space. [Mikolov et al., 2013]
http://arxiv.org/pdf/1301.3781.pdf
4. Word2Vec: Distributed Representations of Words and Phrases and their Compositionality
http://arxiv.org/pdf/1310.4546.pdf

Online Tutorials:
Stanford lecture: CS224n: Natural Language Processing with Deep Learning
http://web.stanford.edu/class/cs224n/index.html
http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

**Neural Machine Translation:**
Following papers:
1. Learning phrase representations using RNN encoder-decoder for statistical machine translation. https://arxiv.org/abs/1406.1078
2. On the properties of neural machine translation: Encoder-decoder approaches.
https://arxiv.org/abs/1409.1259
3. Neural machine translation by jointly learning to align and translate.
https://arxiv.org/abs/1409.0473
4. Learning Representations by Back-propagating Errors
https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Online Tutorials
https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/
http://karpathy.github.io/2015/05/21/rnn-effectiveness/
http://cs231n.github.io/
http://colah.github.io/posts/2015-08-Understanding-LSTMs/