**Abstract**

Deep learning techniques utilize different processing layers to comprehend hierarchical representations of data. In natural language processing (NLP), a variety of model designs and methods have bloomed in recent times. In this paper, we evaluate important deep learning-related models and methods that have been assigned for several NLP assignments and deliver the path of their progression. Comprehensive knowledge of the past, present, and future of deep learning in NLP is proposed. We furthermore correlate, contrast, and shorten the several models.

**Index**

**Natural language processing (NLP)**



Natural language processing (NLP) utilizes computer algorithms and artificial intelligence to enable computers to recognize and respond to human communication. It is an attempt to make computers intelligent by making humans believe they are interacting with another human. While several NLP methods exist, they typically involve breaking speech or text into discrete sub-units and then comparing these to a database of how these units fit together based on experience. It was formulated to build software that generates and comprehends natural languages so that a user can have natural conversations with a computer instead of through programming or artificial languages like Java or C.

Examples of NLP: Amazon Echo, Text-to-speech apps, which are now found on most iOS and Android platforms

**Stages of Natural Language Processing (NLP)**

**Stage 1:**

The first task of NLP is to understand the natural language received by the computer. A built-in statistical model is used to perform a speech recognition routine that converts the natural language to a programming language.

**Stage 2:**

The next task is called the part-of-speech (POS). The grammatical form of a word is identified in this process i.e., nouns, verbs, adjectives, tenses, etc. Now the computer can understand the meaning of the speech that was made.

**Stage 3:**

The final task is text-to-speech conversion. At this stage, the computer programming language is converted to the audible or textual format.

**Word2Vec**

Word embeddings were revolutionized by Mikolov and his co-workers, it is one of the most popular techniques for Vector representation of a particular word.

Consider the following similar sentences: **Have a good day** and **Have a great day**. They hardly have different meanings. If we construct an exhaustive vocabulary (let’s call it V), it will have V = {Have, a, good, great, day}.

Assume a one-hot encoded vector

Have = [1,0,0,0,0] ; a=[0,1,0,0,0] ; good=[0,0,1,0,0] ; great=[0,0,0,1,0] ; day=[0,0,0,0,1] ;

[1,0,0,0,0]

[0,1,0,0,0]

V = [0,0,1,0,0]

[0,0,0,1,0]

[0,0,0,0,1]

If we try to visualize these encodings, we can think of a 5-dimensional space, where each word occupies only one of the dimensions. This means ‘good’ and ‘great’ are as different as ‘day’ and ‘have’, which is not true.

Our objective is to have words with similar contexts occupy close spatial positions. To overcome this, we use the CBOW (Common Bag Of Words) model

**CBOW Model**

CBOW model enumerates the conditional probability of a target word given in the context word neighboring it the skip-gram model does the contrasts of the CBOW model, by predicting the surrounding context words

given the central target word. The context words are supposed to be located symmetrically to the target words within a distance one at the same to the window size in both directions. In unsupervised settings, the word embedding dimension is resolved by the accuracy of prediction. As the embedding dimension increases, the accuracy of prediction also increases. The CBOW model is a simple fully connected neural network with one hidden layer.

The input layer- takes the one-hot vector of context word has V neurons

the hidden layer- has N neurons.

The output layer- is the softmax probability of overall words in the vocabulary.

Finally, the three layers are connected by a weight matrix

**W ∈ RV ×N and W**

**0**

**∈ RH×V**

**Skip-Gram** **model**

In a collection of sentences the model loops and tries to use the current word to predict its context is called the “Skip-Gram” model. **“Window size**” is a parameter attuned to set the word limit in each context. This model in its basic form trains a simple neural network with a single hidden layer to perform a certain task, the goal is to learn the weights of the hidden layer. “Word vectors” is a term used to denote these weights.

Training a neural network to do the following:

Given a specific word as the input from a sentence, look at the words nearby and pick one at random. The network is going to specify the probability for every word in the vocabulary of being the “nearby word” that has been chosen. For example, in the trained network the input word is “America”, and the output probabilities are going to be much higher for words like “the states” and “U.S.A” than for unrelated words like “sugar” and “salt”.

We’ll train the neural network to do this by feeding it word pairs found in our training documents. The below example shows some of the training samples (word pairs) we would take from the sentence “The quick brown fox jumps over the lazy dog.” I’ve used a small window size of 2 just for the example. The word highlighted in blue is the input word.

Graphical user interface, application, Teams

Description automatically generated

We’re going to represent an input word like “ants” as a one-hot vector. This vector will have 10,000 components (one for every word in our vocabulary) and we’ll place a “1” in the position corresponding to the word “ants”, and 0s in all of the other positions.

The output of the network is a single vector (also with 10,000 components) containing, for every word in our vocabulary, the probability that a randomly selected nearby word is that vocabulary word.

Here’s the architecture of our neural network.

Schematic

Description automatically generated with medium confidence

There is no activation function on the hidden layer neurons, but the output neurons use softmax.

For our example, we’re going to say that we’re learning word vectors with 300 features. So the hidden layer is going to be represented by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for every hidden neuron).

300 features is what Google used in their published model trained on the Google news dataset (you can download it from [here](https://code.google.com/archive/p/word2vec/)). The number of features is a “hyper parameter” that you would just have to tune to your application (that is, try different values and see what yields the best results).

If you look at the *rows* of this weight matrix, these are actually what will be our word vectors!

A picture containing chart

Description automatically generated

So the end goal of all of this is really just to learn this hidden layer weight matrix — the output layer we’ll just toss when we’re done! The 1 x 300 word vector for “ants” then gets fed to the output layer. The output layer is a softmax regression classifier.

Specifically, each output neuron has a weight vector which it multiplies against the word vector from the hidden layer, then it applies the function exp(x) to the result. Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from *all* 10,000 output nodes.

Here’s an illustration of calculating the output of the output neuron for the word “car”.

Graphical user interface, application, Teams

Description automatically generated

If two different words have very similar “contexts” (that is, what words are likely to appear around them), then our model needs to output very similar results for these two words. And one way for the network to output similar context predictions for these two words is if *the word vectors are similar*. So, if two words have similar contexts, then our network is motivated to learn similar word vectors for these two words! Ta da!

And what does it mean for two words to have similar contexts? I think you could 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 you — the network will likely learn similar word vectors for the words “ant” and “ants” because these should have similar contexts.

We need few additional modifications to the basic skip-gram model which are important for actually making it feasible to train. Running gradient descent on a neural network that large is going to be slow. 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. The authors of Word2Vec addressed these issues in their second [paper](http://arxiv.org/pdf/1310.4546.pdf).

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.

It’s worth noting that subsampling frequent words and applying Negative Sampling not only reduced the compute burden of the training process, but also improved the quality of their resulting word vectors as well.

**Subsampling:**

There are two “problems” with common words like “the”:

1. When looking at word pairs, (“fox”, “the”) doesn’t tell us much about the meaning of “fox”. “the” appears in the context of pretty much every word.
2. We will have many more samples of (“the”, …) than we need to learn a good vector for “the”.

Word2Vec implements a “subsampling” scheme to address this. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is related to the word’s frequency.

If we have a window size of 10, and we remove a specific instance of “the” from our text:

1. As we train on the remaining words, “the” will not appear in any of their context windows.
2. We’ll have 10 fewer training samples where “the” is the input word.

Note how these two effects help address the two problems stated above.

**Negative Sampling:**

As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here’s how it works.

When training the network on the word pair (“fox”, “quick”), recall that the “label” or “correct output” of the network is a one-hot vector. That is, for the output neuron corresponding to “quick” to output a 1, and for *all* of the other thousands of output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of “negative” words (let’s say 5) to update the weights for. (In this context, a “negative” word is one for which we want the network to output a 0 for). We will also still update the weights for our “positive” word (which is the word “quick” in our current example).

The paper says that selecting 5–20 words works well for smaller datasets, and you can get away with only 2–5 words for large datasets.

Recall that the output layer of our model has a weight matrix that’s 300 x 10,000. So we will just be updating the weights for our positive word (“quick”), plus the weights for 5 other words that we want to output 0. That’s a total of 6 output neurons, and 1,800 weight values total. That’s only 0.06% of the 3M weights in the output layer!

In the hidden layer, only the weights for the input word are updated (this is true whether you’re using Negative Sampling or not).

The “negative samples” (that is, the 5 output words that we’ll train to output 0) are chosen using a “unigram distribution”.

Essentially, the probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.

**Intuitive Understanding of Attention Mechanism in Deep Learning**

A TensorFlow Implementation of Neural Machine Translation with Attention

**Caution**

This is a slightly advanced tutorial and requires basic understanding of sequence to sequence models using RNNs. Please refer my earlier [*blog here*](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7) wherein I have explained in detail the concept of Seq2Seq models.

**Table of Contents**

1. Introduction
2. The central idea behind Attention
3. Why the name Attention?
4. How does Attention work?
5. Code Walk through
6. Visualizing the Results
7. References

**1. Introduction**

Attention is one of the most influential ideas in the Deep Learning community. Even though this mechanism is now used in various problems like image captioning and others,it was initially designed in the context of Neural Machine Translation using Seq2Seq Models. In this blog post I will consider the same problem as the running example to illustrate the concept. We would be using attention to design a system which translates a given English sentence to Marathi, the exact same example I considered in my earlier [blog](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7).

So what’s wrong with seq2seq models?

The seq2seq models is normally composed of an encoder-decoder architecture, where the encoder processes the input sequence and encodes/compresses/summarizes the information into a context vector (also called as the “thought vector”) of a fixed length. This representation is expected to be a good summary of the entire input sequence. The decoder is then initialized with this context vector, using which it starts generating the transformed output.

***A critical and apparent disadvantage of this fixed-length context vector design is the incapability of the system to remember longer sequences. Often is has forgotten the earlier parts of the sequence once it has processed the entire the sequence. The attention mechanism was born to resolve this problem.***

Let’s break this down into finer details. Since I have already explained most of the basic concepts required to understand Attention in my previous [*blog*](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7), here I will directly jump into the meat of the issue without any further adieu.

**2. The central idea behind Attention**

For the illustrative purposes, I will borrow the same example that I used to explain Seq2Seq models in my previous [*blog*](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7).

Input (English) Sentence: “Rahul is a good boy”

Target (Marathi) Sentence: “राहुल चांगला मुलगा आहे”

The only change will be that instead of an LSTM layer that I used in my previous explanation, here I will use a GRU layer. The reason being that LSTM has two internal states (hidden state and cell state) and GRU has only one internal state (hidden state). This will help simplify the the concept and explanation.

Recall the below diagram in which I summarized the entire process procedure of Seq2Seq modelling.

Diagram

Description automatically generated

In the traditional Seq2Seq model, we discard all the intermediate states of the encoder and use only its final states (vector) to initialize the decoder. This technique works good for smaller sequences, however as the length of the sequence increases, a single vector becomes a bottleneck and it gets very difficult to summarize long sequences into a single vector. This observation was made empirically as it was noted that the performance of the system decreases drastically as the size of the sequence increases.

The central idea behind Attention is not to throw away those intermediate encoder states but to utilize all the states in order to construct the context vectors required by the decoder to generate the output sequence.

**3. Why the name Attention?**

Let’s name each of the intermediate states of the encoder as below:

Engineering drawing

Description automatically generated

Encoder GRU

Notice that since we are using a GRU instead of an LSTM, we only have a single state at each time step and not two states, which thus helps to simplify the illustration. Also note that attention is useful specially in case of longer sequences but for the sake of simplicity we will consider the same above example for illustration.

Recall that these states (h1 to h5) are nothing but vectors of fixed length. To develop some intuition think of these states as vectors which store local information within the sequence. For example;

h1 stores the information present in the start of the sequence (words like ‘Rahul’ and ‘is’) while h5 stores the information present in the later part of the sequence (words like ‘good’ and ‘boy’).

Lets represent our Encoder GRU with the below simplified diagram:

Diagram

Description automatically generated

Compact Representation of Encoder GRU

Now the idea is to utilize all of these local information collectively in order to decide the next sequence while decoding the target sentence.

Imagine you are translating “Rahul is a good boy” to “राहुल चांगला मुलगा आहे”. Ask yourself, how do you do it in your mind?

When you predict “राहुल”, its obvious that this name is the result of the word “Rahul” present in the input English sentence regardless of the rest of the sentence. We say that while predicting “राहुल”, ***we pay more attention*** to the word “Rahul” in the input sentence.

Similarly while predicting the word “चांगला”, we pay more attention to the word “good” in the input sentence.

Similarly while predicting the word “मुलगा”, we pay more attention to the word “boy” in the input sentence. And so on..

Hence the name ***“ATTENTION”.***

As human beings we are quickly able to understand these mappings between different parts of the input sequence and corresponding parts of the output sequence. However its not that straight forward for artificial neural network to automatically detect these mappings.

Thus the Attention mechanism is developed to ***“learn”*** these mappings through Gradient Descent and Back-propagation.

**4. How does Attention work?**

Let’s get technical and dive into the nitty gritty of Attention mechanism.

**Decoding at time step 1**

Continuing the above example, let’s say we now want our decoder to start predicting the first word of the target sequence i.e. “राहुल”

At time step 1, we can break the entire process into **five steps** as below:

Diagram

Description automatically generated

Decoding at time step 1

Before we start decoding, we first need to encode the input sequence into a set of internal states (in our case h1, h2, h3, h4 and h5).

Now the hypothesis is that the next word in the output sequence is dependent on the current state of the decoder (decoder is also a GRU) as well as on the hidden states of the encoder. Thus, at each time step, we consider these two things and follow the below steps:

**Step 1 — Compute a score each encoder state**

Since we are predicting the first word itself, the decoder does not have any current internal state. For this reason, we will consider the last state of the encoder (i.e. h5) as the previous decoder state.

Now using these two components (all the encoder states and the current state of the decoder), we will train a simple feed forward neural network.

Why?

Recall we are trying to predict the first word in the target sequence i.e. “राहुल”. As per the idea behind attention, we do not need all the encoder states to predict this word, but we need those encoder states which store information about the word “Rahul” in the input sequence.

As discussed previously these intermediate encoder states store the local information of the input sequence. So it is highly likely that the information of the word “Rahul” will be present in the states, let’s say, h1 and h2.

Thus we want our decoder to pay more attention to the states h1 and h2 while paying less attention to the remaining states of the encoder.

For this reason, we train a feed forward neural network which will **learn** to identify relevant encoder states by generating a high score for the states for which attention is to be paid while low score for the states which are to be ignored.

Let s1, s2, s3, s4 and s5 be the scores generated for the states h1, h2, h3, h4 and h5 correspondingly. Since we assumed that we need to pay more attention to the states h1 and h2 and ignore h3, h4 and h5 to predict “राहुल”, we expect the above neural to generate scores such that s1 and s2 are high while s3, s4 and s5 are relatively low.

**Step 2— Compute the attention weights**

Once these scores are generated, we apply a SoftMax on these scores to produce the attention weights e1, e2, e3, e4 and e5 as shown above. The advantage of applying SoftMax is as below:

a) All the weights lie between 0 and 1, i.e., 0 ≤ e1, e2, e3, e4, e5 ≤ 1

b) All the weights sum to 1, i.e., e1+e2+3+e4+e5 = 1

Thus, we get a nice probabilistic interpretation of the attention weights.

In our case we would expect values like below: (just for intuition)

e1 = 0.75, e2 = 0.2, e3 = 0.02, e4 = 0.02, e5 = 0.01

This means that while predicting the word “राहुल”, the decoder needs to put more attention on the states h1 and h2 (since values of e1 and e2 are high) while ignoring the states h3, h4 and h5 (since the values of e3, e4 and e5 are very small).

**Step 3— Compute the context vector**

Once we have computed the attention weights, we need to compute the context vector (thought vector) which will be used by the decoder in order to predict the next word in the sequence. Calculated as follows:

context\_vector = e1 \* h1 + e2 \* h2 + e3 \* h3 + e4 \* h4 + e5 \* h5

Clearly if the values of e1 and e2 are high and those of e3, e4 and e5 are low then the context vector will contain more information from the states h1 and h2 and relatively less information from the states h3, h4 and h5.

**Step 4— Concatenate context vector with output of previous time step**

Finally the decoder uses the below two input vectors to generate the next word in the sequence

a) The context vector

b) The output word generated from the previous time step.

We simply concatenate these two vectors and feed the merged vector to the decoder. **Note that for the first time step, since there is no output from the previous time step, we use a special <START> token for this purpose**. This concept is already discussed in detail in my previous [*blog*](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7).

**Step 5— Decoder Output**

The decoder then generates the next word in the sequence (in this case, it is expected to generate “राहुल”) and along with the output, the decoder will also generate an internal hidden state, and lets call it as “d1”.

**Decoding at time step 2**

Now in order to generate the next word “चांगला”, the decoder will repeat the same procedure which can be summarized in the below diagram:

The changes are highlighted in **green circles**

Diagram

Description automatically generated

Decoding at time step 2

**Decoding at time step 3**

Diagram

Description automatically generated

Decoding at time step 3

**Decoding at time step 4**

Diagram

Description automatically generated

Decoding at time step 4

**Decoding at time step 5**

Diagram

Description automatically generated

Decoding at time step 5

Once the decoder outputs the <END> token, we stop the generation process.

Note that unlike the fixed context vector used for all the decoder time steps in case of the traditional Seq2Seq models, here in case of Attention, we compute a separate context vector for each time step by computing the attention weights every time.

Thus using this mechanism our model is able to find interesting mappings between different parts of the input sequence and corresponding parts of the output sequence.

Note that during the training of the network, we use teacher forcing in order to input the actual word rather than the predicted word from the previous time step. This concept also has been explained in my previous [*blog*](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7).

**5. Code Walk through**

As in case of any NLP task, after reading the input file, we perform the basic cleaning and preprocessing as follows:

Create a class to map every word to an index and vice-versa for any given vocabulary:

We use the tf.data input pipeline to create the dataset and then load it later in mini batches. To read more about the input pipeline in TensorFlow, go through the official documentations [*here*](https://www.tensorflow.org/guide/datasets)and [*here*](https://www.tensorflow.org/api_docs/python/tf/data/Dataset).

Now using the model sub-classing API of TensorFlow, we define the model as follows. To read more about model sub classing, read the official documentation [*here*](https://www.tensorflow.org/guide/keras#model_subclassing).

**Note**: Please read the comments in the below section of the code to get better understanding using the concepts we discussed above. Most of the important lines of the code point to the corresponding section of the explanation given above.

Define Optimizer, Loss Function and Checkpoints

Using Eager Execution, we train the network for 10 epochs. To read more about Eager Execution, refer the official documentation [*here*](https://www.tensorflow.org/guide/eager).

Inference setup and testing:

**6. Visualizing the Results**

Chart, timeline, treemap chart

Description automatically generated

If you are new to heat maps, this is how you can interpret the above plot:

Notice that the cell at the intersection of “father” and “बाबांनी” is pretty dark This means when the decoder predicts the word “बाबांनी”, it is paying more attention to the input word “father” (which is what we wanted).

Similarly while predicting the word “कॅमेरा”, the decoder pays a lot of attention to the input word “camera”. And so on.

Chart, timeline

Description automatically generated

Chart, timeline

Description automatically generated

Chart, timeline

Description automatically generated

Chart, timeline

Description automatically generated

Chart, timeline

Description automatically generated

**Conclusion**

The first thing to be noted is that the translation results are much better than the results from my previous [*blog*](https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7). Secondly the model is able to find the correct local mappings between the input and the output sequences which do match with our intuition.

Given more data and with more hyper parameter tuning, the results and mappings will definitely improve by a good margin.

Using LSTM layers in place of GRU and adding Bidirectional wrapper on the encoder will also help in improved performance.

Deep Learning models are generally considered as black boxes, meaning that they do not have the ability to explain their outputs. However, Attention is one of the successful methods that helps to make our model interpretable and explain why it does what it does.

The only disadvantage of the Attention mechanism is that it is a very time consuming and hard to parallelize system. To solve this problem, Google Brain came up with the “Transformer Model” which uses only Attention and gets rid of all the Convolutional and Recurrent Layers, thus making it highly parallelizable and compute efficient.

machines automatically learn how to detect sentiment without human input.