Contents

List of Figures

List of Tables

Chapter 1

Literature Review

1.1 A Neural Probabilistic Language Model

1.1.1 Introduction

Goal of statistical language modeling is to learn the joint probability function of sequences of words in a language. This is intrinsically difficult because of the curse of dimensionality. NPLM proposes to fight the curse of dimensionality by learning a distributed representation of words which allows each training sentence to inform the model about an exponential number of semantically neighboring sentences. The model learns simultaneously a distributed representation for each word along with the probability function for word sequences, expressed in terms of these representations. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar to the words forming an already seen sentence. The idea of the proposed model can be summarized as follows

- associate with each word in the vocabulary a distributed word feature vector (a real-valued vector in $\mathbb{R}^m$, where m is the size of embedding vector.)
- express the joint probability function of word sequences in terms of the feature vectors of these words in the sequence
- learn simultaneously the word feature vectors and the parameters of that probability function

1.1.2 NPLM Architecture

The basic form of the model is shown in the figure 1.1. The objective is to learn the function f(w_t, w_{t-1} ...., w_{t-n}) = P(w_t | w_1^{t-1}), in the sense that it gives high out-of-sample likelihood. A mapping C from any element of V to a real vector C(i) ϵ $\mathbb{R}^m$. It represents the distributed feature vector associated with each word in the vocabulary. In practice, C is represented by a |V| x m matrix (of free parameters)

From the direct architecture figure 1.1, f(i, w_{t-1}, w_{t-2}...., $wt-n$) = g(i, $C(w_{t-1})$, $C(w_{t-2})$ ..., C(w_{t-n})). Softmax function is used in the output layer of the neural network to get the probability of the target word.

The function f is the composition of two mappings (C and g), with C being shared across all words in the context. THe function g may be implemented by a feed-forward or recurrent neural network or another parameterized function, with parameters θ.

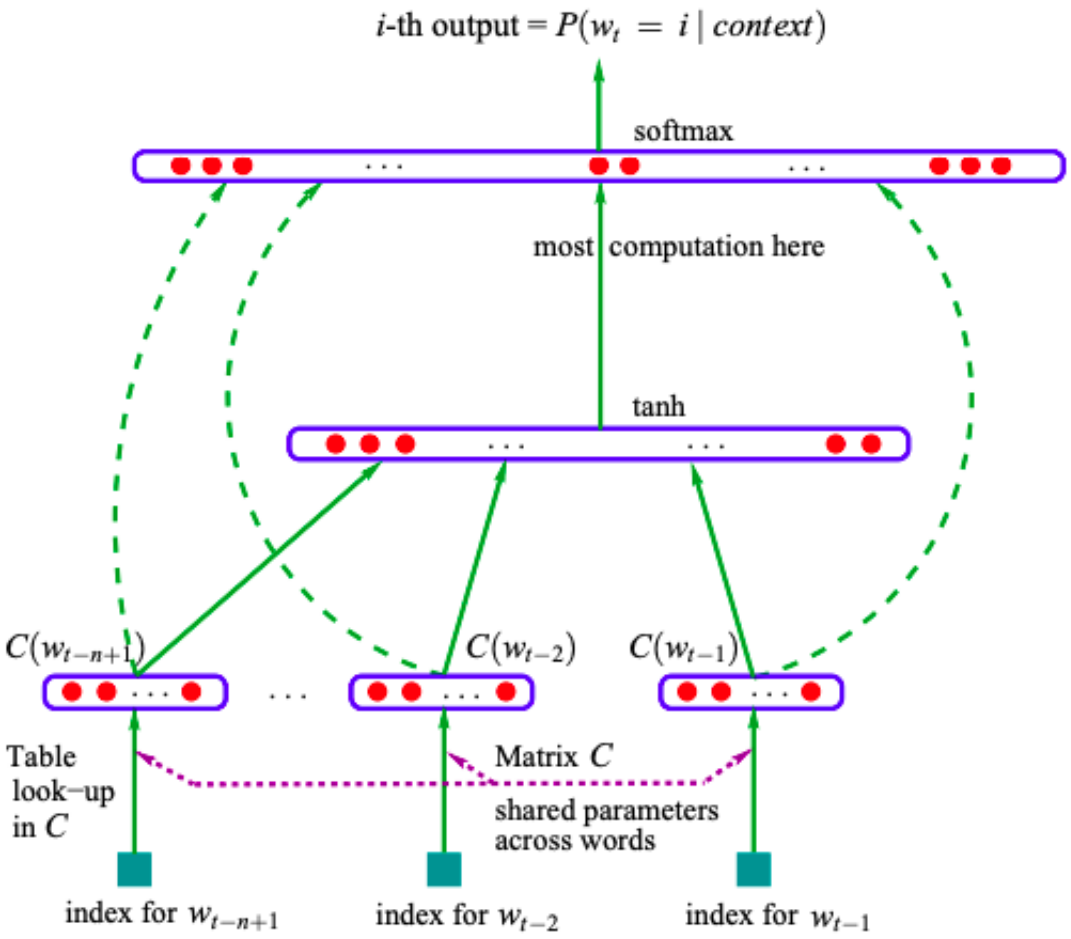

Figure 1.1: Direct Architecture of Probabilistic Language Model

1.1.3 Conclusion and Results

The main result of this experiment is that the neural network performs much better than the smoothed trigram. Greater the length of the context words and higher the number of hidden units, the model becomes more efficient. Moreover, direct architecture was found to be better by about 2% than the cycling architecture.

It can be deduced that the neural probabilistic model performs better due to the advantage of the learned distributed representation to fight the curse of dimensionality.

1.2 Attention is all you need

1.2.1 Introduction

Before the introduction of the transformer, the dominant sequence transduction models were based on the complex recurrent or convolutional neural networks that include an encoder and a decoder. But this paper introduced a new network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train.

Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing the modeling of dependencies without regard to their distance in the input or output sequences. In all but a few cases, however, such attention mechanisms are used in conjunction with a recurrent network.

1.2.2 Transformer Model Architecture

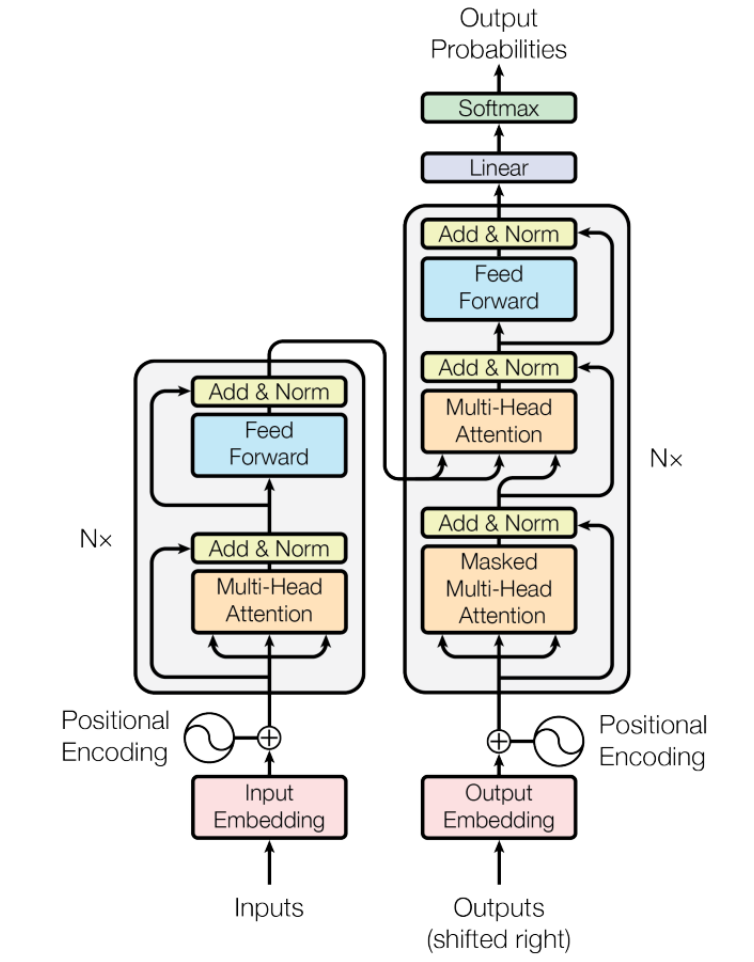

Figure 1.2: The Transformer - model architecture

Encoder and Decoder Stacks

The encoder is composed of a stack of N = 6 identical layers. Each layer has two sublayers: multi-head self-attention mechanism, and the simple position wise fully connected feed-forward network. Residual connection around each of two sub layers, followed by a layer normalization is employed in the encoder.

The decoder is also composed of a stack of N = 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, residual connections around each of the sub layers, followed by layer normalization is employed. Self attention sub-layer in the decoder stack is modified to prevent the positions from attending to subsequent positions.

Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

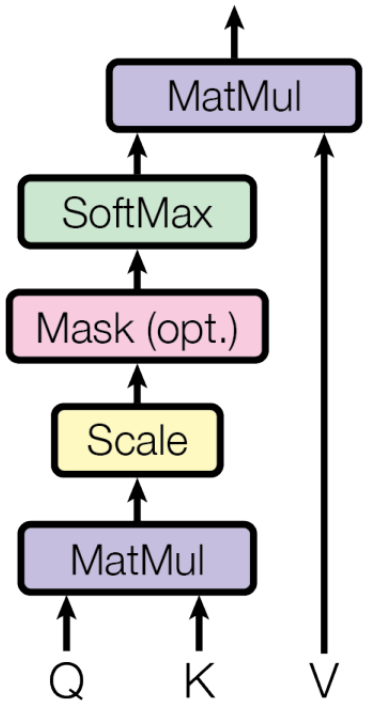

Figure 1.3: Scaled Dot-Product Attention

In scaled dot-product attention, the input consists of queries and keys of dimension d_k and values of dimension d_v. We compute the dot products of the query with all keys, divide each by d_k and apply a softmax function to obtain the weights on the values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1.1}$$

In multi-head attention, instead of performing a single attention function with d_{model} -dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in figure 1.4.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_2)W^o \tag{1.2}$$
$$\text{where } head_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V)$$

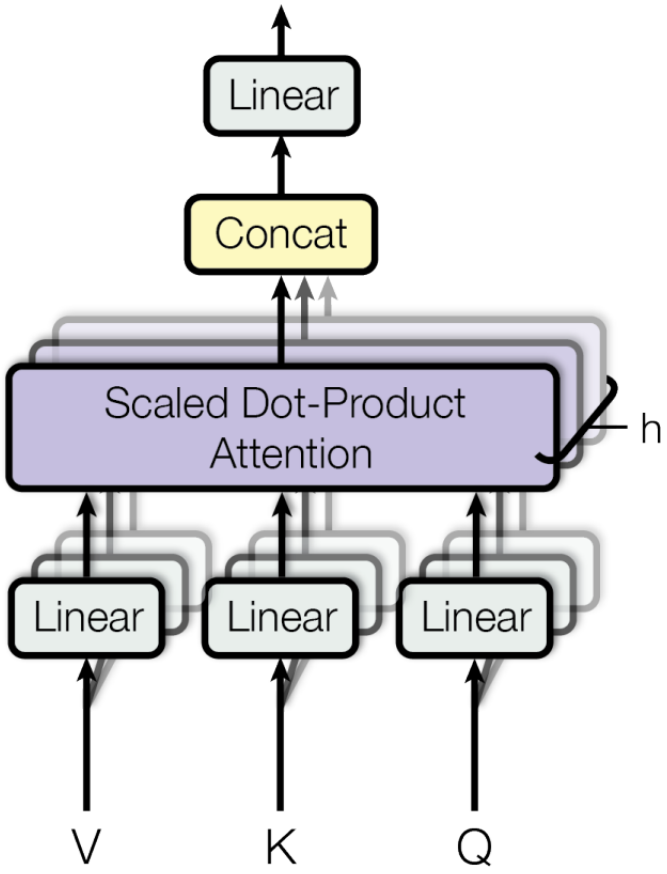

Figure 1.4: Multi-Head Attention consists of attention layer running in paralle

Position wise Feed Forward Neural Network

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically.

This consists of two linear transformations with a ReLU activation in between.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{1.3}$$

Embedding and Softmax

Similarly to other sequence transduction models, Learned embeddings are used to convert the input tokens and output tokens of dimension d_{model}. Usual learned linear transformation and softmax function are used to convert the decoder output to predict next-token probabilities.

Positional Encoding

Since the model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, some information about the relative or absolute position of the tokens in the sequence must be injected. To this end, "positional encodings" are added to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension model as the embeddings, so that the two can be summed. Sine and cosine functions of different frequencies are used for positional encoding as follow

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{model}}}) \tag{1.4}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{model}}}) \tag{1.5}$$

where pos is the position and i is the dimension. That is, each dimension of the positinal encoding corresponds to a sinusoid.

1.2.3 Conclusion and Results

In the machine translation task, the transformer model outperformed all previous state of the art models which have used the RNNs, GRUs and LSTMs. To evaluate if the Transformer can generalize to other tasks, experiments on English constituency parsing were performed and despite the lack of task-specific tuning, the transformer model performed surprisingly well, yielding better results than all previously reported models with the exception of the Recurrent Neural Network Grammar.

In this work, the Transformer, the first sequence transduction model based entirely on attention, replacing the recurrent layers most commonly used in encoder-decoder architectures with multi-headed self-attention is presented.

Till today, various transformer models have been developed which have performed better on all kinds of natural language processing tasks like language modeling, text classifications, questions answering, machine translation, sentence similarity, summarization, etc.

Chapter 2

Word Embeddings

2.1 Introduction

A very basic definition of a word embedding is a real number, vector representation of a word. Typically, these days, words with similar meaning will have vector representations that are close together in the embedding space (though this hasn't always been the case).

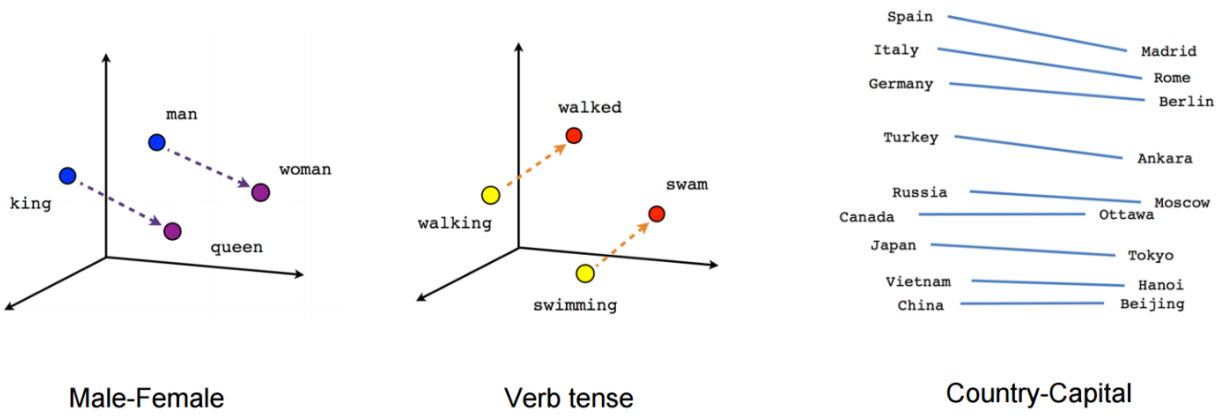

Figure 2.1: Word2Vec

When constructing a word embedding space, typically the goal is to capture some sort of relationship in that space, be it meaning, morphology, context, or some other kind of relationship.

By encoding word embeddings in a densely populated space, we can represent words numerically in a way that captures them in vectors that have tens or hundreds of dimensions instead of millions (like one-hot encoded vectors).

Different word embeddings are created either in different ways or using different text corpora to map this distributional relationship, so the end result are word embeddings that help us on different down-stream tasks in the world of NLP.

2.2 Why Word Embeddings?

Words aren't things that computers naturally understand. By encoding them in a numeric form, we can apply mathematical rules and do matrix operations to them. This makes them amazing in the world of machine learning, especially.

Take deep learning for example. By encoding words in a numerical form, we can take many deep learning architectures and apply them to words. Convolutional neural networks have been applied to NLP tasks using word embeddings and have set the state-of-the-art performance for many tasks.

Moreover, word embeddings can be pre-trained and can be used for variety of NLP applications like text classification, question-answering, auto-complete, spelling correction, speech processing, etc.

2.3 Word Embeddings Methods

2.3.1 One-Hot Encoding(Count Vectorizing)

One of the most basic ways we can numerically represent words is through the one-hot encoding method (also sometimes called count vectorizing).

In one-hot encoding, to represent a word, a vector of dimension equal to the number of unique words in the corpus is created. Each unique word has a unique dimension and will be represented by 1 in that dimension with 0s everywhere else.

Let, **Corpus :** {I have a pen}

Then, the vector representation of words will be

I : [1, 0, 0, 0]

have : [0, 1, 0, 0]

a : [0, 0, 1, 0]

pen : [0, 0, 0, 1]

As in above example, each unique word is assigned a vector with its dimension equal 1 and all other equal 0.

Disadvantage of one-hot encoding is words are represented by huge and sparse vectors that captures no relational information.

2.3.2 Word2Vec

In 2013, with Word2Vec , Mikolov et al. at Google completely changed the embedding paradigm: from then on, embedding will be the weights of a neural network that are adjusted to minimize some loss, depending on the task. Embedding had become a neural network algorithm.

Word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Embeddings learned

through word2vec have proven to be successful on a variety of downstream natural language processing tasks.

There are two major learning approaches for Word2Vec.

Continuous Bag-of-Words (CBOW)

It predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

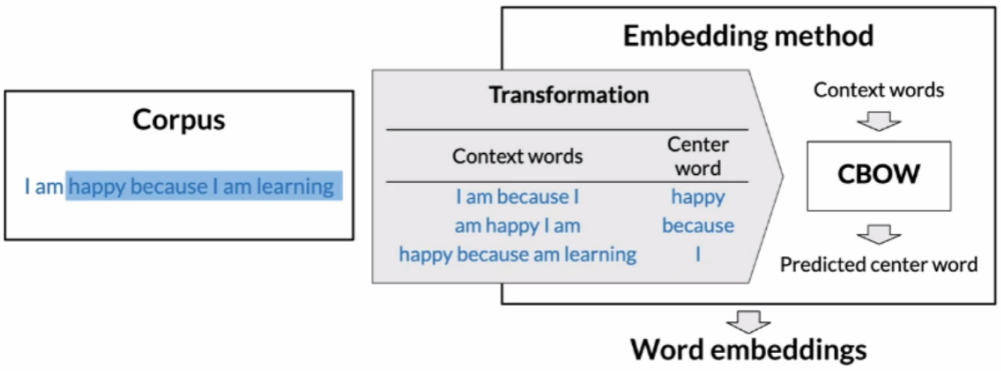

Figure 2.2: CBOW Model

Continuous Skip-Gram

This method learns an embedding by predicting the surrounding words given the context. The context is the current word.

Both of these learning methods use local word usage context (with a defined window of neighboring words). The larger the window is, the more topical similarities that are learned by the embedding. Forcing a smaller window results in more semantic, syntactic, and functional similarities to be learned.

Using Word2Vec model, high quality embeddings can be learned pretty efficiently, especially when comparing against neural probabilistic models. That means low space and low time complexity to generate a rich representation.

More than that, the larger the dimensionality, the more features we can have in our representation. But still, we can keep the dimensionality a lot lower than some other methods. It also allows us to efficiently generate something like a billion word corpora, but encompass a bunch of generalities and keep the dimensionality small.

2.3.3 GloVe

GloVe is an extension of word2vec, and a much better one at that. There are a set of classical vector models used for natural language processing that are good at capturing global statistics of a corpus, like LSA (matrix factorization). They're very good at global information, but they don't capture meanings so well and definitely don't have the cool analogy features built in.

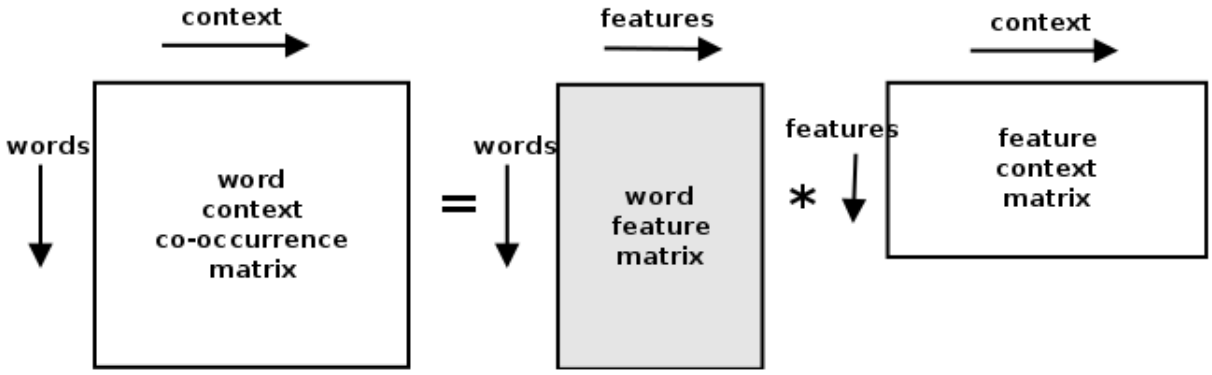

Figure 2.3: Glove Model

GloVe's contribution was the addition of global statistics in the language modeling task to generate the embedding. There is no window feature for local context. Instead, there is a word-context/word co-occurrence matrix that learns statistics across the entire corpora.

2.3.4 Others

Some other word embedding methods are fastText(Facebook, 2016), BERT (Google, 2018), ELMo(Allen Institute for AI, 2018) and GPT-2(OpenAI, 2018).

2.4 Feature Vectors and Similarity Scores

Word vectors represent words in a way that encodes their meaning. A vector is simply an array of fractions. Here's a vector taken from the Google News model. It's the vector for the word "fast". It consists of 300 floating point values: [0.0575, -0.0049, 0.0474, ..., -0.0439]

The straight line distance between two points is called the "Euclidean" or "L2" distance, and it can be extended to any number of dimensions. Here's the formula for the distance between two vectors a and b with an arbitrary number of dimensions (n dimensions):

$$dist_{L2}(a,b) = \sqrt{\sum_{i=0}^{n}(a_i - b_i)^2} \tag{2.1}$$

Here's the important insight, word2vec learns a vector for each word in a vocabulary such that words with similar meanings are close together and words with different meanings are farther apart. That is, the Euclidean distance between a pair of word vectors becomes a measure of how dissimilar they are.

The Euclidean distance is what we intuitively understand as distance, and it's a workable distance metric for comparing word vectors, but in practice another metric called the Cosine similarity gives better results.

$$Similarity(A, B) = \frac{A.B}{||A|| * ||B||} = \frac{\sum_{i=1}^{n} A_i * B_i}{\sum_{i=1}^{n} A_i^2 * \sum_{i=1}^{n} B_i^2} \tag{2.2}$$

The Cosine similarity between two vectors a and b is found by calculating their dot product, and dividing this by their magnitudes. The cosine similarity is always a value between -1.0 and 1.0.

2.5 Data Cleaning and Tokenization

Following details should be followed for proper word embedding while data pre-processing and tokenization.

1. **Letter case :** In enligsh language, the letter case (eg: 'Eagle'or 'eagle') doesn't change the meaning of the words. Hence, it is better to have all data in lower case for word embedding.
2. **Punctuation :** Punctuation symbols like ", ', !, ?, etc. should be handled before word embeddings.
3. **Numbers :** Numbers (0, 1, 2, 3 ....) should be removed before word embeddings if they are not important for semantic meaning.
4. **Special characters :** Special characters like $\delta, \Delta, \alpha, \beta$, etc. should be handled before word embeddings.
5. **Special words :** Special words like '#nlp', emojis, etc. should be cleaned.

2.6 Evaluation of Word Embeddings Model

2.6.1 Extrinsic Evaluation

In extrinsic evaluation, word embeddings are tested on external tasks like named entity recognition, parts-of-speech tagging, etc. It evaluates the actual usefulness of embeddings. However, it is time consuming and more difficult to troubleshoot.

2.6.2 Intrinsic Evaluation

Similarity and analogy test are used for intrinsic evaluation.

Chapter 3

Probabilistic Language Model

3.1 Introduction

Language model is used to estimate the probability of the word sequences and to estimate the probability of a word following the sequence of words. The concept can be used to auto-complete a sentence with most likely suggestions as shown below:

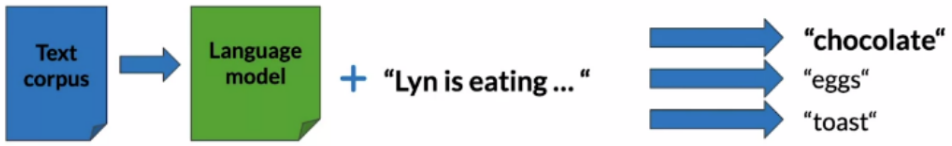

Figure 3.1: Language Model for Auto-complete

Some other applications of the language model are:

1. Speech Recognition
2. Spelling Correction
3. Augmentative Communication

3.2 N-gram

A N-gram is a sequence of N-words. For example:

Corpus : I am happy because I am learning.

Uni-grams : {I, am, happy, because, I, am, learning }

Bi-grams : {I am, am happy, happy because, . . . }

Tri-grams : {I am happy, am happy because, . . . }

3.3 Sequence Notation

Let's define a standard sequence notation to represent the sequence of words in our corpus. Let our corpus be as shown below where n^{th} is represented as W_n. Let the number of words in the corpus be m = 500

Corpus: This is great. ... teacher drinks tea.

Hence, W_1 = This, W_2 = is, W_3 = great and so on...

Let us represent the sequence of words as W_s^e where s represent the starting index and e represent the ending index of the words in the corpus. So,

$W_1^m = W_1\ W_2\ W_3$ ..... $W_{m-1}\ W_m$

Similarly, $W_1^3 = W_1\ W_2\ W_3$ = This is great

3.4 N-gram Probability

N-gram probability is the probability of the occurrence of the sequence of N-words in a corpus. From uni-gram, bi-gram and tri-gram probability, we can deduce for N-gram probability denoted by $P(W_N|W_1^{N-1})$ as

$$P(W_N|W_1^{N-1}) = \frac{C(W_1^{N-1}\,W_N)}{C(W_1^{N-1})} = \frac{C(W_1^N)}{C(W_1^{N-1})} \tag{3.1}$$

3.4.1 Uni-gram Probability

Uni-gram Probability is the probability of occurrence of a word in a given corpus.

Corpus : I am happy because I am learning.

Size of corpus : m = 7

$$P(I) = \frac{2}{7} \tag{3.2}$$

$$P(happy) = \frac{1}{7} \tag{3.3}$$

Hence, we can see that Probability of uni-gram is given by:

$$P(w) = \frac{C(w)}{m} \tag{3.4}$$

where C(w) = frequency of the word in the corpus

3.4.2 Bi-gram Probability

Bi-gram Probability is the probability of occurrence of sequence of two words in a corpus.
Corpus : I am happy because I am learning.
Size of corpus : m = 7

$$P(am|I) = \frac{C(I\,am)}{C(I)} = \frac{2}{2} = 1 \tag{3.5}$$

$$P(happy|I) = \frac{C(I\,happy)}{C(I)} = \frac{0}{7} = 0 \tag{3.6}$$

$$P(learning|am) = \frac{C(am\,learning)}{C(am)} = 1/2 \tag{3.7}$$

Hence, we can see that Probability of bi-gram is given by:

$$P(y|x) = \frac{C(x\,y)}{C(x)} \tag{3.8}$$

where C(w) = frequency of the word in the corpus

3.4.3 Tri-gram Probability

Tri-gram Probability is the probability of the occurrence of the sequence of three words in a corpus.
Corpus : I am happy because I am learning.
Size of corpus : m = 7

$$P(happy|I\,am) = \frac{C(I\,am\,happy)}{C(I\,am)} = \frac{1}{2} \tag{3.9}$$

Hence, we can see that Probability of bi-gram is given by:

$$P(W_3|W_1^2) = \frac{C(W_1^2\,W_3)}{C(W_1^2)} = \frac{C(W_1^3)}{C(W_1^2)} \tag{3.10}$$

where C(w) = frequency of the word in the corpus

3.5 Probability of a sequence

From the conditional probability and chain rule, we have

$$P(B|A) = \frac{P(A,B)}{P(A)} \tag{3.11}$$

From (3.11), we get

$$P(A,B) = P(A).P(B|A) \tag{3.12}$$

Using (3.12), we can deduce

$$P(A,B,C,D) = P(A).P(B|A).P(C|A,B).P(D|A,B,C) \tag{3.13}$$

Now, the probability of the sequence of words: "the teacher drinks tea" is given by:

$$\begin{aligned} &P(the\,teacher\,drinks\,tea) = P(the).P(teacher|the) \\ &P(drinks|the\,teacher).P(tea|the\,teacher\,drinks) \end{aligned} \tag{3.14}$$

3.6 Approximation of Sequence Probability

When using n-gram language model, the exact sentence for the required n-gram might not be present exactly in corpus due to which the frequency of the words sequence becomes zero and hence the probability. For example:

Input : the teacher drinks tea

P(the teacher drinks tea) = P(the) . P(teacher | the) P(drinks | the teacher) . P(tea | the teacher drinks)

Here,

$$P(tea|the\,teacher\,drinks) = \frac{C(the\,teacher\,drinks\,tea)}{C(the\,teacher\,drinks)} \tag{3.15}$$

In the above equation (3.15), the frequency of "the teacher drinks tea" and "the teacher drinks" are both likely 0.

Hence, in order to calculate the required probability of the sequence of the words, as per Markov assumption, only last N words matter. Suppose, only last one words matter then, we have

P(the teacher drinks tea)
= P(the) . P(teacher | the) P(drinks | the teacher) . P(tea | the teacher drinks)
≈ P(the) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

Hence, we can model entire sentence with n-gram following Markov Assumption as follow

$$P(W_n|W_1^{n-1}) \approx P(W_n|W_{n-N+1}^{n-1}) \tag{3.16}$$

3.7 Starting and Ending Sentences

3.7.1 Start of sentence symbol $\langle s \rangle$

For an input sentence, "the teacher drinks tea", we have

P(the teacher drinks tea) ≈ P(the) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

After, we add a start token $\langle s \rangle$, we have

Input : $\langle s \rangle$ the teacher drinks tea

P(the teacher drinks tea) ≈ P(the | $\langle s \rangle$) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

In this way, we can add start token to determine the possible starting words and calculate the probability of sequence considering first word in N-gram language model. For Tri-gram and N-gram, we add (N - 1) start token $\langle s \rangle$ in the beginning of the sequence.

3.7.2 End of sentence symbol $\langle /s \rangle$

For an input sentence, "the teacher drinks tea", we have

P(the teacher drinks tea) ≈ P(the) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

After, we add a start token $\langle s \rangle$ and an end token $\langle /s \rangle$, we have

Input : $\langle s \rangle$ the teacher drinks tea $\langle /s \rangle$

P(the teacher drinks tea) $\approx$ P(the | $\langle s \rangle$) . P(teacher | the) P(drinks | teacher) . P(tea | drinks) . P ($\langle /s \rangle$ | tea)

For bi-gram, tri-gram or any N-gram model, we add end token just once at the end of sentence unlike start token

3.8 Count Matrix

Count matrix is the matrix containing the frequency of occurrence of N-gram. Rows in count matrix represent the unique corpus of (N - 1) gram and columns represent the unique corpus word.

Corpus : $\langle s \rangle$ I study I learn $\langle /s \rangle$

Bi-gram Count Matrix is as follow:

	$\langle s \rangle$	$\langle /s \rangle$	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

Table 3.1: Count Matrix for bi-gram model

3.9 Probability Matrix

Probability matrix is the matrix containing the probability of the occurrence of N-gram words sequence. It can be created by dividing each elements of the count matrix by total sum of all row elements. For example:

Corpus : $\langle s \rangle$ I study I learn $\langle /s \rangle$

	$\langle s \rangle$	$\langle /s \rangle$	I	study	learn	sum
$\langle s \rangle$	0	0	1	0	0	1
$\langle /s \rangle$	0	0	0	0	0	0
I	0	0	0	1	1	2
study	0	0	1	0	0	1
learn	0	1	0	0	0	1

Table 3.2: Count Matrix for bi-gram model with 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

Table 3.3: Probability Matrix for bi-gram model

From the above probability matrix, we can calculate the probability of sentence and also predict the next word based on higher probability.

For example :

Input : $\langle s \rangle$ I learn $\langle /s \rangle$

P(Input) = P($\langle s \rangle$ I learn $\langle /s \rangle$)

= P(I | $\langle s \rangle$) . P(learn | I). P ($\langle /s \rangle$ | learn)

= $1 \times 0.5 \times 1$

= 0.5

3.10 Generative Language Model

Generative language model is used to generate a most likely sentences.

Algorithm for the generative language model is:

Step 1 : Choose sentence start

Step 2 : Choose next bi-gram starting with the previous word

Step 3 : Continue until $\langle /s \rangle$ is picked

For example:

Assume our corpus be:

$\langle s \rangle$ Lyn drinks chocolate $\langle /s \rangle$

$\langle s \rangle$ John drinks tea $\langle /s \rangle$

$\langle s \rangle$ Lyn eats chocolate $\langle /s \rangle$

Then, the generative language model can generate a sentence as follow:

Step 1: ($\langle s \rangle$, Lyn) or ($\langle s \rangle$, John)

Step 2: (Lyn, eats) or (Lyn drinks)

Step 3: (drinks, tea) or (drinks, chocolate)

Step 4: (tea, $\langle /s \rangle$)

3.11 Train, Validation and Test Split

The recommended split of corpus into train, validation and test data is as follow

Type of corpus	Train	Validation	Test
Small	80%	10%	10%
Large	98%	1%	1%

Table 3.4: Train, Validation and Test Split

3.12 Language Model Evaluation

We can use two different approaches to evaluate and compare language models:

3.12.1 Extrinsic evaluation

This involves evaluating the models by employing them in an actual task (such as machine translation) and looking at their final loss/accuracy. This is the best option as it's the only way to tangibly see how different models affect the task we're interested in. However, it can be computationally expensive and slow as it requires training a full system.

3.12.2 Intrinsic evaluation

This involves finding some metric to evaluate the language model itself, not taking into account the specific tasks it's going to be used for. While intrinsic evaluation is not as "good" as extrinsic evaluation as a final metric, it's a useful way of quickly comparing models. Perplexity is an intrinsic evaluation method.

3.12.3 Perplexity

Perplexity is an evaluation metric for language models. For better language model, perplexity value should be smaller. Perplexity can be calculated as follow:

$$PP(W) = P(s_1, s_2, s_3, s_4, ..., s_m)^{\frac{-1}{m}} \tag{3.17}$$

where,

$\mathbf{W}$ = test set containing m sentences s

$S_i = i^{th}$ sentence in the test set, each ending with $\langle /s \rangle$

$\mathbf{m}$ = number of all words in entire test set $\mathbf{W}$ including $\langle /s \rangle$ but not including $\langle s \rangle$

Similarly, Perplexity for the bi-gram model can be calculated as:

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m} \prod_{j=1}^{|S_i|} \frac{1}{P(w_j^{(i)} | w_{j-1}^{(i)})}} \tag{3.18}$$

where, $w_j^{(i)} = j^{th}$ word in i^{th} sentence

If we concatenate all the sentences in W, we can get perplexity as

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m} \frac{1}{P(w_i|w_{i-1})}} \tag{3.19}$$

where, $w_i = i^{th}$ word in test set

3.12.4 Log Perplexity

Log perplexity is calculated by taking logarithm on both sides of equation (3.19) and we get

$$logPP(W) = \frac{-1}{m} \sum_{i=1}^{m} log_2(P(w_i|w_{i-1})) \tag{3.20}$$

3.13 Vocabulary

Vocabulary is a set of words from the training corpus. We can create vocabulary from training corpus using following criteria.

1. Min word frequency f
2. Max $|V|$, include words by frequency
3. Using $\langle UNK \rangle$ for unknown token

For example: Suppose our corpus is as follow

$\langle s \rangle$ Lyn drinks chocolate $\langle /s \rangle$

$\langle s \rangle$ John drinks tea $\langle /s \rangle$

$\langle s \rangle$ Lyn eats chocolate $\langle /s \rangle$

let min frequency to be added in vocabulary be f = 2. So, now corpus will be as follow:

$\langle s \rangle$ 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 $\langle /s \rangle$

Hence, vocabulary = $\{Lyn, drinks, chocolate\}$

Suppose,

Input Query : $\langle s \rangle$ Adam drinks chocolate $\langle /s \rangle$

So, the input query after processing becomes

Ouput : $\langle s \rangle$ $\langle UNK \rangle$ drinks chocolate $\langle /s \rangle$

In this way, vocabulary can be created and unknown words can be handled.

3.14 Missing N-gram in training corpus

N-grams made of known words still might be missing in the training corpus. N-gram probability is given as:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})} \tag{3.21}$$

In equation (3.21), both $C(w_{n-N+1}^{n-1}, w_n)$ and $C(w_{n-N+1}^{n-1})$ can be 0 in the probability matrix which causes the probability to be indefinite. It can be solved by using smoothing.

3.14.1 Laplacian Smoothing

In Laplacian smoothing, probability is calculated as follow:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V} \tag{3.22}$$

3.14.2 K-smoothing

In K-smoothing, probability is calculated as follow

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + K}{C(w_{n-1}) + k * V} \tag{3.23}$$

Chapter 4

Sequence Modelling

4.1 Introduction

Sequence Modelling is the ability of a computer program to model, interpret, make predictions about or generate any type of sequential data, such as audio, text etc. For example, a computer program that can take a piece of text in English and translate it to French is an example of a Sequence Modelling program (because the type of data being dealt with is text, which is sequential in nature). An AI algorithm called the Recurrent Neural Network, is a specialised form of the classic Artificial Neural Network (Multi-Layer Perceptron) that is used to solve Sequence Modelling problems. Recurrent Neural Networks are like Artificial Neural Networks which has loops in them. This means that the activation of each neuron or cell depends not only on the current input to it but also its previous activation values.

4.2 Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As we read a text, we understand each word based on your understanding of previous words. We don't throw everything away and start thinking from scratch again. Our thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

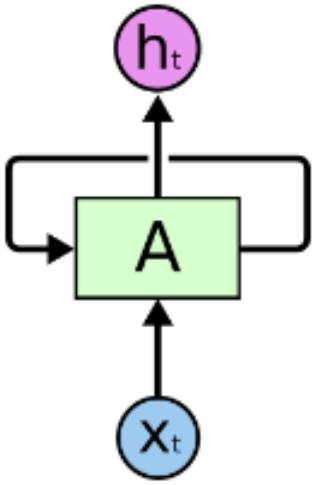

Figure 4.1: Recurrent Neural Networks

In the above diagram, a chunk of neural network, A, looks at some input xt and outputs a value ht. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

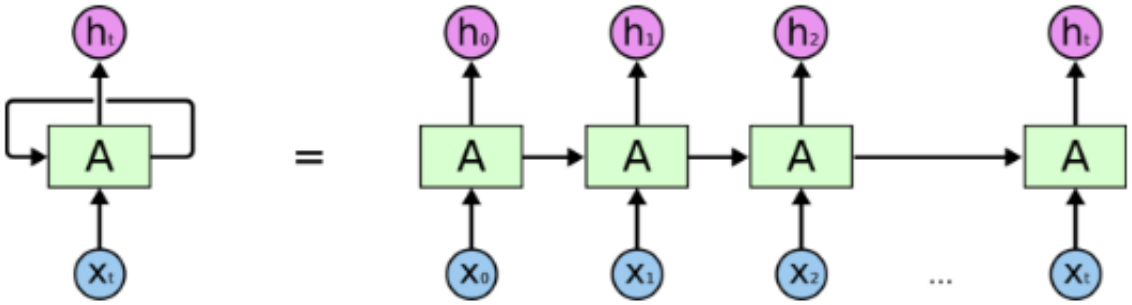

Figure 4.2: An unrolled recurrent neural networks

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning, etc.

4.3 Vanishing Gradients with RNNs

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the sky," we don't need any further context - it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

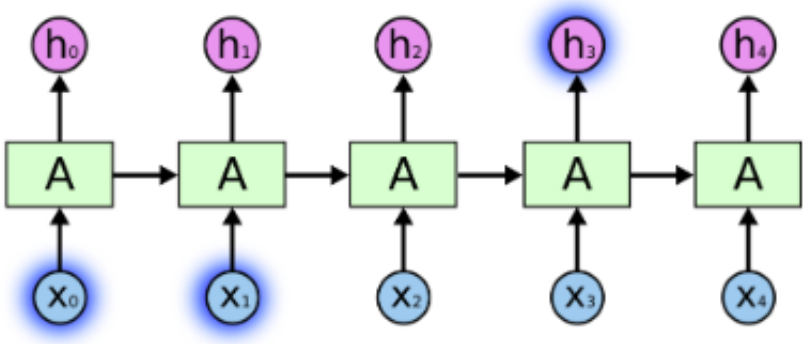

Figure 4.3: Short Sequence in RNNs

But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

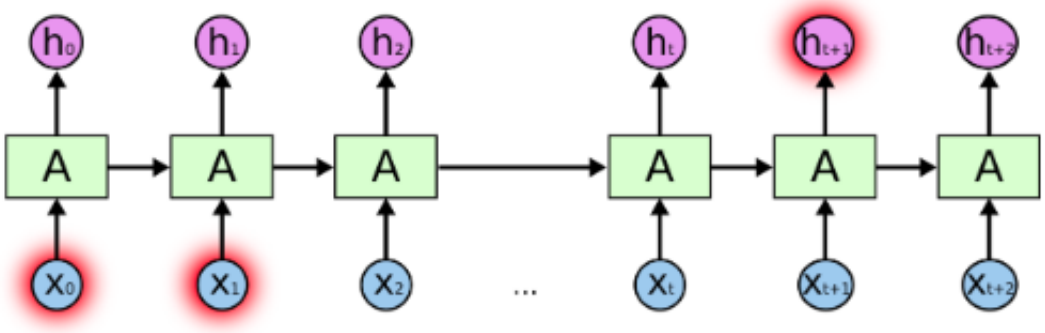

Figure 4.4: Long Sequence in RNNs

In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice,

RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

4.4 LSTMs

Long Short Term Memory networks - usually just called "LSTMs" - are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter and Schmidhuber (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer as shown below.

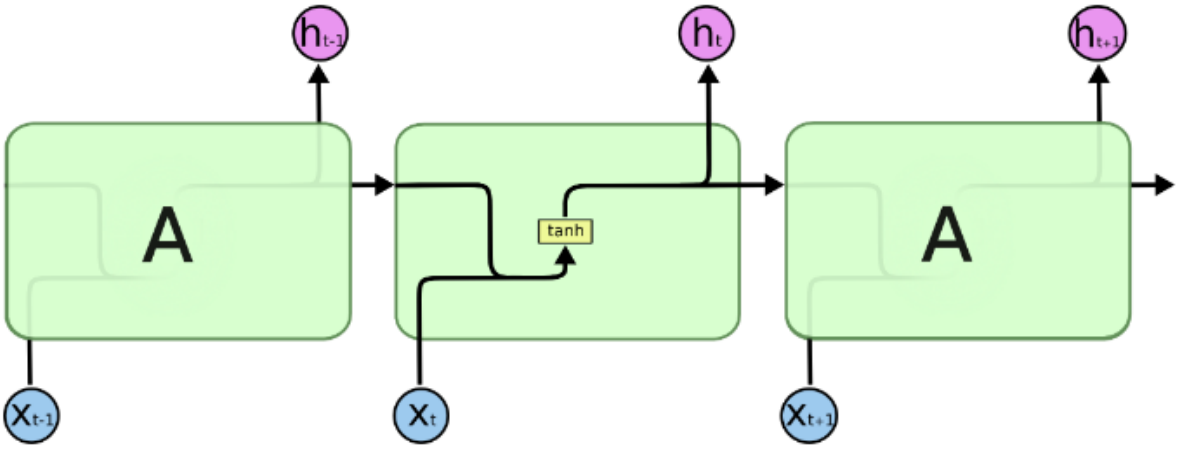

Figure 4.5: The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

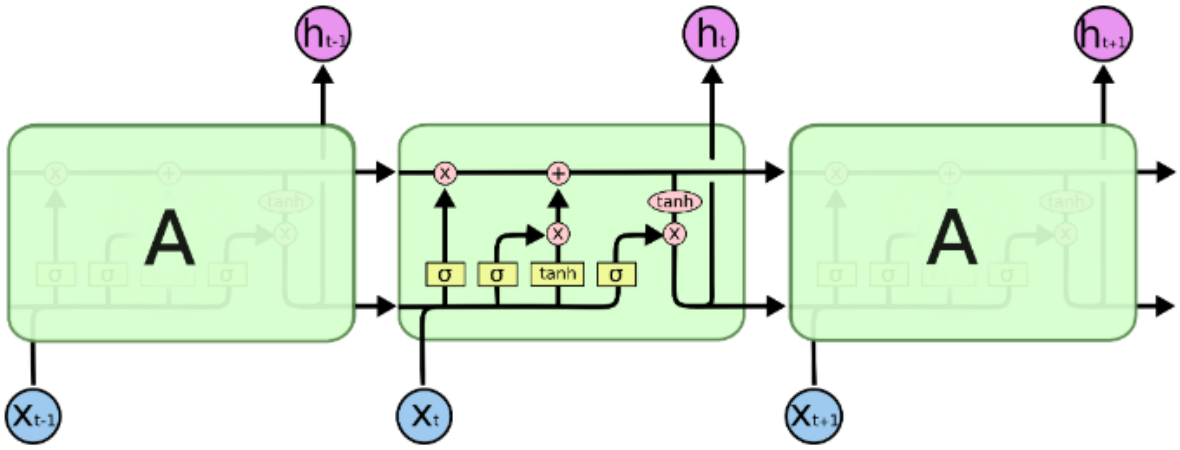

Figure 4.6: The repeating module in an LSTM contains four interacting layers.

4.4.1 LSTM Cell

The following figure shows the operations of an LSTM cell:

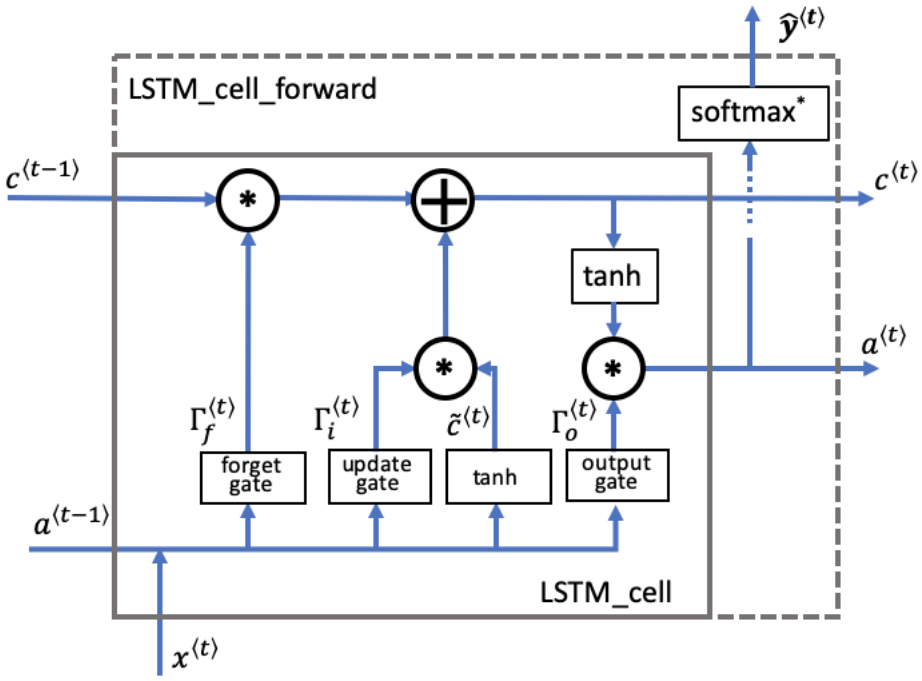

Figure 4.7: LSTM Cell

Forget Gate Γ_f

Assume, we are reading words in a piece of text and plan to use an LSTM to keep track of grammatical structures, such as whether the subject is singular("dog") or plural("dogs). If the subject changes its state (from a singular word to a plural word), the memory of the previous state becomes outdated, so it is better to forget that outdated state.

The forget gate is a tensor containing values between 0 and 1. If a unit in the forget gate has a value close to 0, the LSTM will forget the stored state in the corresponding unit of the previous cell state. If a unit in the forget gate has a value close to 1, the LSTM will mostly remember the corresponding value in the stored state.

$$\Gamma_f^{<t>} = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \tag{4.1}$$

where,

- W_f = forget gate weight W_f
- b_f = forget gate bias b_f
- $\Gamma_f^{<t>}$ = Forget Gate

Explanation of the equation:

- W_f contains the weights that govern the forget gate's behavior.
- The previous time step's hidden state $a^{<t-1>}$ and current time step's input $x^{<t>}$ are concatenated together and multiplied by W_f.
- A sigmoid function is used to make each of the gate tensor's values $\Gamma_f^{<t>}$ range from 0 to 1.
- The forget gate $\Gamma_f^{<t>}$ has the same dimensions as the previous cell state $c^{<t-1>}$.
- Multiplying the tensors $\Gamma_f^{<t>} * c^{<t-1>}$ is like applying a mask over the previous cell state.
- If a single value in $\Gamma_f^{<t>}$ is 0 or close to 0, then the product is close to 0. This keeps the information stored in the corresponding unit in $c^{<t-1>}$ from being remembered for the next time step.
- Similarly, if one value is close to 1, the product is close to the original value in the previous cell state. The LSTM will keep the information from the corresponding unit of $c^{<t-1>}$, to be used in the next time step.

Candidate Value $\tilde{c}^{<t>}$

The candidate value is a tensor containing information from the current time step that may be stored in the current cell state $c^{<t>}$. The parts of the candidate value that get passed on depend on the update gate. The candidate value is a tensor containing values that range from -1 to 1. The tilde " " is used to differentiate the candidate $\tilde{c}^{<t>}$ from the cell state $c^{<t>}$.

$$\tilde{c}^{<t>} = tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \tag{4.2}$$

Update Gate Γ_i

Update gate can be used to decide what aspects of the candidate $\tilde{c}^{<t>}$ to add to the cell state $c^{<t>}$. The update gate is a tensor containing values between 0 and 1. When a unit in the update gate is close to 1, it allows the value of the candidate $\tilde{c}^{<t>}$ to be passed onto the hidden state

$c^{<t>}$. When a unit in the update gate is close to 0, it prevents the corresponding value in the candidate from being passed onto the hidden state.

$$\Gamma_i^{<t>} = \sigma(W_i[a^{<t-1>}, x^{<t>}] + b_i) \tag{4.3}$$

Explanation of the equation:

- Similar to the forget gate, here $\Gamma_i^{<t>}$, the sigmoid produces values between 0 and 1.
- The update gate is multiplied element-wise with the candidate, and this product $\Gamma_i^{<t>} * \tilde{c}^{<t>}$ is used in determining the cell state $c^{<t>}$.

Cell State $c^{<t>}$

The cell state is the "memory" that gets passed onto future time steps. The new cell state $c^{<t>}$ is a combination of the previous cell state and the candidate value.

$$c^{<t>} = \Gamma_f^{<t>} * c^{<t-1>} + \Gamma_i^{<t>} * \tilde{c}^{<t>} \tag{4.4}$$

Explanation of the equation

- The previous cell state $c^{<t-1>}$ is adjusted (weighted) by the forget gate $\Gamma_f^{<t>}$.
- the candidate value $\tilde{c}^{<t>}$ is adjusted (weighted) by the update gate $\Gamma_i^{<t>}$.

Output Gate Γ_o

The output gate decides what gets sent as the prediction (output) of the time step. The output gate is like the other gates, in that it contains values that range from 0 to 1.

$$\Gamma_o^{<t>} = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \tag{4.5}$$

Explanation of the equation

- The output gate is determined by the previous hidden state $a^{<t-1>}$ and the current input x^t
- The sigmoid makes the gate range from 0 to 1.

Hidden State $a^{<t>}$

The hidden state gets passed to the LSTM cell's next time step. It is used to determine the three gates $(\Gamma_f, \Gamma_u, \Gamma_o)$ of the next time step. The hidden state is also used for the prediction $y^{<t>}$.

$$a^{<t>} = \Gamma_o^{<t>} * tanh(c^{<t>}) \tag{4.6}$$

Explanation of the equation

- The hidden state $a^{<t>}$ is determined by the cell state $c^{<t>}$ in combination with the output gate Γ_o.
- The cell state state is passed through the tanh function to rescale values between -1 and 1.
- The output gate acts like a "mask" that either preserves the values of $\tanh(c^{<t>})$ or keep those values from being included in the hidden state $a^{<t>}$.

Prediction $y_{pred}^{<t>}$

The prediction in this use case is a classification, so you'll use a softmax.

$$y_{pred}^{<t>} = softmax(W_y a^{<t>} + b_y) \tag{4.7}$$

4.5 Bidirectional RNNs

A typical state in an RNN (simple RNN, GRU, or LSTM) relies on the past and the present events. A state at time t depends on the states x_1, x_2, ..., x_{t-1} and x_t. However, there can be situations where a prediction depends on the past, present, and future events.

For example, predicting a word to be included in a sentence might require us to look into the future, i.e., a word in a sentence could depend on a future event. Such linguistic dependencies are customary in several text prediction tasks.

Thus, capturing and analyzing both past and future events is helpful in the above-mentioned scenarios.

To enable straight (past) and reverse traversal of input (future), Bidirectional RNNs, or BRNNs, are used. A BRNN is a combination of two RNNs - one RNN moves forward, beginning from the start of the data sequence, and the other, moves backward, beginning from the end of the data sequence. The network blocks in a BRNN can either be simple RNNs, GRUs, or LSTMs.

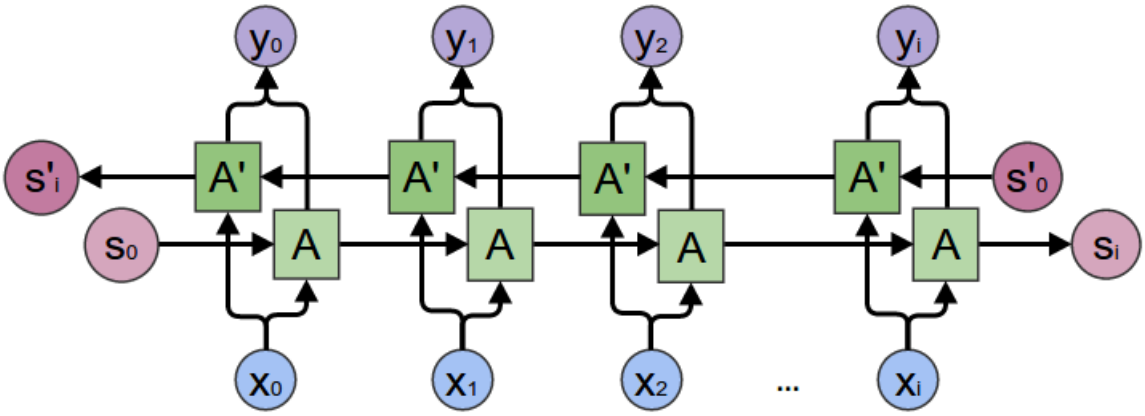

Figure 4.8: Bidirectional RNN

4.6 Sequence Modelling Applications

Some applications of sequence modelling are:

- **Language Modelling** : Auto-generation of next probable word.(Previous sequence of words are sequential data)
- **Image Captioning** (with the help of computer vision): Generating captions for images. (captions are sequential data)
- **Video Frame Prediction** (with the help of computer vision): Predict the subsequent frames of video given the previous ones. (the frames in a video are sequential in nature)
- **Classifying songs** (audio) as Jazz, Rock , Pop etc (genre). Here, audio is of sequential nature.
- **Composing Music** (music is sequential in nature)