Complexity, Learnability & French-Chinese MT

Internship Report
June - August 2022

— M1AM Hanning YANG —
Université Grenoble Alpes - France
LIG/GETALP/Ch. Boitet

Contents

Abstract

This is a report on my M1-MSIAM internship (Applied Mathematics program). I studied Machine Translation in general, French-Chinese in particular. I learnt different architectures of Machine Translation systems and some mathematical aspects of Statistical Machine Translation and Neural Machine Translation which is illustrated by Long Short-Term Memory. Moreover, I implemented a Sequence to Sequence model in Pytorch to test various recent improvements. Following is the plan of the internship in the beginning.

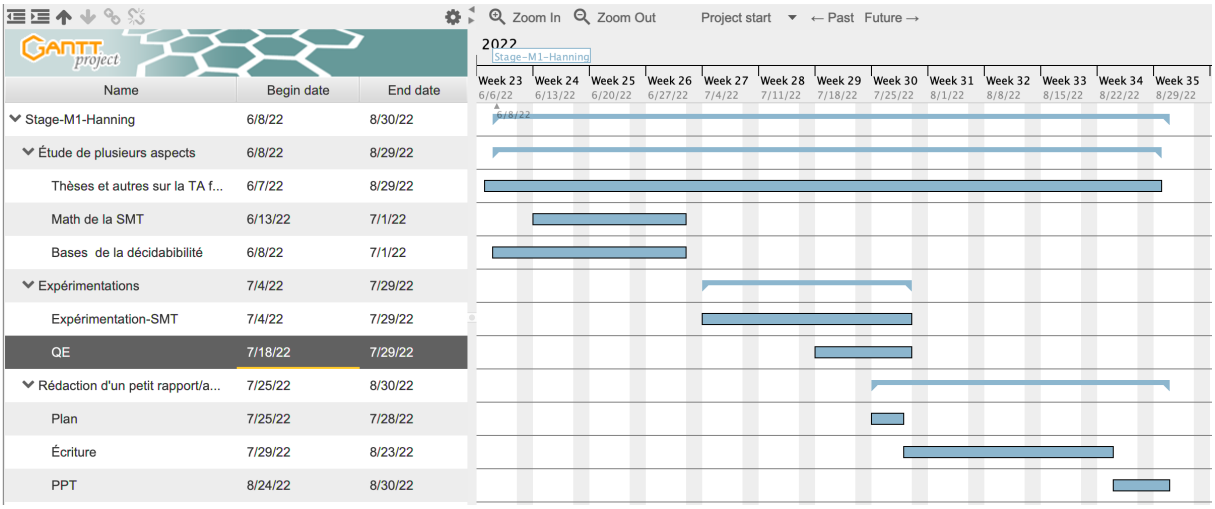

Figure 1: Gantt plan in the beginning

Introduction

This internship is for M1 Applied Mathematics. The requirement for the internship is that it should be related to at least one course of our program, which means that I should implement mathematical methods during the process and gain some mathematical understanding. It was difficult for me to find an internship. I ended up finding an internship in GETALP laboratory by emailing MIAI.

The interesting part of this internship is that I learnt two kinds of machine translation, which are statistical machine translation (SMT) and neural machine translation (NMT). In SMT, I gained insights into parameter estimate. Unfortunately, I wasn't able to realize a model in Moses due to technical reasons. However, I moved forward to NMT. I grasped the idea of NMT and realized a model in Pytorch.

This report will be divided into three parts, which are computability theory, SMT and NMT respectively. Computability theory will mainly talk about primitive recursive functions, Turing machine and recursion theory. In SMT, a fundamental equation will be introduced first and N-Gram followed. NMT will have two main parts. They are Seq2Seq model and experiment part.

Part I

Computability Theory

Chapter 1

Recursive Functions

Introduction

In this chapter, a formal characterization of recursive function will be introduced. With this will come a mathematical characterization of a class of objects which are called Turing machines. The use of Turing machines helps to explain what computation is by demarcating the so-called "computable functions". Then we will study recursion theory to study the notion of computability. To progress this chapter, we have studied the following source: [3][7].

Contents

1.1 Primitive Recursive Functions

In this chapter, a formal characterization of recursive function will be introduced. With this will come a mathematical characterization of a class of objects which are called Turing machines. The use of Turing machines helps to explain what computation is by demarcating the so-called "computable functions". Then we will study recursion theory to study the notion of computability.

1.1.1 Basic Functions

- **The zero function**: $(\forall\ m, n, [m^{(n)} \in RP])\ 0^{(0)}$ is enough.
- **The successor function**: $s = \lambda x[x+1]$
- **The projection function**: $a^{(n)}_{i_1 \leq i \leq n} = \lambda x_1 ... x_n[x_i]$

1.1.2 Composition Operator

Suppose $\lambda y_1 ... y_k f(y_1,\ ...,\ y_k)$ is a k-ary total function and $\lambda \overrightarrow{x}\, g_1(\overrightarrow{x}),\ ...\ \lambda \overrightarrow{x}\, g_k(\overrightarrow{x})$ are n-ary total functions.The composition operator σ is defined by

$$(\forall\ \overrightarrow{x}) \quad [h(\overrightarrow{x}) = \sigma(f^{(k)}(g_1^{(\overrightarrow{x})}, ..., g_k^{(\overrightarrow{x})}))]$$

1.1.3 Primitive Recursion

Suppose that $g^{(n)}$ is an n-ary function and $h^{(n+2)}$ is an $(n+2)$-ary function. The recursive function $f^{(n+1)}$ is defined by

$$f(0, \overrightarrow{x}) = g(\overrightarrow{x}) \qquad (1)$$
$$f(y+1, \overrightarrow{x}) = h(y, f(y, \overrightarrow{x}), \overrightarrow{x}) \qquad (2)$$

There is a unique function $f^{(n+1)} = \rho[g^{(n)}, h^{(n+2)}]$ that satisfies (1) and (2), demonstrated by noetherian induction on $\mathbb{N}^{n+1}$.

1.1.4 Primitive Recursive Function

The set of primitive recursive function is the least set generated from the initial functions, composition and recursion.

1.1.5 Example

- Define the sum as $f = \lambda xy[x+y]$
  Proof:
  $f(0,x) = x = a^{(1)}_1(x)$
  $f(y+1+x) = (x+y)+1 = s(x+y) = h(y, f(y,x), x)$
  $h(y, f(y,x), x) = s(f(y,x))$
  If $f = \rho[a^{(1)}_1, \sigma[s, a^{(2)}_3]]$
  $g = a^{(1)}_1$
  $h = \sigma[s, a^{(2)}_3]$

1.2 Turing Machines

1.2.1 Motivation

The primary motivation behind the study about Turing Machines is the motive to capture the notion of computability. By the term "computability", we mean whether for a given problem, does there exist an algorithm that can solve it. i.e. what all computational problems can be solved?

1.2.2 Definition

The symbols $q_0, q_1, q_2, q_3, ...$ will be regarded as denoting internal states; the symbols $S_0, S_1, S_2, ...$ will be regarded as symbols which various machines may be capable of printing; the symbols R and L will represent a move of one square to the right and one square to the left, respectively.
An **tape** is a finite sequence (possibly empty) of symbols chosen from the list: $q_1, q_2, q_3, q_4, ...$; $S_0, S_1, S_2, ...$; R, L.
A **quadruple** is an expression having one of the following forms:
(1) $q_i\ S_j\ S_k\ q_l$
(2) $q_i\ S_j\ R\ q_l$
(3) $q_i\ S_j\ L\ q_l$
(4) $q_i\ S_j\ q_k\ q_l$
A Turing machine will be defined so as to consist entirely of quadruples. A quadruple of the form 1,2 or 3 above specifies the next act of a Turing machine when in internal configuration q_i and scanning a square on which appears the symbol S_j. Quadruple 1 indicates that the next act is to replace S_j by S_k on the scanned square and to enter internal configuration q_l. Quadruple 2 indicates motion of one square to the right followed by entry into internal configuration q_l. Quadruples of type 3 similarly indicate motion leftward.
A **Turing machine** Z is a finite (nonempty) set of quadruples that contains no two quadruples whose first two symbols are the same.
The symbols S_0 will also be written B. and S_1 will be written |.
With each number n we associate the tape expression $\overline{n}$ where $\overline{n} = |^{n+1}B$.
With each k-tuple $(n_1, n_2, ..., n_k)$ of integers we associate the tape expression $(\overline{n_1, n_2, ..., n_k})$, where

$$(\overline{n_1, n_2, ..., n_k}) = \overline{n_1}B\overline{n_2}B \dots B\overline{n_k}B$$

Let Z be a Turing machine. Then, for each n, we associate with Z an n-ary function

$$\Psi_Z^{(n)}(x_1, x_2, ..., x_n)$$

An n-ary function $\lambda x_1 \cdots x_n[f(x_1, ..., x_n)]$ is partially computable if there exists a Turing machine Z such that

$$(\forall \overrightarrow{x}) f(\overrightarrow{x}) = \begin{cases} \Psi_Z^{(n)}(\overrightarrow{x}) & \text{if and when the computation halts} \\ \downarrow & \text{otherwise} \end{cases}$$

In this case we say that Z computes f. If, in addition, f is a total function, then f is called computable.

1.2.3 A Simple Example

Addition. Let $f = \lambda xy[x+y]$. We shall construct a Turing machine Z which computes f, that is

$$\forall\ x, y, \quad \Psi_Z^{(2)}(x, y) = x + y$$

We take Z to consist of the quadruples

$$\begin{array}{llll} q_1 & | & B & q_1 \\ q_1 & B & R & q_2 \\ q_2 & | & R & q_2 \\ q_2 & B & R & q_3 \\ q_3 & | & B & q_3 \end{array}$$

1. Let's start with the configuration, $q_0(\overline{m_1, m_2}) = q_0\overline{m_1}B\overline{m_2} = q_0||^{m_1}B||^{m_2}$.

2. Then

$$\begin{aligned} q_0\overline{m_1}B\overline{m_2} &= q_0||^{m_1}B||^{m_2} \\ &\vdash q_0B|^{m_1}B||^{m_2} \\ &\vdash Bq_1|^{m_1}B||^{m_2} \\ &\vdash ... \\ &\vdash B|^{m_1}q_1B||^{m_2} \\ &\vdash B|^{m_1}Bq_2||^{m_2} \\ &\vdash B|^{m_1}Bq_2B|^{m_2} \end{aligned}$$

3. If it converges, the output will be $\rightarrow B|^{m_1}Bq_3B|^{m_2}$. Thus, the result is

$$\begin{aligned} \Psi_Z^{(2)}(m_1, m_2) &=< B|^{m_1}Bq_3B|^{m_2} > \\ &= m_1 + m_2 \end{aligned}$$

1.3 Fundamental Theorems

1.3.1 Gödel Index

We see a number as an index for a problem/function if it is the Gödel number of a programme that solves/calculates the problem/function.

1.3.1.1 Gödel Index for Computable Function

Suppose $f^{(n)}$ ia an n-ary computable function. A number a is an index for $f^{(n)}$ if $f^{(n)} = \varphi_a^{(n)}$.

1.3.1.2 Padding Lemma

Each partial recursive function has $\aleph_0$ distinct indices.

1.3.2 Universal Turing Machine

A Universal Turing Machine (UTM) is a Turing machine which works as follows: Taking (e, x) as input, it simulates the action of T_e on input x.

1.3.3 s-m-n Theorem

s-m-n Theorem: For every $m, n \geq 1$, there exists a recursive function s_n^m of $m+1$ variables such that for all $x, y_1, ..., y_m$,

$$\lambda z_1 ... z_n [\varphi_x^{(m+n)}(y_1, ..., y_m, z_1, ..., z_n)] = \varphi^{(n)}_{s_n^m(x, y_1, ..., y_m)}$$

In practical terms, the theorem says that for a given programming language and positive integers m and n, there exists a particular algorithm that accepts as input the source code of a program with $m+n$ free variables, together with m values. This algorithm generates source code that effectively substitutes the values for the first m free variables, leaving the rest of the variables free.

1.3.4 Enumeration Theorem

Enumeration Theorem:

$$(\forall n \in \mathbb{N})(A \subseteq \mathbb{N})(\forall x \in \mathbb{N})\ [\lambda \overrightarrow{y} \cdot \varphi_z^{A;n+1}(x, \overrightarrow{y}) = \varphi_x^{A;n}]$$

There are universal programs that simulate all the programs. A program is universal if upon receiving the Gödel number of a program it simulates the program indexed by the number.

Consider the function $\lambda xy[\psi(x, y)]$ defined as follows

$$\lambda xy[\psi(x, y)] = \varphi_x(y)$$

In an obvious sense $\lambda x[\psi(x, \_)]$ is a universal function for the unary functions

$$\varphi_0, \varphi_1, \varphi_2, \varphi_3, ...$$

The **universal function** for n-ary computable functions is the $(n+1)$-ary function $\psi_U^{(n)}$ defined by

$$\lambda e x_1 ... x_n [\psi_U(e, x_1, ..., x_n)] = \varphi_e^{(n)}(x_1, ..., x_n)$$

We write ψ_U for $\psi_U^{(1)}$.

By enumeration theorem, for each n, the universal function $\psi_U^{(n)}$ is computable.

1.3.5 Recursion Theorem

Let f be a total unary computable function. Then there is a number n such that $\varphi_{f(n)} = \varphi_n$

Proof:

• Consider the function ψ defined as follows:

$$\psi(x,y) = \begin{cases} \varphi_{\varphi_x(x)}(y) & \text{if } \varphi_x(x) \ \ s\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Clearly, the function ψ is computable, since the following algorithm computes it: run φ_x on input x; if it halts with output w, run φ_w on input y; if it halts output the result. Equivalently, if φ_U is the function corresponding to the Universal Turing Machine, then $\psi(x,y) := \varphi_U(\varphi_U(x,x),y)$.

• Now by the s-m-n theorem there is a total, computable primitive recursive function d such that

$$\varphi_{d(x)}(y) = \psi(x,y)$$

namely, d is defined by $d(x) = S_1^1(e,x)$ where e is the index of ψ.

• Since f and d are both total computable functions, $f \circ d$ is also a total computable function. let z be one of its index, i.e.

$$\varphi_z = f \circ d$$

Note that since φ_z is total, we have that in particular $\varphi_z(z) \ \downarrow$, so $\psi(z,y) = \varphi_{\varphi_z(z)}(y)$ for all y.

• For every y, we get

$$\varphi_{d(z)}(y) = \psi(z,y) = \varphi_{\varphi_z(z)}(y) = \varphi_{(f\circ d)(z)}(y)$$

and so

$$\varphi_{d(z)} = \varphi_{f(d(z))}$$

and hence $d(z)$ is the fixed point n that we were looking for.

Intuitive Consequences

Basically, the intuitive consequences of the recursive theorem is that, when defining a computable function φ_n, we may use the number n in the definition.

For example: there is an n such that $W_n = dom(\varphi_n) = \{n\}$.

• Define

$$\psi(x,y) = \begin{cases} x & \text{if } x = y \\ \uparrow & \text{if } x \neq y \end{cases}$$

By the s-m-n theorem there is a recursive f such that $\varphi_{f(x)}(y) = \psi(x,y)$. Let n be the fixed point of f. Then for all y

$$\varphi_n(y) = \varphi_{f(n)}(y) = \psi(n,y)$$

so $W_n = \{n\}$

• Define $\psi(x, y) = x$. Then by the s-m-n theorem there is a recursive f such that $\varphi_{f(x)}(y) = \psi(x, y)$. Let n be a fixed point of f. Then for all y we have $\varphi_n(y) = \varphi_{f(n)}(y) = \psi(n, y) = n$.

This situation has some interesting applications: since we can see n as representing the code of φ_n, the function φ_n can be said to "output its own code".

Part II

Statistical Machine Translation

Chapter 2

Statistical Machine Translation

Introduction

This chapter will be based on the fundamental statistical equation from a highly cited paper from 1993 by Peter F. Brown* from IBM T.J. Waston Research Center, "The mathematics of statistical machine translation: Parameter estimation"[1]. Then we will introduce one of the models used in statistical machine translation, which is the language model. The most common method for language modeling is the use of N-Gram language models. We use another 2 references as sources[2][3].

Contents

2.1 Fundamental statistical equation

According to these authors, the general idea behind statistical machine translation is the following:

There is a sentence we want to translate from French to English. Since we have a large number of parallel texts of English and French translations, we will use these to determine a statistical probability that a given English sentence corresponds to the French sentence, do this for many English sentences, and then pick the English sentence with the highest probability. The same principle applies to a word, a paragraph, or a whole text.

[1]Peter F. Brown, Stephen A. Delle Pietra, Vincent J. Bella Pietra, Robert L. Mercer, *The mathematics of statistical machine translation: Parameter estimate,* 1994

[2]`https://devopedia.org/n-gram-model`

[3]`https://towardsdatascience.com/perplexity-in-language-models-87a196019a94`

The probability for the English translation is determined using a commonly-used equation called Bayes Rule, which is $P(s|o) = P(o|s)P(s)/P(o)$, which means the probability of the state given the observation, $P(s|o)$, equals the probability of the observation given the state, $P(o|s)$, times the probability of the state happening in general, $P(s)$, divided by the probability of the observation happening in general, $P(o)$. In this case the state is the English translation and the observation is the original French sentence. Since $P(o)$, the probability of the French sentence, is the same for every English translation, and we are only connected with comparing the probabilities of different English translations, we need only consider $P(o|s)P(s)$.

Now if we replace the variables of the equation, s with f for a French sentence and o with e for an English sentence, we arrive at the fundamental equation of statistical machine translation:

$$\tilde{e} = \underset{e \in e^*}{\operatorname{argmax}} P(e|f) = \underset{e \in e^*}{\operatorname{argmax}} P(f|e)P(e)$$

This means that one English translation $\tilde{e}$ is the English sentence that maximizes the equation $p(f|e)p(e)$.

To calculate $\tilde{e}$ we must understand the meaning behind $P(f|e)$ and $P(e)$. $P(f|e)$ in terms of Bayes rule represents the likelihood: how likely is it that the French sentence would be a translation of (or would occur given) the English sentence. $P(e)$ represents the prior (our prior knowledge): how likely is it that the English sentence e would ever be used.

2.2 Language Model

Language models determine the probability of the next word by analyzing the text in data. These models interpret the data by feeding it through algorithms.

The algorithms are responsible for creating rules for the context in natural language. The models are prepared for the prediction of words by learning the features and characteristics of a language. With this learning, the model prepares itself for understanding phrases and predicting the next words in sentences.

For training a language model, a number of probabilistic approaches are used. These approaches vary on the basis of the purpose for which a language model is created. The amount of text data to be analyzed and the math applied for analysis makes a difference in the approach followed for creating and training a language model.

For example, a language model used for predicting the next word in a search query will be absolutely different from those used in predicting the next word in a long document (such as Google Docs). The approach followed to train the model would be unique in both cases.

2.2.1 N-Gram

N-gram is an algorithm based on statistical language model. Its basic idea is to perform a sliding window operation of size N on the content of the text according to bytes,

forming a sequence of byte fragments of length N. Each byte segment is called a gram. The frequency of occurrence of all grams is counted, and filtered according to a pre-set threshold to form a list of key grams, which is the vector feature space of this text. A gram is a feature vector dimension. The basic hypothesis is that the occurrence of the Nth word is only related to the previous $N-1$ words, and not related to any other words, and the probability of the entire sentence is the product of the probability of occurrence of each word. These probabilities can be obtained by directly counting the number of simultaneous occurrences of N words from the corpus.

N-Gram is the simplest model that assigns probabilities to sentences and sequences of words. An N-Gram is a sequence of n words: a 2-gram (which we will call bigram) is two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (trigram) is a three-word sequence of words like "please turn your", or "turn your homework". We will see how to use N-Gram to estimate the probability of the last word of an N-Gram given the previous words, and also to assign probabilities to entire sequences.

Let's start with computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "its water is so transparent that" and we want to know the probability that the next word is the:

$$P(the|its\ water\ is\ so\ transparent\ that) \qquad (2.1)$$

One way to estimate this probability is from the relative frequency counts: take a very large corpus, count the number of times we see "its water is so transparent that", and count the number of times this is followed by "the". This would be answering the question "Out the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = C(its\ water\ is\ so\ transparent\ that\ the)/C(its\ water\ is\ so\ tran$$

With a large enough corpus, we can compute these counts and estimate the probability from equation 2.2.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that the corpus isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple expressions of the example sentence may have counts of zero.

Similarly, if we wanted to know the joint probability of an entire sequence of words like "its water is so transparent", we could do it by asking "out of all possible sequences of five words, how many of them are its water is so transparent". We would have to get the count of its water is so transparent and divide by the sum of the counts of all possible five word sequences. That is a lot of work. For this reason, Brown will use more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W.

In the following, we will use $P(the)$ to represent the probability of a particular random variable X_i taking on the value "the". We'll represent a sequence of N words either as $w_1...w_n$ or $w_{1:n}$ (so the expression $w_{1:n-1}$ means the string $w_1, w_2, ..., w_{n-1}$). For the

joint probability of each word in a sequence having a particular value $P(X_1 = w_1, X_2 = w_2, ..., X_n = w_n)$, we will use $P(w_1, w_2, ..., w_n)$.

We can compute the probabilities of entire sequences like $P(w_1, w_2, ..., w_n)$ by decomposing the probability through the chain rule of probability:

$$\begin{aligned} P(X_1...X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_{1:2})\dots P(X_n|X_{1:n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \end{aligned} \tag{2.3}$$

Applying the chain rule to words, we get

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2})...P(w_n|w_{1:n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \end{aligned} \tag{2.4}$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 2.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule, we still don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$.

The intuition of the N-Gram approach is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

The bigram , for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(the|its\ water\ is\ so\ transparent\ that) \tag{2.5}$$

we approximate it with the probability

$$P(the|that) \tag{2.6}$$

When we use a bigram to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \tag{2.7}$$

The assumption that the probability of a word depends only on the previous word is called a Markov assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the N-Gram (which looks $n-1$ words into the past).

We'll use N here to mean the N-Gram size, so $N = 2$ means bigrams and $N = 3$ means trigrams. Then the general equation for this N-Gram approximation to the conditional probability of the next word in a sequence is as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \tag{2.8}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting equation 2.7 into equation 2.4:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \qquad (2.9)$$

An intuitive way to estimate these bigram or N-Gram probabilities is called maximum likelihood estimation (MLE). We get the MLE estimate for the parameters of an N-Gram by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \qquad (2.10)$$

We can simplify equation 2.10 since the sum of all bigram counts that start with a given word must be equal to the unigram count for that word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \qquad (2.11)$$

We will work through an example using a mini-corpus of three sentences. First we need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. Also, we need a special end-symbol </s>.

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

Here are the calculations for some of the bigram probabilities from this corpus

$$P(I|<s>) = \frac{2}{3} = 0.67 \qquad P(Sam|<s>) = \frac{1}{3} = 0.33 \qquad P(am|I) = \frac{2}{3} = 0.67$$
$$P(</s>|Sam) = \frac{1}{2} = 0.5 \qquad P(Sam|am) = \frac{1}{2} = 0.5 \qquad P(do|I) = \frac{1}{3} = 0.33$$

For the general case of MLE N-Gram parameter estimation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}\ w_n)}{C(w_{n-N+1:n-1})} \qquad (2.12)$$

Equation 2.12 estimates the N-Gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. The ratio is called a relative frequency. The use of relative frequencies as a way to estimate probabilities is an example of MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$).

2.2.2 Perplexity in Language Models

Perplexity (sometimes called PP for short) is an evaluation measure for language models. The perplexity of a language model on a test set is the inverse probability of the test set, normalized by the number of words.For a test set $W = w_1 w_2 ... w_N$, :

$$\begin{aligned} PP(W) &= P(w_1 w_2 ... w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} \end{aligned}$$

2.2.2.1 Probability of the test set

First of all, we want our model to assign high probabilities to sentences that are real and syntactically correct, and low probabilities to fake, incorrect, or highly infrequent sentences. Assuming our dataset is made of sentences that are in fact real and correct, this means that one model will be the one that assigns the highest probability to the test set. Intuitively, if a model assigns a high probability to the test set, it means that it is not surprising to see it (it's not perplexed by it), which means that it has a good understanding of how the language works.

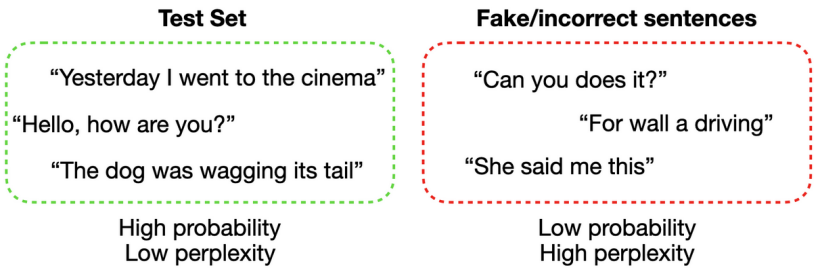

Figure 2.1: Low perplexity VS High perplexity

2.2.2.2 Normalising

However, it's worth nothing that datasets can have varying numbers of sentences, and sentences can have varying numbers of words. Adding more sentences introduces more uncertainty, so other things being equal a larger test set is likely to have a lower probability than a smaller one. Ideally, we'd like to have a metric that is independent of the size of the dataset. We could obtain this by normalising the probabilities of the test set by the total number of words, which would give us a per-word measure.

For example, let's take a unigram model:

$$P(W) = P(w_1, w_2, ...w_N) = P(w_1)P(w_2)...P(w_N) = \prod_{i=1}^{N} P(w_i)$$

To normalize this probability which is given by a product, we can take the log probability, which turns the product into a sum:

$$ln(P(W)) = ln(\prod_{i=1}^{N} P(w_i)) = \sum_{i=1}^{N} lnP(w_i)$$

Now we can normalize this by dividing by N to obtain the per-word log probability:

$$\frac{ln(P(W))}{N} = \frac{\sum_{i+1}^{N} lnP(w_i)}{N}$$

Then remove the log by exponentiating:

$$\begin{aligned} e^{\frac{ln(P(W))}{N}} &= e^{\frac{\sum_{i=1}^{N} lnP(w_i)}{N}} \\ (e^{ln(P(W))})^{\frac{1}{N}} &= (e^{\sum_{i=1}^{N} lnP(w_i)})^{\frac{1}{N}} \\ P(W)^{\frac{1}{N}} &= (\prod_{i=1}^{N} P(w_i))^{\frac{1}{N}} \end{aligned}$$

Normalization is obtained by taking the N-th root.

2.2.2.3 Bringing it all together

Now going back to our original equation for perplexity, we can see that we can interpret it as the inverse probability of the test set, normalized by the number of words in the test set:

$$\begin{aligned} PP(W) &= P(w_1 w_2 ... w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} \end{aligned}$$

• Since we are taking the inverse probability, a lower perplexity indicates a better model.

• In this case W is the test set. It contains the sequence of words of all sentences one after the other, including the start-of-sentence and end-of-sentence tokens, <SOS> and <EOS>.

For example, a test set with two sentences would look like this: W = (<SOS>, This, is, the, first, sentence, ., <EOS>, <SOS>, This, is, the, second, one, ., <EOS>)

N is the count of all tokens in our test set, including SOS/EOS and punctuation. In this example, N = 16.

2.2.3 Advantages and disadvantages of the N-gram approach

The advantage of the N-gram model is that it contains all the information that the first N-1 words can provide. These words have a strong binding force on the appearance of the current word, but its disadvantage is that it requires a considerable amount of training text to determine the parameters of the model. When N is large, the parameter

space of the model is too large. So the common value of N is generally 1,2. There is also a data smoothing problem caused by data sparseness. The main solution is to make the sum of all N-gram probabilities 1 and make some ε.

In addition, compared with the linguistic rule model of word representation in continuous space (such as the word vector constructed by word2vec), the N-gram language model has the following limitations:

The N-gram model is constructed based on discrete unit words that do not have any genetic attributes between each other, and thus does not have the semantic advantage satisfied by word vectors in continuous space: words with similar meanings have similar word vectors, and thus become a system When the model adjusts parameters for a word or word sequence, words and word sequences with similar meanings will also change.

Therefore, if the keyword weight is known to be very large, the N-gram model may be more appropriate.

Part III

Neural Machine Translation

Chapter 3

Long Short-Term Memory (LSTM) and Sequence to Sequence model

Introduction

The chapter will present the basic concepts of feed-forward neural networks and recurrent neural networks. The presentation is adopted from 4 references[①] [②] [③] [④].

Long short-term memory (LSTM) networks are an extension for recurrent neural networks, which basically extends the memory. With regard to this, the working principle of LSTM will be clearly explained. Followed by LSTM, Sequence-to-Sequence model is about to be introduced. A typical Seq2Seq model consists of an encoder and decoder which are themselves two separate neural networks combined into giant network, Both encoder and decoder are typically LSTM or GRU models. In this chapter, we will only use LSTM models.

Contents

[①] https://builtin.com/data-science/recurrent-neural-networks-and-lstm

[②] https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[③] https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b

[④] https://medium.com/analytics-vidhya/seq2seq-models-french-to-english-translation-using-encoder-de

3.1 Feed-Forward Neural Networks and Recurrent Neural Networks

Recurrent neural networks are a class of neural networks that are helpful in modelling data sequences. Derived from feed-forward networks, RNNs try to mimic some characteristics of human brain functioning. RNNs produce predictive results in data sequence that other algorithms can't. RNNs are now heavily used in Machine Translation.

3.1.1 Feed-Forward Neural Networks and connections

A neural network simply consists of nodes. These nodes are connected in some way. Then each node holds a number, and each connection holds a weight.

These nodes are split between the input, hidden and output layer. In practice, there are many layers and there is no general best number of layers.

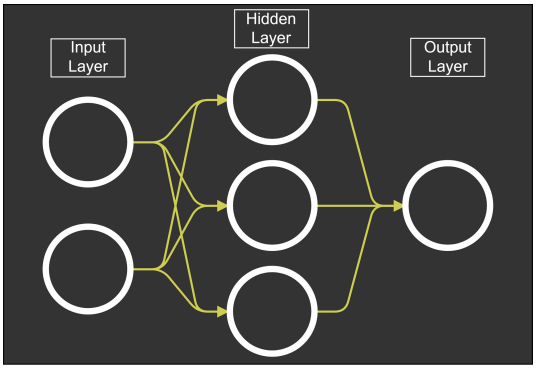

Figure 3.1: Feedforward Neural Network example

In Fig 3.1, white circles corresponding to nodes, and yellow arrows are the connections (each having a weight) from one node to another node.

The idea is that we input data into the input layer, which sends the numbers from our data ping-ponging forward, through the different connections, from one node to another in the network. Once we reach the output layer, we hopefully have the number we wished for.

The input data is just our dataset, where each observation is run through sequentially from $x = 1, \ldots, x = i$. Each node has some activation - a value between 0 and 1, where 1 is the maximum activation and 0 is the minimum activation a node can have.

3.1.1.1 From input layer to hidden layer

Each node has an activation a and each arc that connects to a new node has a weight w. Then we can multiply activations by weights and pass it to a single node in the next

layer, from the first weights and activations $w_1 a_1$ all the way to $w_n a_n$:

$$w_1 a_1 + w_2 a_2 + w_n a_n = \text{ activation of a new node}$$

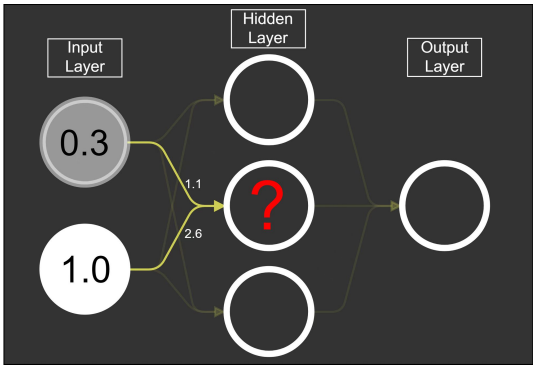

Figure 3.2: Input layer to hidden layer

That is, multiply n number of weights and activations, to get the activation of a new node.

$$1.1 \times 0.3 + 2.6 \times 1.0 = 2.93$$

the procedure is the same moving forward in the network of nodes, hence the name feedforward neural network.

3.1.1.2 Activation Functions

We also have an activation function, most commonly a sigmoid function, which just scales the output to be between 0 and 1.

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

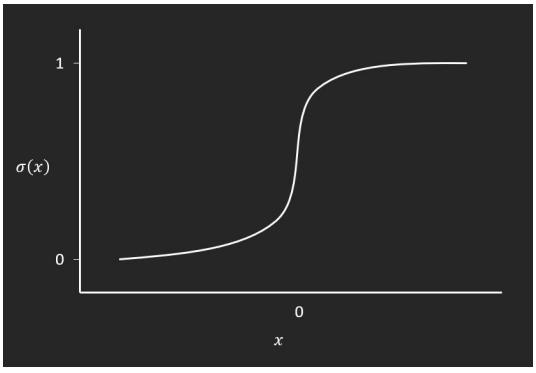

Figure 3.3: Sigmoid function

We wrap the equation for new nodes with the activation:

$$\sigma(w_1 a_1 + w_2 a_2 + ... + w_n a_n) = \text{ activation of a new node}$$

3.1.1.3 From hidden layer to output layer

The activation function is only used in the hidden layer. The output node is simply the sum of the hidden layer outputs times the weights between hidden layer and the output layer.

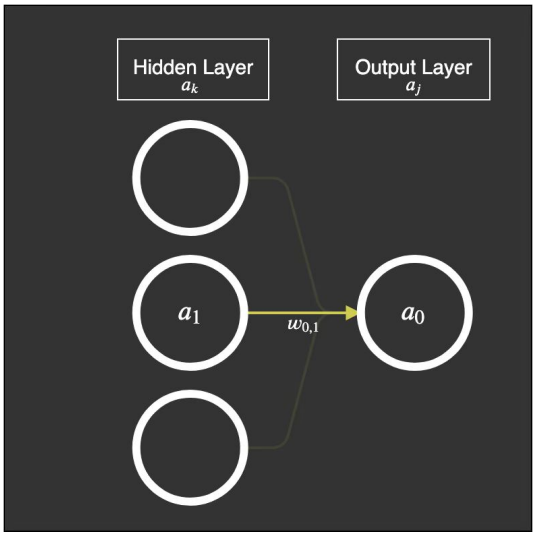

Figure 3.4: Hidden layer to output layer

3.1.2 Recurrent Neural Networks

In a RNN the information cycles also through loops. When it makes a decision, it considers the current input and also what it has learned from the input.

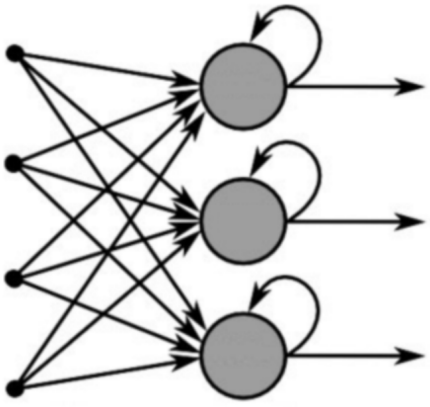

Figure 3.5: Recurrent Neural Networks

In the above diagram, the input layer X_t processes the initial input and passes it to the middle layer A. The middle layer consists of multiple hidden layers, each with its activation functions, weights and biases. These parameters are standardized across the

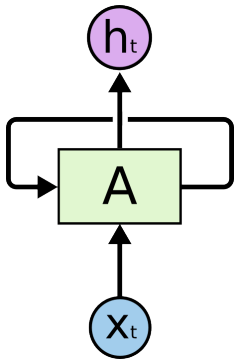

Figure 3.6: a loop of RNN

hidden layer so that instead of creating multiple hidden layers, it will create one and loop it over. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

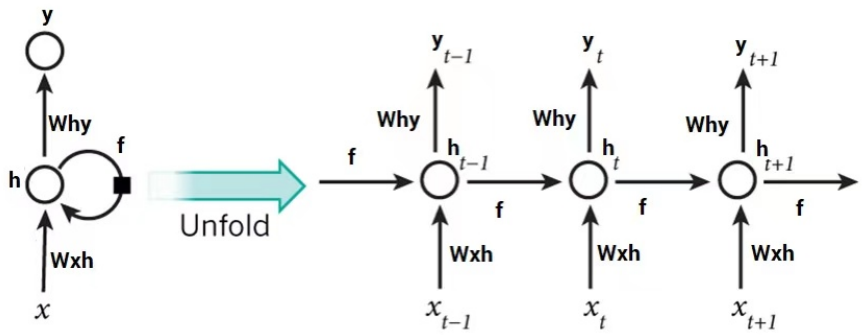

Figure 3.7: An unrolled RNN

x_t	input data at current timestamp
y_t	output
Wxh	weights for transforming x_t to RNN hidden state (not prediction)
Why	weights for transforming RNN hidden state to prediction
h_t	hidden state
$circle$	RNN cell

3.2 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are an extension of RNN that extend the memory, capable of learning long-term dependencies. LSTMs are used as the building blocks for the layers of a RNN. LSTM assigns data "weights" which helps RNNs to either

let new information in, forget information or give it importance enough to impact the output.

LSTMs enable RNNs to remember inputs over a long period of time. This is because LSTMs contain information in a memory, like the memory of a computer. The LSTM can read, write and delete information from its memory.

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.

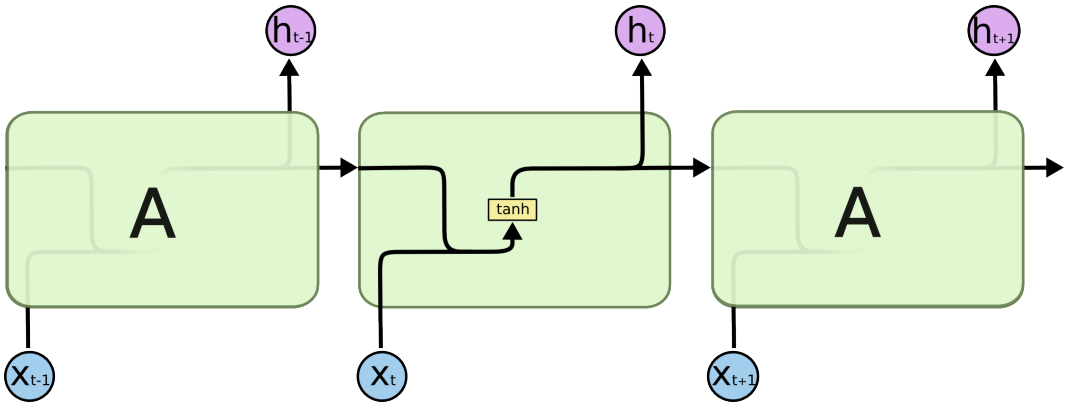

Figure 3.8: Repeating module in a standard RNN

Instead, the repeating module of LSTMs has four layers interacting in a very special way.

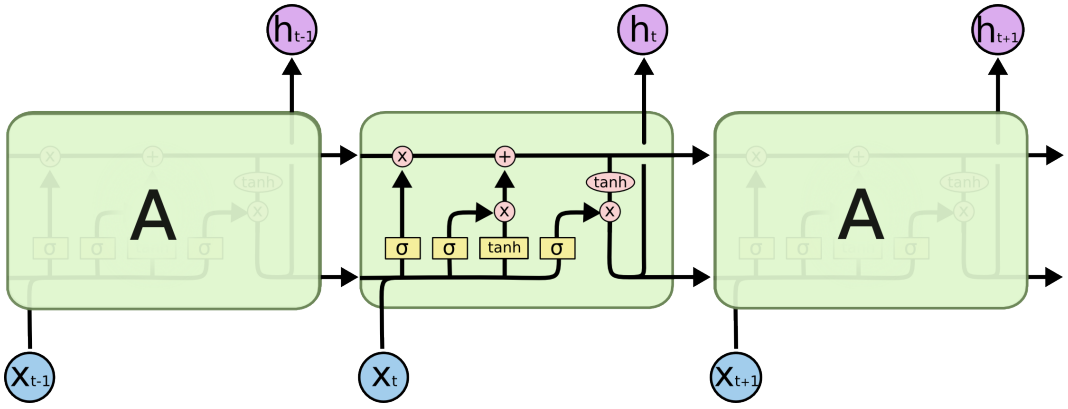

Figure 3.9: Repeating module in an LSTM

3.2.1 LSTM architecture

The key to LSTMs is the cell. Every unit of the LSTM network is known as a "cell". Each cell is composed of 3 inputs (x_t, h_{t-1}, C_{t-1}) and 2 outputs (h_t, C_t).

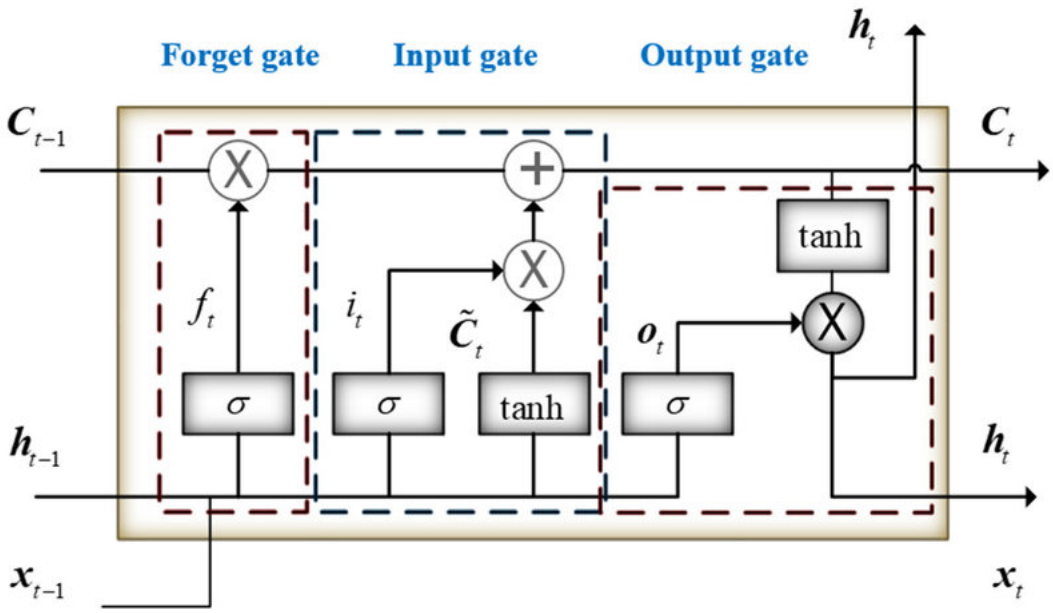

Figure 3.10: a simple LSTM black box

Inputs	x_t	taken at timestamp t
	h_{t-1}	previous hidden state
	C_{t-1}	previous cell state
Outputs	h_t	updated hidden state
	C_t	current cell state

The entire rectangle in Fig 3.10 is called an LSTM "cell". It is analogous to the circle from the previous RNN diagram. These are the parts that make up the LSTM cell:

1. The "Cell State"
2. The "Hidden State"
3. The Gates: "Forget", "Input", and "Output"

LSTM uses a special mechanism of controlling the memorizing process, popularly referred to as *gate mechanism*. The goals of gates in LSTM are to store the memory components in analog format and produce a probabilistic score by doing point-wise multiplication using sigmoid activation function, which outputs a number between 0 and 1. The gates in LSTM regulate the flow of information in and out of the LSTM cells. There are 3 types of gates.

- An input gate determines what new information should be added to the networks long-term memory, given the previous hidden state and new input data.
- An output gate updates and finalizes the next hidden state.
- A forget gate eliminates unnecessary information.

3.2.2 "Cell State" vs "Hidden State"

Hidden State - Conceptual Interpretation

The characterization of a timestamp data can mean different things. For example, we are processing the phrase "the sky is blur, therefore the babe elephant is crying". If we want the LSTM network to be able to classify the sentiment of a word in the context of the sentence, the hidden state at t=3 would be an encoded version of "is", which we would then further process to obtain the predicted sentiment. If we want the LSTM network to be able to predict the next word based on the current series of words, the hidden state at t=3 would be an encoded version of the prediction for the next word (ideally, "blue"), which we would again process outside of the LSTM to get the predicted word. As seen, characterization is an abstract term that merely serves to illustrate how the hidden state is more concerned with the most recent time-step. Also, we need to note that hidden state does not equal the output or prediction, it is merely an encoding of the most recent time-step. **Cell State - Conceptual Interpretation**

The cell state is more concerned with the entire data. If we are processing the word "elephant", the cell state contains information of all words right from the start of the phrase. As we can see in the Fig 3.10, each time a time-step pf data passes through an LSTM cell, a copy of the time-step data is filtered through a forget gate, and another copy through the input gate; the results of both gates are incorporated into the cell state from processing the previous time-step and gets passed on to get modified by the next time-step yet again. The weights in the forget gate and input gate figure out how to extract features from such information so as to determine which time-steps are important (high forget weights), which are not (low forget weights), and how to encode information from the current time-step into the cell state (input weights). As a result, not all time-steps are incorporated equally into the cell state – some are more significant that others. To summarize, the cell state is basically the global or aggregate memory of the LSTM network over all time-steps.

3.2.3 Core idea behind LSTMs

The first step in LSTM is to decide what information will be thrown away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer". It looks at h_{t-1} and x_t and outputs a number in the range of 0-1 for each number in the cell state C_{t-1}. A value of 0 means "let nothing through" while a value of 1 means "let everything through".

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

The second step is to decide what new information is going to be stored in the cell state. It is divided into two parts. Firstly, a sigmoid layer called the "input gate layer" decides which values will be updated. Secondly, a tanh layer creates a vector of new values, $\tilde{C}_t$, that could be added to the cell state. In the next step, these two wiil be combined to create an update to the cell state.

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_C \cdot [h_{t-1,x_t}] + b_C) \end{aligned}$$

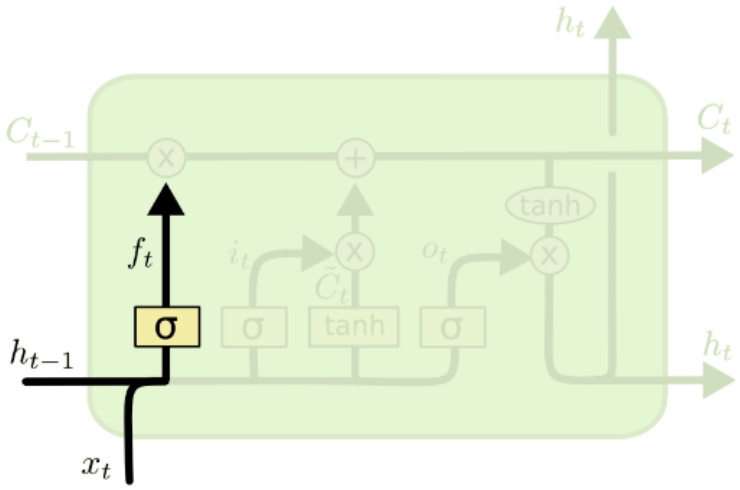

Figure 3.11: Forget Gate

Now is the time to transform the old cell state C_{t-1} into the new cell state, C_t. The old state will be multiplied by f_t, forgetting the irrelevant information. The input gate remembers relevant information and adds it to the current cell state with tanh activation.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

The final step is to decide the output. The output will be based on the cell state after filter. A sigmoid layer will be run to decide to output what parts of the cell state. Then put the cell state through tanh and multiply it by the output of the sigmoid layer so that only the decided parts will be outputted.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t], b_o)$$
$$h_t = o_t * tanh(C_t)$$

3.3 Sequence to sequence model

Sequence to sequence model is a model that tries to map input text with fixed length to output text fixed length where the length of input and output to the model may differ. This allows us to use this model in machine translation.

Sequence to sequence model is divided in two phases, which are training phase and inference phase respectively.

3.3.1 Training phase

Training phase is a process in 2 parts encoder and decoder. After setting up the encoder and decoder models, the models will be trained and every timestamp will be predicted by reading input word by word or char by char. Before going through training and testing,

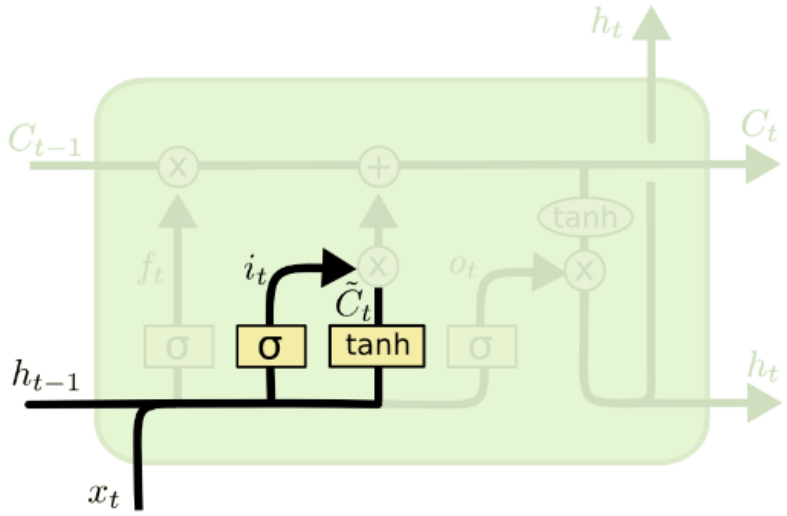

Figure 3.12: Input Gate

data needs to be cleaned and tokens need to be added to specify the start and end of sentences so that the model will understand when to end.

3.3.1.1 Encoder

The encoder is an LSTM network. At each timestamp word is read or processed and it captures the contextual information at every timestamp from the input sequences passed to the encoder model.

• Xi: since we are going to use word-level encoding in our machine translation system, input at each timestamp will be each word in the sentence, which means in the example of Fig 17 X1 = 'Je', X2 = 'ne'. Each word is represented in the form of a vector. For this, each word is replaced by word index of that word in some corpus. Most frequent words have smaller word index than the less frequent words.

• h0 and c0: initial hidden state and context vectors which are all zeros (generally) and fed at the 0th timestamp to the encoder.

• hi and ci: hidden state and context vectors after timestamp i. These vectors in simple terms represent what the encoder has seen until this timestamp. For example, h3 and c3 will remember that the network has seen "Je ne parle". The size of each of these vectors is equal to the number of units of LSTM. The state obtained after the last timestamp is fed into decoder as decoder initial states.

• Yi: output at timestamp i. It is the probability distribution over the entire vocabulary which is generated by using the Softmax activation function.

3.3.1.2 Decoder

The decoder is also an LSTM network that reads the entire target sequence or sentence word-by-word and predicts the same sequence offset by one timestamp. However, unlike

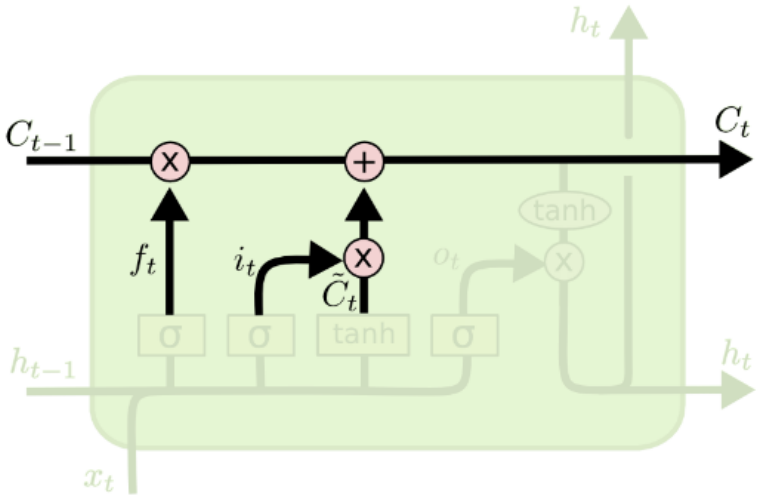

Figure 3.13: Figure 14: Cell state update mechanism

encoder, decoder behaves differently in training and inference phase. Two special tokens need to be added to the output sentence. These tokens are "<start>" (at the beginning of the string) and "<end>" (at the end of the string). The model will be trained to predict the outputs by using backpropagation with time and appropriate loss function.

3.3.2 Inference phase

Machine learning model inference is the process of deploying a machine learning model to a production environment to infer a result from input data. At this point, the model will be processing new and unseen input data. When a model performs inference, it is producing a result based on the trained algorithm. This means model inference is within the deployment phase of a machine learning lifecycle. The results that are inferred are usually observed and continuously monitored, at which point the model can be retrained or optimised as a separate phase of a model lifecycle.

After training the encoder-decoder model, the model will be tested on new unseen input sequences for which the target sequence is unknown. Here are the steps to process for decoding the test sequence:

1. Encode the entire input sequence and initialize the decoder with the internal states of the encoder.

2. Pass <start>as an input to the decoder.

3. Run the decoder for one timestamp with the internal states.

4. The output will be the probability for the next word. The word with the maximum probability will be selected.

5. Pass the maximum probability word as an input to the decoder in the next timestamp and update the internal states with the current timestamp.

6. Repeat steps 3-5 until <end>is generated or the maximum length of the target sequence is reached.

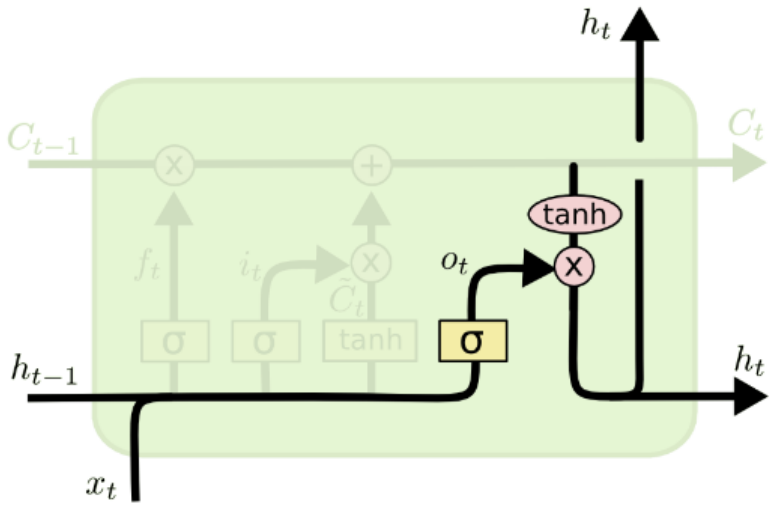

Figure 3.14: Output Gate

Figure 3.15: Machine Translation Model

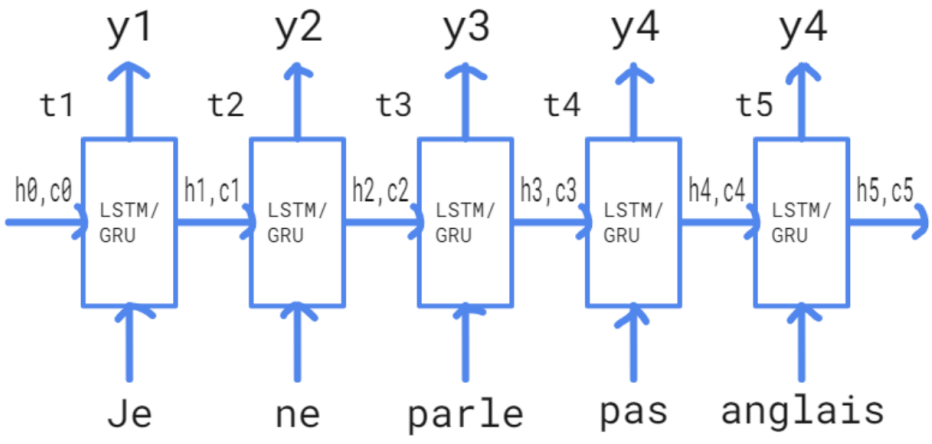

Figure 3.16: Encoder architecture

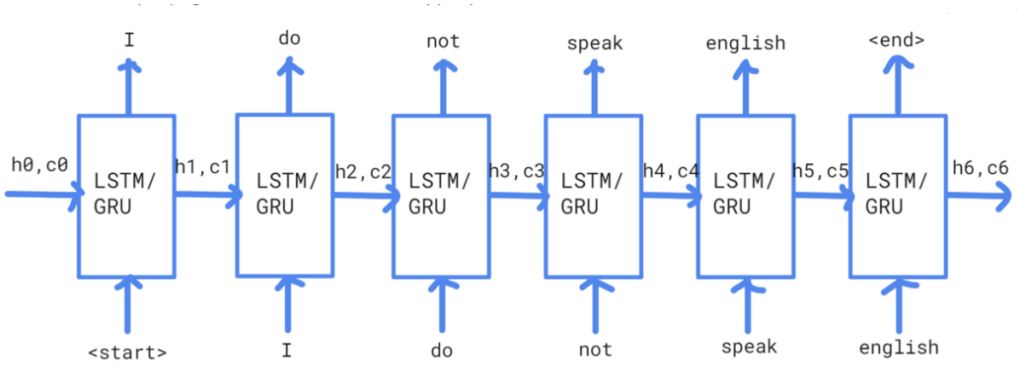

Figure 3.17: Decoder architecture

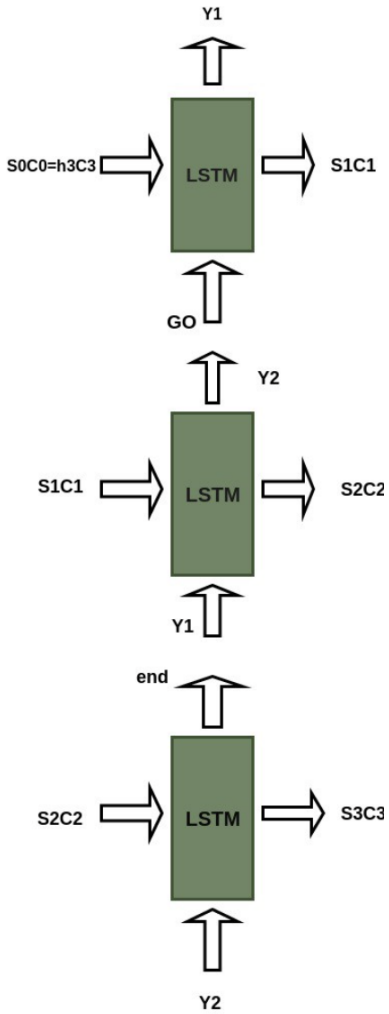

Figure 3.18: Inference Phase

Chapter 4

Experiment

Introduction

We have learned the method of Seq2Seq model with LSTM networks in chapter 3 and perplexity in chapter 2. In this chapter, we will realize Seq2Seq model in Pytorch and perplexity as the evaluation. One of disadvantage of Seq2Seq model is that previous memory would be lost to a large extent. The Seq2Seq model is going to be 0improved by using one technique introduced in an article, Sequence to Sequence Learning with Neural Networks, which is source sequence reversal. We are going to do a comparison in the value of perplexity to prove source sequence reversal can take effect. The model realization will be divided into three parts, which are data preparation, model establishment and model training.

Contents

4.1 Improvement of Seq2Seq model

As we talked in previous chapter, we use LSTM networks in Seq2Seq model. However, we can find out that after the filter of three gates in LSTM networks the biggest loss would be previous memory and the current memory is biased towards the latest memory, which is also a deficiency of LSTM networks. Once the sequence is too long, the long-range dependency will be weak.

Based on LSTM networks, we know that in Fig 17 the context vector C saves the most information at the source end at time 4, and the information at time 1 also saves

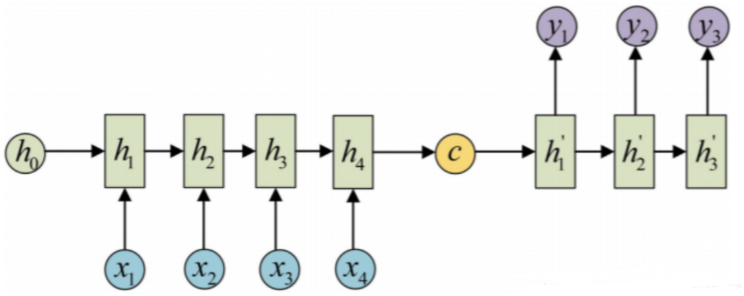

Figure 4.1: Seq2Seq model

very little. However when translating, the first translated word is likely to depend on the input at the source time 1, but at this time C contains very little information. If the first word is not translated well, as the input of its time step (the output of each time step of the target is the input of the next time step), then the subsequent quality is also greatly reduced. In response to this problem, academia has proposed various improvements. The easiest technique was introduced in the article, Sequence to Sequence Learning with Neural Networks[①], which is source sequence reversal. An example of source sequence reversal will be that the input of "I love you" becomes "you love I". In this way, the most saved context vector C must be I, and it will be more accurate when the decoder translates "I" for the first time.

4.2 Realization of Seq2Seq model in Pytorch

4.2.1 Data Preparation

The data set uses the Multi30k data set that comes with pytorchtext, and selects French and English translations since word segmentation is easier.

```
Two men are at the stove preparing food.
A man in green holds a guitar while the other man observes his shirt.
A man is smiling at a stuffed lion
A trendy girl talking on her cellphone while gliding slowly down the
street.
A woman with a large purse is walking by a gate.
Boys dancing on poles in the middle of the night.
A ballet class of five girls jumping in sequence.
Four guys three wearing hats one not are jumping at the top of a
staircase.
A black dog and a spotted dog are fighting
```

Figure 4.2: English corpus

The general process of processing data in pytorchtext is to define a field, process the data of the dataset through field, create a vocabulary list for field, and create a data iterator.

[①]Ilya Sutskever, Oriol Vinyals, Quoc V. Le, *Sequence to Sequence Learning with Neural Networks,* 2014

```
Deux hommes aux fourneaux pr√©parent √† manger.
Un homme en vert tient une guitare tandis qu'un autre homme observe sa
chemise.
Un homme sourit √† un ours en peluche.
Une fille branch√©e parle √† son portable tout en glissant lentement dans
la rue.
Une femme avec un gros sac passe par une porte.
Des gar√ßons dansent sur des barres au milieu de la nuit.
Une classe de ballet, compos√©e de cinq filles, sautent en cadence.
Quatre gars, dont trois portent des chapeaux, sautent du haut d'un
escalier.
Un chien noir et un chien √† t√¢ches se battent.
```

Figure 4.3: French corpus

1. Define field

```python
# define field
# create a tokenizer
spacy_en=spacy.load("en_core_web_sm")# English tokenizer
spacy_de=spacy.load("de_core_news_sm")# French tokenizer

def en_seq(text):
    return [word.text for word in spacy_en.tokenizer(text)]


# source sequence reversal
def de_seq(text):
    return [word.text for word in spacy_de.tokenizer(text)][::-1]

# process source sequence
SRC=Field(tokenize=de_seq,
          init_token="<sos>",
          eos_token="<eos>",
          lower=True)

# process target sequence
TRG=Field(tokenize=en_seq,
          init_token="<sos>",
          eos_token="<eos>",
          lower=True)
```

Figure 4.4: field

2. Create vocabulary list and require every word appears at leats twice.

3. After the vocabulary list is established, we establish an iterative index. If batch size=1, it does not matter if the text lengths are not equal. But when batch size>1, there is a problem. The same batch of samples are not the same length. Generally, we will pad the sentence to fill it up. Fortunately, the torchtext iterator automatically helps pad filling, and the BucketIterator iterator selects the most appropriate length as the fixed length of all sentences. Pad it when lower than the word length, and cut it when higher than this length. The fixed length is based on the most suitable length of all samples in the data set.

We must need the position of padding since when calculating the loss in loss function padding part doesn't participate in calculation.

```
{'src': ['un', 'homme', 'en', 'vert', 'tient', 'une', 'guitare', 'tandis', "qu'un", 'autre', 'homme', 'observe',
'sa', 'chemise', '.'], 'trg': ['a', 'man', 'in', 'green', 'holds', 'a', 'guitar', 'while', 'the', 'other', 'man',
 'observes', 'his', 'shirt', '.']}
{'src': ['un', 'homme', 'sourit', 'à', 'un', 'ours', 'en', 'peluche', '.'], 'trg': ['a', 'man', 'is', 'smiling',
'at', 'a', 'stuffed', 'lion']}
{'src': ['une', 'fille', 'branchée', 'parle', 'à', 'son', 'portable', 'tout', 'en', 'glissant', 'lentement', 'dan
s', 'la', 'rue', '.'], 'trg': ['a', 'trendy', 'girl', 'talking', 'on', 'her', 'cellphone', 'while', 'gliding', 's
lowly', 'down', 'the', 'street', '.']}
{'src': ['une', 'femme', 'avec', 'un', 'gros', 'sac', 'passe', 'par', 'une', 'porte', '.'], 'trg': ['a', 'woman',
 'with', 'a', 'large', 'purse', 'is', 'walking', 'by', 'a', 'gate', '.']}
```

Figure 4.5: processed corpus

4.2.2 Model Establishment

Seq2Seq model is mainly composed of two parts, namely the encoder and the decoder. We have 3 modules here, which are encoder, decoder, and Seq2Seq, which integrates the two parts. The network depth of the source and the target are both 4 layers.

4.2.3 Model Training

CrossEntropyLoss is used as loss function. We initialized parameter, established train and evaluate functions. The loss function is simply expressed as: perplexity. In chapter 2, we learned that the lower the perplexity is, the better the model is.

4.2.4 Results

Since we are using CPU here. The convergence speed is low.

As we can see from the results, with using source sequence reversal, it can reach not only less time consumption but also lower perplexity.

```
Epoch: 01 | Time: 32m 16s
        Train Loss: 5.131 | Train PPL: 169.253
         Val. Loss: 4.729 |  Val. PPL: 113.182
Epoch: 02 | Time: 31m 48s
        Train Loss: 4.674 | Train PPL: 107.161
         Val. Loss: 4.047 |  Val. PPL:  57.199
Epoch: 03 | Time: 31m 29s
        Train Loss: 4.238 | Train PPL:  69.287
         Val. Loss: 3.645 |  Val. PPL:  38.285
Epoch: 04 | Time: 31m 44s
        Train Loss: 3.959 | Train PPL:  52.387
         Val. Loss: 3.426 |  Val. PPL:  30.753
Epoch: 05 | Time: 30m 22s
        Train Loss: 3.757 | Train PPL:  42.811
         Val. Loss: 3.228 |  Val. PPL:  25.240
Epoch: 06 | Time: 31m 6s
        Train Loss: 3.577 | Train PPL:  35.757
         Val. Loss: 3.101 |  Val. PPL:  22.214
Epoch: 07 | Time: 39m 22s
        Train Loss: 3.402 | Train PPL:  30.021
         Val. Loss: 2.964 |  Val. PPL:  19.376
  [Bookmarked 29 Aug 2022 at 18:08:20]
Epoch: 08 | Time: 31m 25s
        Train Loss: 3.261 | Train PPL:  26.064
         Val. Loss: 2.873 |  Val. PPL:  17.685
Epoch: 09 | Time: 955m 38s
        Train Loss: 3.094 | Train PPL:  22.068
         Val. Loss: 2.765 |  Val. PPL:  15.878
Epoch: 10 | Time: 31m 3s
        Train Loss: 2.990 | Train PPL:  19.895
         Val. Loss: 2.687 |  Val. PPL:  14.685
| Test Loss: 2.666 | Test PPL:  14.385 |
```

Figure 4.6: result with sequence reversal

```
Epoch: 01 | Time: 105m 36s
        Train Loss: 5.130 | Train PPL: 169.048
         Val. Loss: 4.867 |  Val. PPL: 129.908
Epoch: 02 | Time: 60m 44s
        Train Loss: 4.859 | Train PPL: 128.935
         Val. Loss: 4.262 |  Val. PPL:  70.970
Epoch: 03 | Time: 1576m 9s
        Train Loss: 4.510 | Train PPL:  90.910
         Val. Loss: 3.927 |  Val. PPL:  50.743
Epoch: 04 | Time: 32m 15s
        Train Loss: 4.326 | Train PPL:  75.620
         Val. Loss: 3.727 |  Val. PPL:  41.564
Epoch: 05 | Time: 38m 44s
        Train Loss: 4.150 | Train PPL:  63.459
         Val. Loss: 3.566 |  Val. PPL:  35.382
Epoch: 06 | Time: 36m 34s
        Train Loss: 3.973 | Train PPL:  53.153
         Val. Loss: 3.425 |  Val. PPL:  30.724
Epoch: 07 | Time: 1458m 36s
        Train Loss: 3.835 | Train PPL:  46.306
         Val. Loss: 3.334 |  Val. PPL:  28.058
Epoch: 08 | Time: 76m 33s
        Train Loss: 3.713 | Train PPL:  40.962
         Val. Loss: 3.202 |  Val. PPL:  24.577
Epoch: 09 | Time: 157m 37s
        Train Loss: 3.624 | Train PPL:  37.478
         Val. Loss: 3.203 |  Val. PPL:  24.599
Epoch: 10 | Time: 41m 6s
        Train Loss: 3.546 | Train PPL:  34.688
         Val. Loss: 3.097 |  Val. PPL:  22.135
| Test Loss: 3.082 | Test PPL:  21.808 |
```

Figure 4.7: result without source sequence reversal

Conclusions and perspectives

A Turing machine is a machine capable of enumerating some arbitrary subset of valid strings of an alphabet. These strings are part of a recursively enumerable set. A Turing machine has a tape of infinite length on which it can perform read and write operations. The essential point of recursion theory is to study this notion of computability, e.f. which functions are computable and how computable they are.

Language models measure the fluency of the output and are an essential part of statistical machine translation. Mathematically, they assign each sentence a probability that indicates how likely that sentence is to occur in a text.N-Gram language models use the Markov assumption to break the probability of a sentence into the product of the probability of each word, given the history of preceding words. Language models are optimized on perplexity.

Neural machine translation models are often based on the seq2seq architecture. The seq2seq architecture is an encoder-decoder architecture which consists of two LSTM networks: the encoder LSTM and the decoder LSTM. The input to the encoder LSTM is the sentence in the original language; the input to the decoder LSTM is the sentence in the translated language with a start-of-sentence token. The output is the actual target sentence with an end-of-sentence token.

Even though N-Gram is easy to train the parameter because of maximum likelihood estimation and includes all the information of previous $N-1$ words. It still has some disadvantages. For example, it can lead to out of vocabulary problem (zero probability problem) due to data sparse. in academia, some methods appeared to deal with this problem, such as sub-word N-gram, which deserves to dig deeper in the future. On the other hand, there are also some different ways to improve Seq2Seq model in NMT, such as introducing attention mechanism which was born to help memorize long source sentences in neural machine translation.

Following is the final progress of the internship:

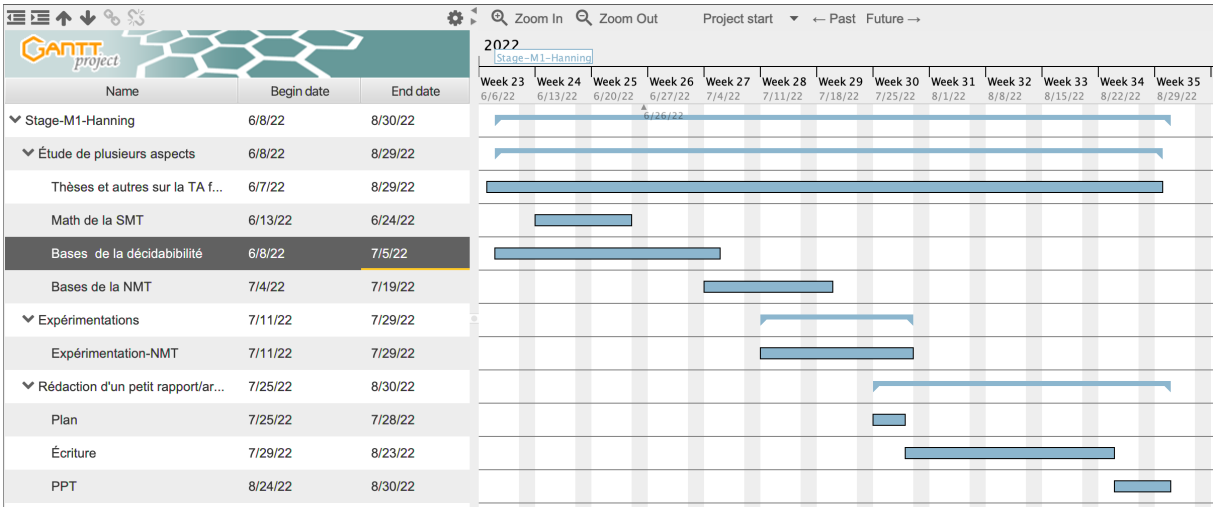

Figure 4.8: Final Gantt

Bibliography

[1] Lingxiao WANG, *Outils et environnements pour l'amélioration incrémentale, la post-édition contributive et l'évaluation continue de systèmes de TA. Application à la TA français-chinois,* Université de Grenoble, 2015.

[2] Ying ZHANG, *Modèles et outils pour des bases lexicales "métier" multilingues et contributives de grande taille, utilisables tant en traduction automatique et automatisé que pour des services dictionnairiques variés,* Université de Grenoble, 2016.

[3] Davis, Martin, *Computability and unsolvability,* Courier Corporation, 1958.

[4] Boitet, Christian and Blanchon, Hervé and Seligman, Mark and Bellynck, Valérie, *MT on and for the Web,* IEEE, 2010.

[5] Deutsch, Daniel and Dror, Rotem and Roth, Dan, *Re-Examining System-Level Correlations of Automatic Summarization Evaluation Metrics,* arXiv preprint arXiv:2204.10216, 2022.

[6] Peter F. Brown, Stephen A. Delle Pietra, Vincent J. Bella Pietra, Robert L. Mercer, *The mathematics of statistical machine translation: Parameter estimate,* Using Large Corpora, 1994.

[7] Hartley Rogers Jr, *Theory of recursive functions and effective computability,* MIT press, 1987.

[8] Christian Boitet, *Un essai de réponse à quelques questions théoriques et pratiques liées à la traduction automatique: définition d'un système prototype,* Université Joseph-Fourier-Grenoble I, 1976.

[9] Ilya Sutskever, Oriol Vinyals, Quoc V. Le, *Sequence to Sequence Learning with Neural Networks,* Advances in neural information processing systems, 2014.