NLP Preliminary Notes

Tony SeoiHong AuYeung

October 19, 2023

Contents

Chapter 1

Basics

1.1 Regular Expression

https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html

1.1.1 Word

- corpus: a computer-readable collection of text or speech
- lemma: a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense
- types: the number of distinct words in a corpus, $|V|$
- tokens: the total number N of running words
  e.g I like you and I love you. $\Longrightarrow$ 5 types and 7 tokens
- Herdan's Law(or Heap's Law): $|V| = kN^{\beta}$, $k > 0, 0 < \beta < 1$

1.2 Text Normalization

- 1. Define tokens as words, e.g. "I", "and", "you" etc
- 2. Define tokens as characters, e.g. "l", "o", "v", "e" etc
- 3. Byte-Pair Encoding, BPE

Byte-Pair Encoding: iteratively merge the most frequent neighbours. i.e.:

I like you and I love you.
$\Longrightarrow$ Vocabulary: _, d, e, i, k, l, n, o, u, y
1^{st} iteration: _, d, e, i, k, l, n, o, y, ou
2^{nd} iteration: _, d, e, i, k, l, n, o, you
3^{rd} iteration: _, d, e, i, k, l, n, o, you, i_
4^{th} iteration: _, d, i, k, l, n, o, you, i_, e_

After four iterations, we get the following tokens to represent the former sentance: _, d, i, k, l, n, o, you, i_, e_.

props of BPE: could handle the unknown words

1.3 Word Normalization

- Lemmatization: task of determining that two words have the same root, despite their surface differences. e.g. Porter Stemmer

- Sentence Segmentation: segment a sentence into words by punctuation or some roles

1.4 Minimum Edit Distance

- Def - Minimum Edit Distance: minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another

- Def - Levenshtein Distance:= $\#\{operations : sentance1 -> sentance2\}$

To compute the MED, we apply dynamic programming. Given a string with n characters $X = X[1, ..., n]$ and a string with m characters $Y = Y[1, ..., m]$, we define the MED between substrings $X1, ..., i$ and $Y[1, ..., j]$ as $D(i, j)$. Thus, $MED(X, Y) = D(n, m)$. Besides, define $d(t)$ as deleting character t, $i(t)$ as inserting character t and $s(t, q)$ as substituting character t by q. Then,

$$D(i,j) = min \begin{cases} D(i-1,j) + d(X[i]) \\ D(i,j-1) + i(Y[j]) \\ D(i-1,j-1) + s(X[i], Y[j]) \end{cases}$$

Using the Levenshtein distance, the formulation becomes:

$$D(i,j) = min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + 2\mathbb{1}_{X[i] \neq Y[j]} \end{cases}$$

Pseudo-code:

- Def - Alignment: The alignment between 2 strings is exactly the optimal path when calculating their MED.

Algorithm 1 Dynamic Programming for MED

Require: $X = X[1, ..., n], Y = Y[1, ..., m]$
  $D(0, 0) \leftarrow 0$
  for i=1,...,n **do**
    $D(i, 0) \leftarrow D(i - 1, 0) + d(X[i])$
  end for
  for j=1,...,m **do**
    $D(0, j) \leftarrow D(0, j - 1) + i(Y[j])$
  end for
  for i=1,...,n **do**
    for j=1,...,m **do**
      $D(i, j) \leftarrow \min(D(i - 1, j) + d(X[i]), D(i, j - 1) + i(Y[j]), D(i - 1, j - 1) + s(X[i], Y[j]))$
    end for
  end for

Chapter 2

N-gram Language Models

2.1 Intro

Given a sentence(a sequence of words/tokens) $w_{1:n} := w_1 w_2 ... w_n$, we would like to model the language system in the view of probability. Thus, we would like to compute the probs(likelihood):

$$\begin{aligned} p(w_1 w_2 ... w_n) &= p(w_n|w_{1:n-1})p(w_{n-1}|w_{1:n-2})...p(w_2|w_1)p(w_1) \\ &= \Pi_{k=1}^{n} p(w_k|w_{1:k-1}) \end{aligned} \tag{2.1}$$

To reduce computations, we approximately compute the likelihood by assuming the Markov property of the sentence, which is: $p(w_k|w_{1:k-1}) = p(w_k|w_{k-1})$. And thus,

$$p(w_1 w_2 ... w_n) = Pi_{k=1}^{n} p(w_k|w_{k-1}) \tag{2.2}$$

which is called Bigram model.

To use more information, it is natural to improve the bigram model by using more historical information, leading to the N-gram model:

Definition 2.1.1 (N-gram model)

$$p(w_1 w_2 ... w_n) = \Pi_{k=1}^{n} p(w_k|w_{k-N+1:k-1}) \tag{2.3}$$

To estimate the probability by the training data(i.e. training corpus), we use a MLE estimator:

Definition 2.1.2 (counting function)

$$c(event) = \#\{event\} \tag{2.4}$$

For example, $c(vw)$ represents counting for word w appearing after word v.

Definition 2.1.3 (MLE estimator for N-gram)

$$\hat{p}(w_k|w_{k-N+1:k-1}) = \frac{c(w_{k-N+1:k-1}w_k)}{\sum_v c(v|w_{k-N+1:k-1})} = \frac{c(w_{k-N+1:k-1}w_k)}{c(w_{k-N+1:k-1})} \tag{2.5}$$

In specific for bigram model:

$$\hat{p}(w|v) = \frac{c(wv)}{c(v)} \tag{2.6}$$

However, N-gram model faces a few problems:

1. Unknown words: Unseen word in test set or unseen combination of words in test set
   $\Longrightarrow$Solutions:

$$\begin{cases} 1.\ \textit{First, choosing a fixed vocabulary } \tilde{V}\textit{;} \\ \quad \textit{Then, covert the words in training set but not in } \tilde{V} \textit{ to } <UNK> \\ 2.\ \textit{Replace all words that occur fewer than } t \textit{ times in training set to } <UNK> \end{cases}$$

2. Sparsity: A lot of probs might be 0 or very close to 0 because of low diversity of the corpus or the probs product
   $\Longrightarrow$Solutions:

$$\begin{cases} \textit{smoothing} \begin{cases} \textit{Laplace smooting} \\ \textit{Add-k smoothing} \end{cases} \\ \textit{backoff} \begin{cases} \textit{Katz backoff} \\ \textit{Stupid backoff (for huge LM)} \end{cases} \\ \textit{interpolation} \begin{cases} \textit{Absolute Discounting} \\ \textit{Kneser-Ney Discounting} \end{cases} \end{cases}$$

2.2 Smoothing

To solve the 0 probs problem in N-gram model, we would like to add some probability(small) to unseen words or unseen combinations, as well as keep the language model a true probability distribution, which is called smoothing or discounting.

2.2.1 Laplace smoothing

Laplace smoothing(a.k.a Add-1 smoothing) adjusts the probability by simply adding 1 count to each word or combination. In bigram example, that would be:

$$p_{Laplace}(wv) = \frac{c(wv)+1}{\sum_{w'} c(w'v)+1} = \frac{c(w|v)+1}{c(v)+V} \tag{2.7}$$

Cons: A sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.

2.2.2 Add-k smoothing

To move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count $k, 0 < k < 1$

$$p_{Add-k}(w|v) = \frac{c(wv)+k}{c(v)+kV} \tag{2.8}$$

2.3 Backoff

Backoff adjusts the probability by approximating an unseen event through a shorter history. i.e., if the n-gram we need has zero counts, we approximate it by backing off to the (n-1)-gram, and continue backing off until we reach a history that has some counts.

2.3.1 Katz backoff

Given a function α, the Katz backoff is defined as:

$$P_{BO}(w_k|w_{k-N+1:k-1}) = \begin{cases} \hat{p}(w_k|w_{k-N+1:k-1}), \text{ if } c(w_{k-N+1:k}) > 0 \\ \alpha(w_{k-N+1:k-1})P_{BO}(w_k|w_{k-N+2:k-1}), \text{ otherwise} \end{cases} \tag{2.9}$$

2.3.2 Stupid backoff

Stupid backoff gives up the idea of trying to make the language model a true probability distribution. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight.

$$S(w_k|w_{k-N+1:k-1}) = \begin{cases} \dfrac{c(w_{k-N+1:k-1}w_k)}{c(w_{k-N+1:k-1})}, \text{ if } c(w_{k-N+1:k}) > 0 \\ \lambda S(w_k|w_{k-N+2:k-1}), \text{ otherwise} \end{cases} \tag{2.10}$$

where λ is a fixed parameter and 0.4 works well.

2.4 Interpolation

Similar with simple linear interpolation, we would like to estimate the N-gram probability by 1 to N gram:

$$\hat{p}(w_k|w_{k-N+1:k-1}) = \sum_{i=1}^{N} \lambda_i p(w_k|w_{k-N+i}) \tag{2.11}$$

where λs are learned from a held-out corpus, by choosing their values to maximize the likelihood of held-out corpus.

2.4.1 Absolute Discounting

There is an experiment shows that except for the held-out counts for 0 and 1, all the other bigram counts in the held-out set could be estimated pretty well by just subtracting 0.75 from the count in the training set!

Given a fixed discount d, e.g. $d = 0.75$, absolute discounting for bigram is defined as:

$$P_{AD}(w_k|w_{k-1}) = \frac{c(w_{k-1}w_k) - d}{c(w_{k-1})} + \lambda(w_{k-1})P(w_k) \tag{2.12}$$

where $0 \leq d \leq 1$ and the second term is the unigram with an interpolation weight λ.

2.4.2 Kneser-Ney Discounting

To notice that, a word occurs after many different words should be more likely to be predicted. Thus, we first define a measure for this kind of 'continuation':

Definition 2.4.1 (Continuation)

$$\begin{aligned} P_{continuation}(w) &\propto |\{v : c(vw) > 0\}| \\ \Longrightarrow P_{continuation}(w) &= \frac{|\{v : c(vw) > 0\}|}{\sum_{w'} |\{v' : c(v'w') > 0\}|} \end{aligned} \tag{2.13}$$

And following this construction, the Interpolated Kneser-Ney discounting for Bigram is:

$$P_{KN}(w_k|w_{k-1}) = \frac{\max(c(w_{k-1}w_k) - d, 0)}{c(w_{k-1})} + \lambda(w_{k-1})P_{continuation}(w_k) \tag{2.14}$$

$$\lambda(w_{k-1}) = \frac{d}{c(w_{k-1})}|w : c(w_{k-1}w > 0)| \tag{2.15}$$

And the general form is:

$$\begin{aligned} P_{KN}(w_k|w_{k-N+1:k-1}) = &\frac{\max(c_{KN}(w_{k-N+1:k}) - d, 0)}{c_{KN}(w_{k-N+1:k-1})} \\ &+ \lambda(w_{k-N+1:k-1})P_{KN}(w_k|w_{k-N+2:k-1}) \end{aligned} \tag{2.16}$$

$$c_{NK}(\cdot) = \begin{cases} count(\cdot), \textit{ for the highest order} \\ coninuation(\cdot), \textit{ for lower orders} \end{cases} \tag{2.17}$$

$$P_{NK} = \frac{\max(c_{NK}(w) - d, 0)}{\sum_{w'} c_{NK}(w')} + \lambda(\epsilon)\frac{1}{V} \tag{2.18}$$

If we want to include an unknown word ¡UNK¿, it's just included as a regular vocabulary entry with count zero, and hence its probability will be a lambda-weighted uniform distribution $\frac{\lambda(\epsilon)}{V}$.

2.5 Perplexity

After constructing the N-gram language model, we would like to introduce a measure to evaluate different algorithms. Intuitively, likelihood of the whole text L might be a good choice. Therefore, we drive a measure called 'perplexity', which is related to the text likelihood.

Before deriving this, we first present the terms: entropy and cross entropy.

Definition 2.5.1 (Entropy) *Given a randon variable X with probability measure p, then entropy of X is defined as:*

$$H(p) = -\sum_{x \in X} p(x)log(p(x)) \tag{2.19}$$

where the base is usually chosen as 2, which means the entropy is measured in bits.

Definition 2.5.2 (Per-word entropy)

$$\frac{1}{n}H(W_{1:n}) = -\frac{1}{n}\sum_{w_{1:n}\in L} p(w_{1:n})log(p(w_{1:n})) \tag{2.20}$$

To measure the true entropy of a language, we need to consider sequences of infinite length. And by the Shannon-McMillan-Breiman theorem:

Theorem 2.5.1 (The Shannon-McMillan-Breiman theorem) *If the text is stationary and ergodic,*

$$\begin{aligned} H(L) :&= \lim_{n\to\infty}\frac{1}{n}H(W_{1:n}) \\ &= -\lim_{n\to\infty}\frac{1}{n}log(p(w_{1:n})) \end{aligned} \tag{2.21}$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. But natural language is not stationary. Thus, our statistical models only give an approximation to the correct distributions and entropies of natural language.

When we are using a model of p, say m, to approximate the true distribution p. We first define cross-entropy as:

Definition 2.5.3 (Cross-entropy) *Given a randon variable X with probability measure p, and m is an approximation of p, then entropy of X is defined as:*

$$H(p,m) = -\sum_{x\in X} p(x)log(m(x)) \tag{2.22}$$

Following the Shannon-McMillan-Breiman theorem, for a stationary erdogic process:

$$H(L,m) = -\lim_{n\to\infty}\frac{1}{n}log(m(w_{1:n})) \tag{2.23}$$

And it is shown that the cross-entropy $H(p,m)$ is an upper bound on the entropy $H(p)$, i.e.$H(p,m) \geq H(p)$. Thus, the difference between $H(p,m)$ and $H(p)$ is a measure of how accurate a model is.

Given a model $M = P(w_k|w_{k-N+1:k-1})$ and a sequence of words $w_1...w_T$, i.e. a test set, then the cross-entropy could be estimated as:

$$\begin{aligned} H(W) &= -\frac{1}{T}\log P(w_1...w_T) \\ &= -\frac{1}{T}\sum_{k=1}^{T}\log P(w_k|w_{k-N+1:k-1}) \end{aligned} \tag{2.24}$$

Then the perplexity of a model P is defined as 2 raised to the power of this cross-entropy:

$$\begin{aligned} Perplexity(P) :&= 2^{-\frac{1}{T}\sum_{k=1}^{T}\log P(w_k|w_{k-N+1:k-1})} \\ &= \sqrt[T]{\Pi_{k=1}^{T}\frac{1}{P(w_k|w_{k-N+1:k-1}}}) \end{aligned} \tag{2.25}$$

Chapter 3

Vector Semantics and Embeddings

Given vocabulary with $|V|$ distinct words, and N documents(usually some windows of a text). We would like to find a representation for each word(token), by:

$$\begin{cases} \textit{term-document matrix} \in \mathcal{R}^{|V|\times N} : \textit{row-¿emb for words; col-¿emb for doc} \\ \textit{term-term matrix} \in \mathcal{R}^{|V|\times|V|} : \textit{co-occurence of words} \\ \textit{TF-IDF: an improvement of term-doc matrix} \\ \textit{PPMI: an improvement of term-term matrix} \\ \textit{word2vec: short and dense embeddings} \\ \textit{Hidden units in a nn, e.g. RNN, LSTM...} \end{cases}$$

3.1 Term-document matrix and TF-IDF

3.1.1 Term-doc matrix

A term-doc matrix TD is a $|V| \times N$ matrix with elements

$$TD_{ij} = \#\{\textit{occurences of the } i^{th} \textit{ word in the } j^{th} \textit{ document}\}$$

where its rows are embeddings for each word and columns are embeddings for

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 3.1: term-doc matrix

each document.

3.1.2 TF-IDF

There is an issue that: raw frequency is very skewed and not very discriminative. To solve this problem, we follow the follwong 2 intuitions: 1. words that occur nearby frequently (maybe pie nearby cherry) are more important than words that only appear once or twice; 2. give a higher weight to words that occur only in a few documents.
Then, we define:

Definition 3.1.1 (TF-IDF)

- *Term frequency: the frequency of the word t in the document d*

$$tf_{t,d} = \log_{10}(\#\{t \text{ in } d\} + 1) \tag{3.1}$$

- *document frequency: the frequency of the doc d that the word t occurs*

$$df_t = \#\{d \text{ contains } t\} \tag{3.2}$$

- *Inverse df:*

$$idf_t = \log_{10}(\frac{N}{df_t}) \tag{3.3}$$

Then, the TF-IDF weighted value $w_{t,d}$ for wordt in doc d is defined as:

$$w_{t,d} = tf_{t,d} \times idf_t \tag{3.4}$$

3.2 Term-term matrix

Also called term-context matrix and different from term-doc matrix, term-term matrix uses word as column labels. A term-term matrix is of size $|V| \times |V_1|$ with elements are the oc-occurences of 2 terms, where $|V_1|$ is the size of context and it is common use a small size contexts(e.g. a window around the word, for example of 4 words to the left and 4 words to the right).

	computer	data	result	pie	sugar	count(w)
cherry	2	8	9	442	25	486
strawberry	0	0	1	60	19	80
digital	1670	1683	85	5	4	3447
information	3325	3982	378	5	13	7703
count(context)	4997	5673	473	512	61	11716

Figure 3.2: term-term matrix

3.2.1 PPMI

Rather than using the counts(co-occurences) of words as measurement, we use a mutual-information based method.

Definition 3.2.1

- *Mutual information:*

$$I(x,y) = \log_2 \frac{p(x,y)}{p(x)p(y)} \tag{3.5}$$

$$I(X,Y) = \mathbf{E}[I(x,y)] \tag{3.6}$$

$$= \sum_x \sum_y I(x,y) \tag{3.7}$$

- *Pointwise mutual information between a target word w and a context word c:*

$$PMI(w,c) = \log_2 \frac{p(w,c)}{p(w)p(c)} \tag{3.8}$$

Intuition for PMI: an estimate of how much more the two words co-occur than we expect by chance. To restrict the values to be positive, it is common to use the Positive-PMI(PPMI):

Definition 3.2.2 (PPMI)

$$PPMI(w,c) = \max(\log_2 \frac{p(w,c)}{p(w)p(c)}, 0) \tag{3.9}$$

Define f_{ij} as the number of co-occurence of target word w_i and context word c_j, then the MLE estimators are:

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}, p_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}, p_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \tag{3.10}$$

And thus we could form the PPMI matrix by:

$$PPMI_{ij} = \max(\log_2 \frac{p_{ij}}{p_{i*}p_{*j}}, 0) \tag{3.11}$$

However, PMI has the problem of being biased toward infrequent events; very rare words tend to have very high PMI values. To solve this problem, we slightly change the computation of the context prob $p(w)$ by:

$$PPMI_\alpha(w,c) = \max(\log_2 \frac{p(w,c)}{p(w)p_\alpha(c)}, 0) \tag{3.12}$$

$$p_\alpha(c) = \frac{count(c)^\alpha}{\sum_c count(c)^\alpha} \tag{3.13}$$

3.3 word2vec

Although TF-IDF and PPMI provides sensible computations, the embeddings of them are both sparse and long. To solve this problem, we use skip-gram with negative sampling(SGNS) to obtain short dense embeddings.

Intuition: train a classifier on a binary prediction task: "Is word w likely to show up near word c?"

Algorithm 2 word2vec

Require: A corpus with N words

1. Construct samples from window with length L: $(w_i, c_{i\pm l}, +), l = 1, ..., L$
2. Negative sampling from outside the window: $(w_i, c, -)$, c is randomly chosen
3. Training data = positive samples + negative samples
4. Initialize matrix W and C randomly
5. $p(+|w, c) = \sigma(w \cdot c)$
6. Optimize W and C by MLE or MAP, using SGD
7. A word is represented by: w_i or $w_i + c_i$

Chapter 4

HMM and CRF: Sequence Labeling

4.1 Intro

Given a sentence, a classic task is to distinguish the named entities of each word, e.g. names, locations, nouns and verbs etc. That is, given a sequence of words/tokens, we would like to classify their labels: $P(y_{1:N}|x_{1:N})$. And usually, there are two types of models, generative ones($P(y|x) \propto P(x|y)P(y)$) and discriminant ones($P(y|x) = f_\theta(x, y)$), resulting to Hidden Markov Model(HMM) and Conditional Random Field(CRF) respectively.

4.2 Hidden Markov Model

Given training data $\{(x_n, y_n)\}_{n=1}^N$, the likelihood is:

$$\begin{aligned} p(x_{1:N}, y_{1:N}) &= p(x_{1:N}|y_{1:N})p(y_{1:N}) \\ &= p(x_{1:N}|y_{1:N})\Pi_{n=1}^N p(y_n|y_{1:n-1}) \end{aligned} \tag{4.1}$$

To make the previous likelihood computable, HMM makes 2 assumptions:

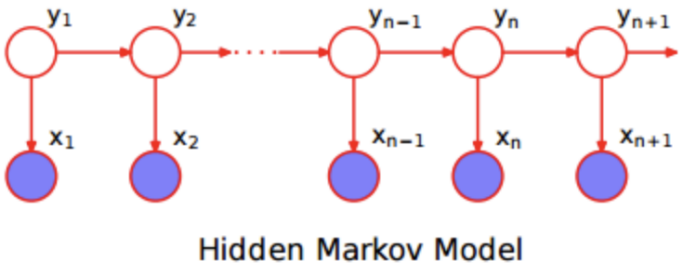

Figure 4.1: HMM

Definition 4.2.1 (HMM assumptions)

1. *Markov property:* $p(y_n|y_{1:n-1}) = p(y_n|y_{n-1})$
2. *Independent outputs:* $p(x_n|y_{1:n}) = p(x_n|y_n)$

Thus,

$$p(x_{1:N}, y_{1:N}) = \Pi_{n=1}^{N} p(x_n|y_n)p(y_n|y_{n-1}) \tag{4.2}$$

Let the labels have K states, we use MLE estimator(naive bayes) to approximate the probabilities $p(x_n|y_n)$ and $p(y_n|y_{n-1})$:

$$\begin{cases} a(i,j) = p(y_n = j|y_{n-1} = i) \\ b_i(x_n) = p(x_n|y_n = i) \\ \quad i, j = 1, ..., K \end{cases} \tag{4.3}$$

Then we obtain $\hat{a}(i,j)$ and $\hat{b}_i(x_n)$. Given a new word sequence $\{\hat{x}_m\}_{m=1}^N$, we classify their labels $\{\hat{y}_m\}_{m=1}^N$ by:

$$\begin{aligned} \hat{y}^* &= \arg\max_{\hat{y}_{1:M}} p(\hat{y}_{1:M}|\hat{x}_{1:M}) \\ &\propto \arg\max_{\hat{y}_{1:M}} \Pi_{m=1}^{M} p(\hat{y}_m|\hat{y}_{m-1})p(\hat{x}_m|\hat{y}_m) \\ &= \arg\max_{\hat{y}_{1:M}} \Pi_{m=1}^{M} \hat{a}(\hat{y}_{m-1}, \hat{y}_m)\hat{b}_{\hat{y}_m}(\hat{x}_m) \end{aligned} \tag{4.4}$$

To solve 4.4, we use the dynamic programming, which results to the Viterbi algorithm:

Algorithm 3 Viterbi

Require: sequence of word $\hat{x}_{1:M}$; estimated transition matrix $\hat{A}_{i,j} = \hat{a}(i,j)$ and estimated emission $\hat{b}_i(x)$
  path=[]
  for k=1:K **do**
    $v[k,1] = \pi_k b_k(\hat{x}_1)$
  end for
  for m=2:M **do**
    for k=1:K **do**
      $k' = \arg\max_{k'} v[k', m-1]\hat{a}(k',k)\hat{b}_k(\hat{x}_m)$
      $v[k,m] = v[k', m-1]\hat{a}(k',k)\hat{b}_k(\hat{x}_m)$
      path.append(k')
    end for
  end for
  $maxprob = max_k v[k,M]$
  path.append($\arg\max_k v[k,M]$)
  return *maxprob*, path

4.3 Conditional Random Fields

Different from HMM, CRFs is discriminant, which means it computes $P(Y|X)$ directly. Let the labels have K states, we adopt the softmax trick from multi-class classification, by using L different feature functions $F_l(X,Y), l = 1, ..., L$.

Then,

$$P(Y|X,w) = \frac{exp(\sum_{l=1}^{L} w_l F_l(X,Y))}{\sum_{Y' \in \mathcal{Y}} exp(\sum_{l=1}^{L} w_l F_l(X,Y'))} \tag{4.5}$$

In nlp tasks, i.e. sequence labeling, it is sensible to assume the feature functions forms like:

$$F_l = \sum_{n=1}^{N} f_l(y_{n-1}, y_n, X, n) \tag{4.6}$$

Therefore, the learning process of w becomes:

$$\begin{aligned}
\hat{w} &= \arg\max_w P(y_{1:N}|x_{1:N}, w) \\
&= \arg\max_w exp(\sum_{l=1}^{L} w_l F_l(X,Y)) \\
&= \arg\max_w exp(\sum_{l=1}^{L} w_l \sum_{n=1}^{N} f_l(y_{n-1}, y_n, X, n)) \\
&= \arg\max_w \sum_{n=1}^{N} \sum_{l=1}^{L} w_l f_l(y_{n-1}, y_n, X, n)
\end{aligned} \tag{4.7}$$

and w could be optimize by SGD easily. Also, it is straightforward to add regularizor, or prior.

To inference a given sequence $\hat{x}_{1:M}$

$$\begin{aligned}
\hat{y}^* &= \arg\max_{\hat{y}_{1:M}} P(\hat{y}_{1:M}|\hat{x}_{1:M}, \hat{w}) \\
&= \arg\max_{\hat{y}_{1:M}} exp(\sum_{m=1}^{M} \sum_{l=1}^{L} \hat{w}_l f_l(\hat{y}_{m-1}, \hat{y}_m, \hat{X}, m)) \\
&= \arg\max_{\hat{y}_{1:M}} \sum_{m=1}^{M} \sum_{l=1}^{L} \hat{w}_l f_l(\hat{y}_{m-1}, \hat{y}_m, \hat{X}, m)
\end{aligned} \tag{4.8}$$

Again, the optimal $\hat{y}_{1:M}$ could be found by Viterbi algorithm:

$$v[k,m] = \max_{k'} v[k', m-1] + \sum_{l=1}^{L} \hat{w}_l f_l(k', k, \hat{x}_m, m) \tag{4.9}$$

Chapter 5

RNN and its variants

5.1 Intro

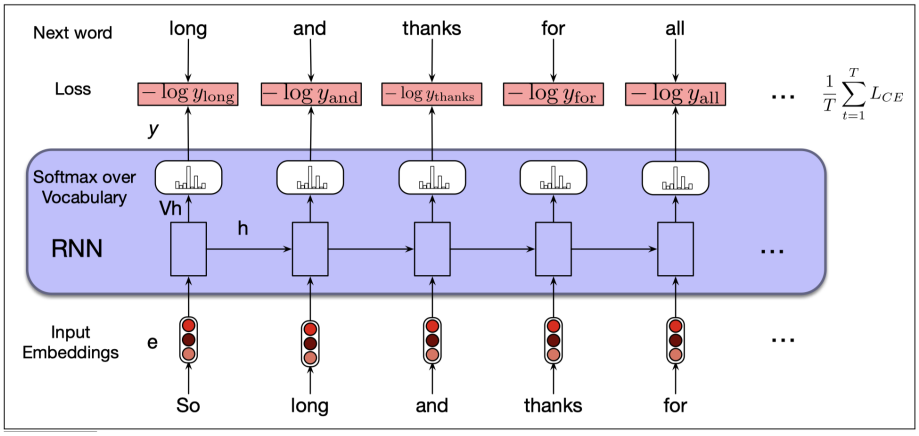

Figure 5.1: classic RNN

A classic RNN module is described as follows:

$$e_t = Ex_t \tag{5.1}$$
$$h_t = g(Uh_{t-1} + We_t) \tag{5.2}$$
$$y_t = softmax(Vh_t) \tag{5.3}$$

where E is described as an embedding transformation, e_t is the corresponding embedding and $g(\cdot)$ is a activation function(e.g. ReLU or tanh).

To reduce computations, we set V as E^T and thus,

Definition 5.1.1 (RNN module)

$$e_t = Ex_t \tag{5.4}$$
$$h_t = g(Uh_{t-1} + We_t) \tag{5.5}$$
$$y_t = softmax(E^T h_t) \tag{5.6}$$

5.2 RNN Applications

5.2.1 Sequence Labeling

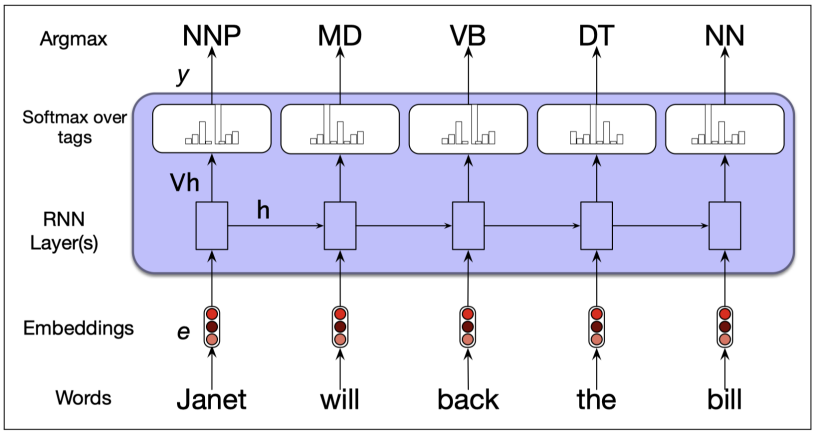

Figure 5.2: RNN for sequence labeling

5.2.2 Sequence Classification

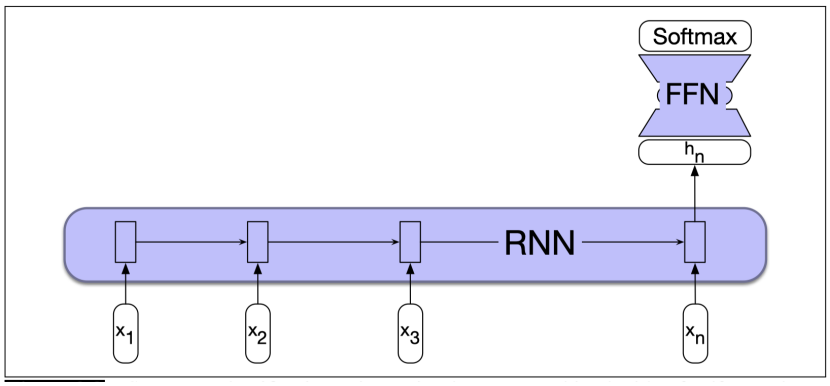

Figure 5.3: RNN for sequence classification

5.2.3 Sequence Generation

5.2.4 Machine Translation

In the machine translation task, different from the previous tasks, we adopt an encoder-decoder framework to generation the translations which might have different length against the original sentence.

To improve model performance, it is common to use "teacher forcing" strategy in machine translation. That is, instead generating the next word in translated sentence using the previous generation, we use the groundtruth translated

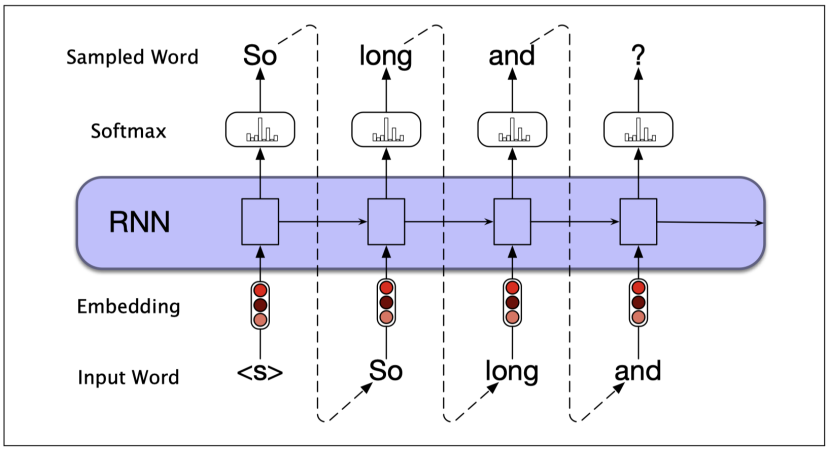

Figure 5.4: RNN for sequence generation, also called autoregressive generation

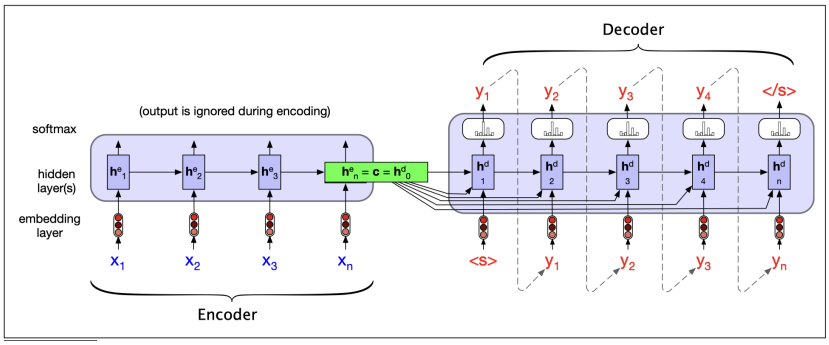

Figure 5.5: RNN for machine translation: a encoder-decoder framework

word at this position as the input, which would learn a translating style close to the training data.

5.3 More Complex Architectures

5.3.1 Stacked RNN

Similar to stacking layers in a simple fully-connected neural net, it is sensible to stack multiple layers of RNN, which could help us to include representations at different levels of abstraction across layers.

5.3.2 Bidirectional RNN

Rather than consider one direction of a sentence in classic RNN, Bidirectional RNN provides more information about the context. To do this, it's typically to run 2 separate RNNs:

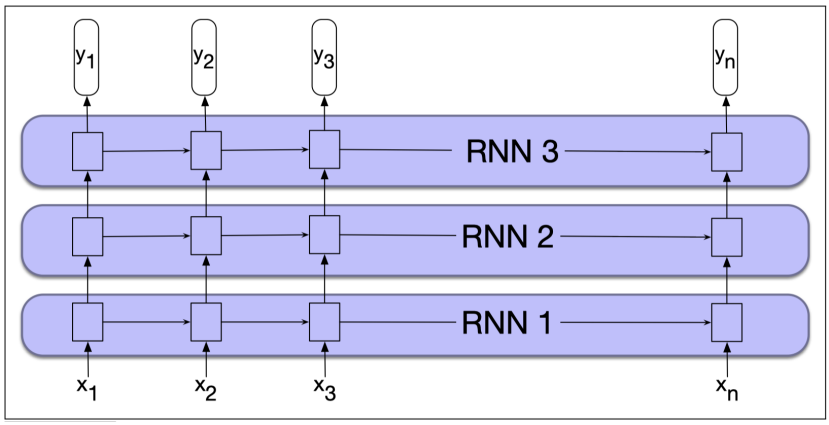

Figure 5.6: Stacked RNN

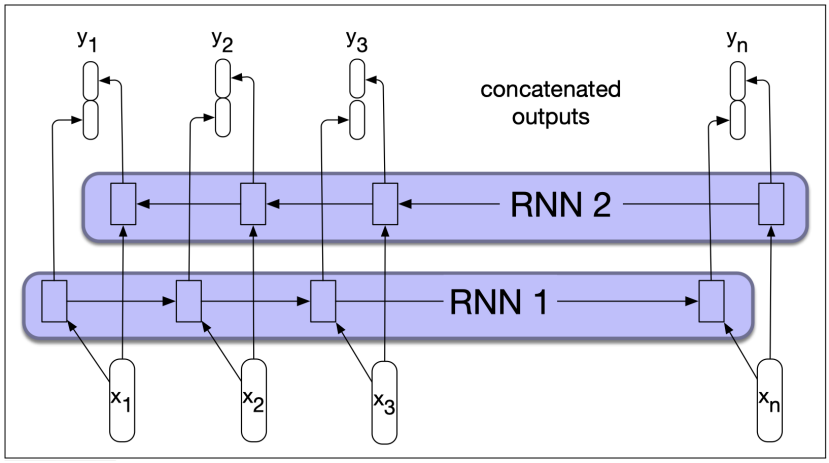

Figure 5.7: Bidirectional RNN

$$h_t^f = RNN_{forward}(x_1, ..., x_t) \tag{5.7}$$
$$h_t^b = RNN_{backword}(x_1, ..., x_t) \tag{5.8}$$
$$h_t = h_t^f \oplus h_t^b \ or \ h_t^f + h_t^b \ or \ h_t^f \otimes h_t^b \tag{5.9}$$

where $\oplus$ is the vector concatenation and $\otimes$ is element-wise multiplication.

5.3.3 LSTM

RNN has two main cons: 1. limited access to historical data, i.e. too focus on local information h_{t-1};2. params would be backprobagated through the whole sentence, see 5.1.1, resulting in grad vanishing, i.e. $grad \to 0$.

To get access to distant info as well as solve the gradient problem, the Long-Short term Memory model(LSTM) is proposed, whose architecture is given as followings:

Definition 5.3.1 (LSTM) *Given token x_t and historical hidden info h_{t-1}, c_{t-1},*

$$f_t = \sigma(U_f h_{t-1} + W_f x_t) \tag{5.10}$$
$$g_t = tanh(U_g h_{t-1} + W_g x_t) \tag{5.11}$$
$$i_t = \sigma(U_i h_{t-1} + W_i x_t) \tag{5.12}$$
$$o_t = \sigma(U_o h_{t-1} + W_o x_t) \tag{5.13}$$
$$\tag{5.14}$$

where f_t is forget gate, i_t is add gate and o_t is output gate. Then,

$$k_t = f_t \odot c_{t-1} \tag{5.15}$$
$$j_t = g_t \odot i_t \tag{5.16}$$
$$c_t = k_t + j_t \tag{5.17}$$
$$h_t = c_t \odot o_t \tag{5.18}$$

where $\odot$ is the Hardamard product.

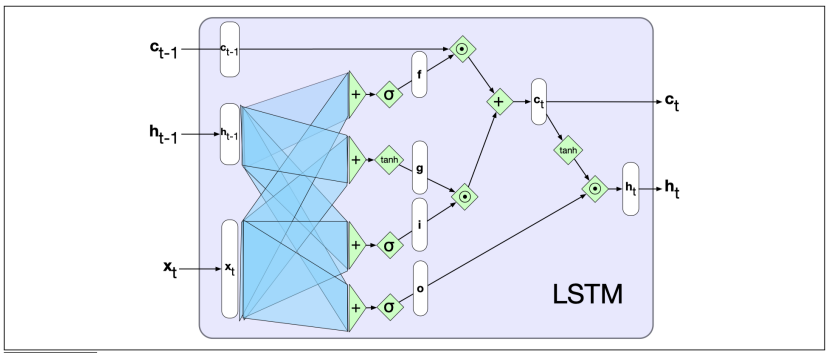

Figure 5.8: LSTM

5.3.4 Attention

Recap that in the task of machine translation, an encoder-decoder framework is used, which encode the given sentence with length N into a hidden state h_N and input this hidden state as an input to decode. This final hidden state is thus acting as a bottleneck: it must represent absolutely everything about the meaning of the source text, since the only thing the decoder knows about the source text is what's in this context vector.

To solve the bottleneck problem, an attention mechanism is proposed, which allows the decoder to get information from all the hidden states of the encoder, not just the last hidden state.

The idea of attention is to create the single fixed-length vector c by taking a weighted sum of all the encoder hidden states. At each state i during decoding, for every state j, compute a score:

$$score(h^d_{i-1}, h^e_j) =< h^d_{i-1}, h^e_j > \tag{5.19}$$

and usually, the inner-product is set as a dot product. Then the weights α_{ij} is given by softmax:

$$\alpha_{ij} = \frac{score(h^d_{i-1}, h^e_j)}{\sum_k score(h^d_{i-1}, h^e_k)} \tag{5.20}$$

Thus, an weighted attention context is given as:

$$c_i = \sum_j \alpha_{ij} h^e_j \tag{5.21}$$

The full attention mechanism is given as follows:

Definition 5.3.2 (Attention)

$$\alpha_{ij} = \frac{score(h^d_{i-1}, h^e_j)}{\sum_k score(h^d_{i-1}, h^e_k)}, \forall j \in encoder \tag{5.22}$$

$$c_i = \sum_j \alpha_{ij} h^e_j \tag{5.23}$$

$$h^d_i = g(W x_t + U h^d_{i-1} + Q c_i) \tag{5.24}$$

$$y = softmax(V h^d_i) \tag{5.25}$$

where g is an activation function and W, U, Q, V are learnable params.

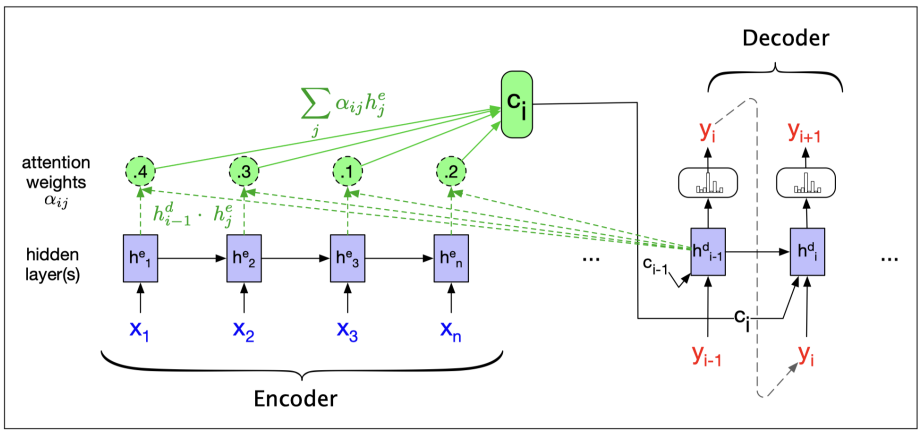

Figure 5.9: Attention

Besides, the score function could also be parameterized by a matrix W_s,

$$score(h^d_{i-1}, h^e_j) = (h^d_{i-1})^T W_s h^e_j \tag{5.26}$$

which enables that the dimension of encoding and decoding hidden state could be different.

Chapter 6

Transformer and Bert

6.1 Intro

$$\textit{Pretrained language model}\begin{cases}\textit{Transformer: self-attention + positional encodings}\\ \textit{BERT: a bidirectional transformer trained via masked language modeling}\end{cases}$$

6.2 Transformer

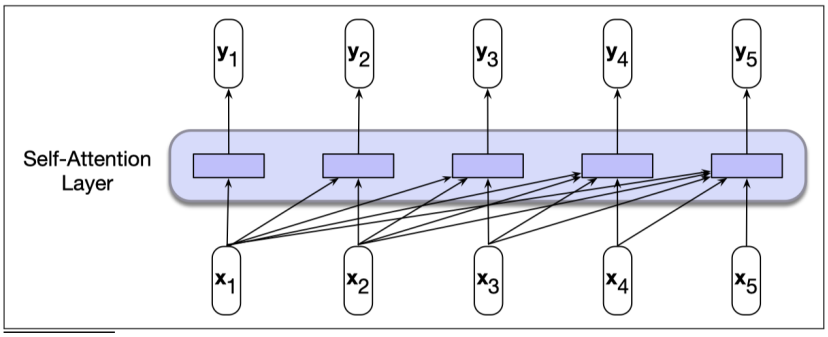

Figure 6.1: Self-attention mechanism

Instead of extracting historical data via the hidden layer h_{i-1} in RNN-based models, Transformer adopts a self-attention mechanism to allow it extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs, resulting in a paral-lelable computing process.

A bit different from attention, self-attention adopts a more sophisticated way to represent how words can contribute to the representation of longer inputs by

defining "query", "key" and "value":

$$Q = XW^Q \in \mathcal{R}^{N \times d_k} \tag{6.1}$$
$$K = XW^K \in \mathcal{R}^{N \times d_k} \tag{6.2}$$
$$V = XW^V \in \mathcal{R}^{N \times d_v} \tag{6.3}$$
$$\tag{6.4}$$

where $W^Q \in \mathcal{R}^{d \times d_k}$, $W^K \in \mathcal{R}^{d \times d_k}$ and $W^V \in \mathcal{R}^{d \times d_v}$.

Then the score matrix with elements $S_{ij} = score(y_i, y_j)$ is computed by:

$$S = softmax(\frac{QK^T}{\sqrt{d_k}}) \tag{6.5}$$

thus, we obtain word representations by:

$$SelfAttention = softmax(\frac{QK^T}{\sqrt{d_k}})V \in \mathcal{R}^{N \times d_v} \tag{6.6}$$

which could be forwarded to a FFN layer and trained via classification style.

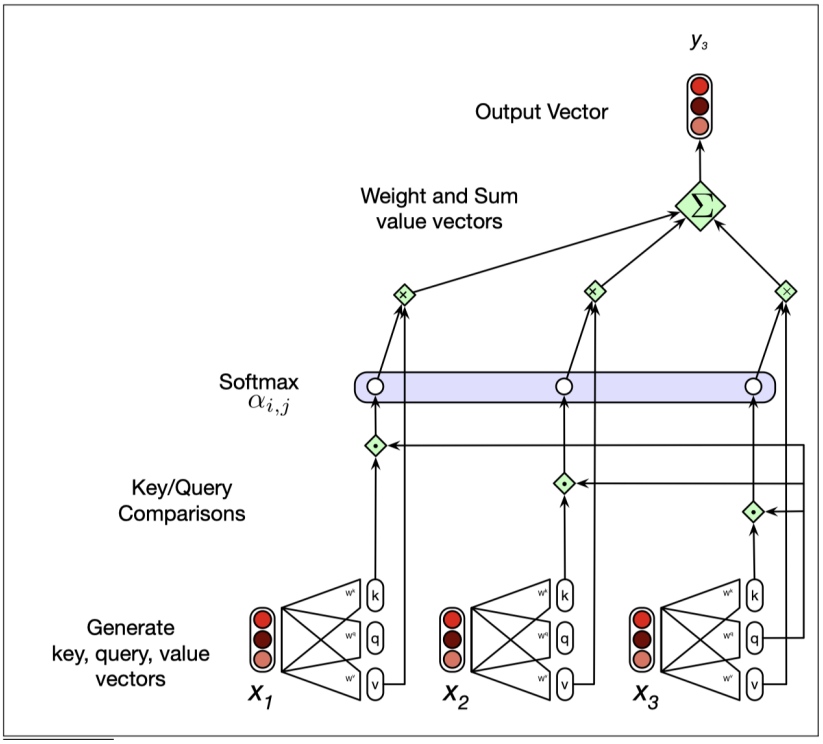

Figure 6.2: Calculating value of y_3, from a transformer

Also, notice that Transformer use the previous tokens as input. To prevent the leaking of future words, we should let the attention weight of these words as 0, which is equivalent to set the upper-triangle elements of S as $-\infty$.

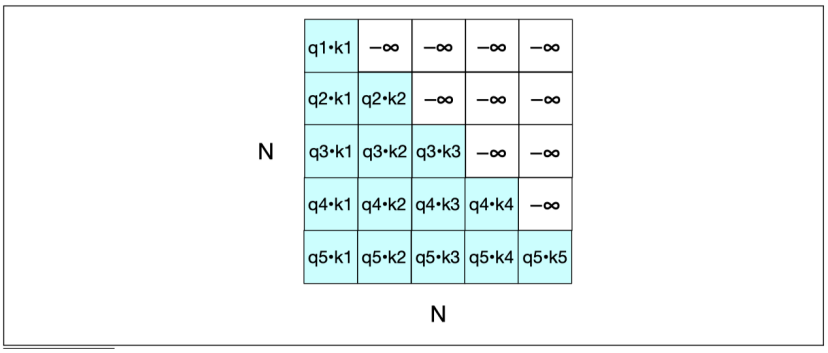

Figure 6.3: Enquiry matrix of Transformer

6.2.1 Transformer Block

After that, a transformer block is constructed with the following modules: a self-attention layer, residual connections, layer normalizations and a feedforward layer, which is shown in 6.4:

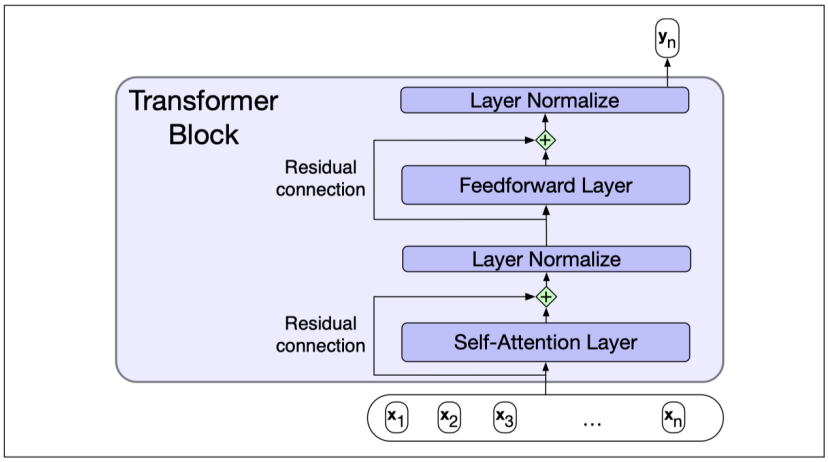

Figure 6.4: Transformer Block

Besides, a residual connection is given by:

$$z = SelfAttention(x) + x \tag{6.7}$$

and the layer normalize is:

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{6.8}$$

$$\sigma^2 = \frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2 \tag{6.9}$$

$$\hat{x} = \frac{x - \mu}{\sigma} \tag{6.10}$$

$$LayerNorm(x) = \gamma\hat{x} + \beta \tag{6.11}$$

where γ and β are learnable params.

6.2.2 Multihead Attention

The different words in a sentence can relate to each other in many different ways simultaneously. Thus, rather than using one representation of words, it makes sense to use multiple representations, resulting in a multihead attention mechanism:

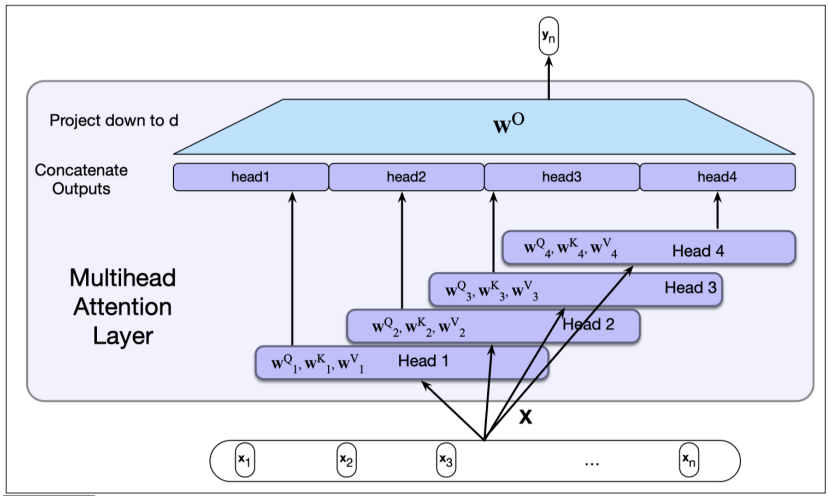

Figure 6.5: Multihead Transformer

6.2.3 Positional Encoding

Since the previous transformer is calculating word representations by weighted sum of the self-attention layer, it looses the positional information of each word! (While RNN-based models hold that, by the architecture designs) To solve this problem, One simple solution is to modify the input embeddings by combining them with positional embeddings specific to each position in an input sequence.

To get these positional embeddings, we start with randomly initialized embeddings corresponding to each possible input position up to some maximum length.

To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding.

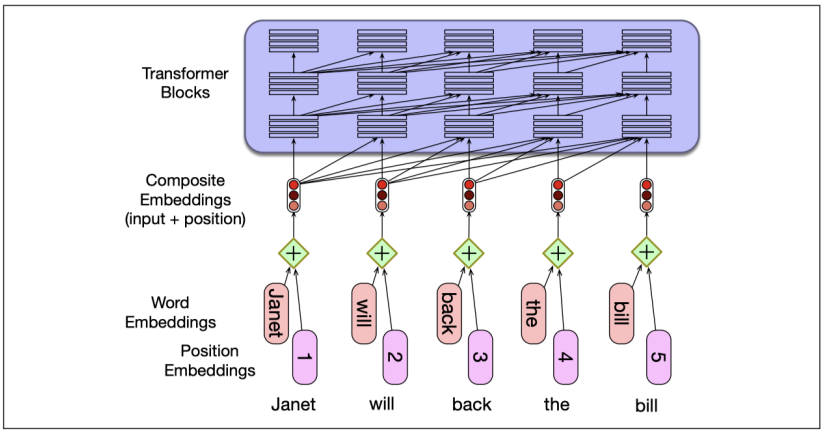

Figure 6.6: Positional encoding

6.2.4 Training a transformer: text generation task

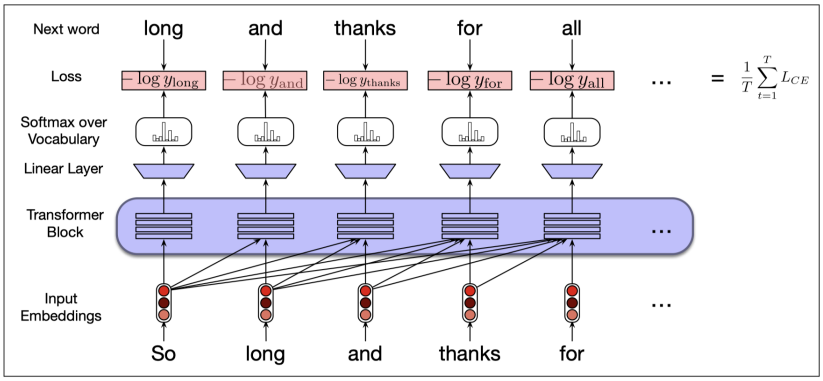

Figure 6.7: Transformer training for text generation, using a teacher forcing mechanism

6.2.5 Text generating

By greedy strategy or beam search

6.3 BERT

In addition to the single direction information used in Transformer, we would like to use information from the whole text/paragraph, resulting to a bidirectional

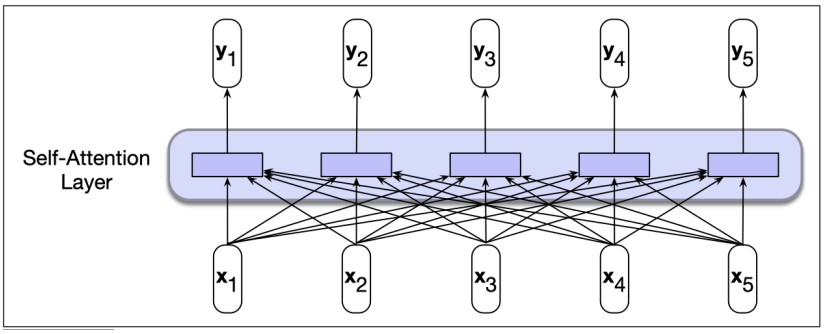

Figure 6.8: Bidirectional self-attention

self-attention mechanism. However, training this bidirectional self-attention model becomes a problem, as the future word is leaking and thus we're not able to train it via a text generating task.

To solve this problem, masked language modeling is proposed, leading to a popular large language model called BERT! BERT is trained by predicting the masked inputs:

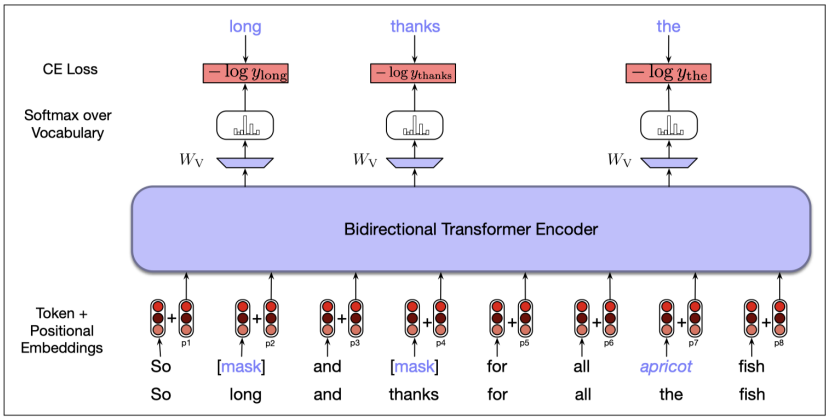

Figure 6.9: BERT training

where the training set is designed as: randomly choose 15% words to be masked, and

- 80% of them are replaced by token [MASK]
- 10% of them are replaced by any randomly chosen token
- 10% of them just stay still

The architecture of Bidirectional Transformer is almost the same as Transformer, without setting the upper-triangle of enquiry matrix as $-\infty$.

6.3.1 SpanBERT

In many NLP applications like question answering, syntactic parsing, coreference and semantic role labeling etc, the interest lies in identifying and classifying constituents or phrases, motivating a span based BERT training.

In SpanBERT, instead of nasking single word, we mask a sequence of words, i.e. a span, by randomly choose 15% spans to be masked, and

- 80% of them are replaced by tokens [MASK]
- 10% of them are replaced by any randomly chosen tokens
- 10% of them just stay still

Besides, we add an additional loss to let the model learn how to predict the tokens in a span by the boundary of this span, called Span Boundary Objective(SBO). Suppose a span starts with token x_s and ends with x_e, to predict a token x_i with positional encoding p_i in this span:

$$s = FFN([p_i; x_s; x_e]) \tag{6.12}$$

$$z = solfmax(Es) \tag{6.13}$$

$$L_{SBO} = -logP(x_i|z) \tag{6.14}$$

And then, the objective of SpanBERT is:

$$L_{BERT} = L_{MLM} + L_{SBO} \tag{6.15}$$

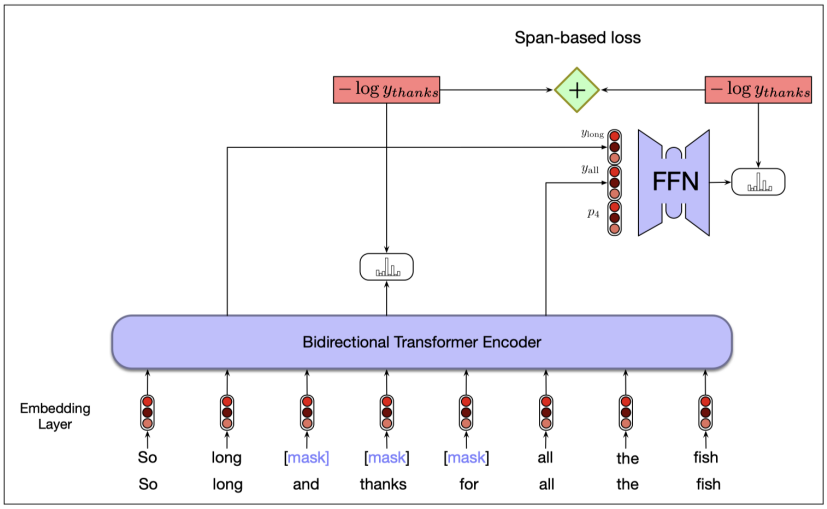

Figure 6.10: SBO in SpanBERT