10

DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

10.1 Word embeddings

10.1.1 Overview

With machine learning and deep learning methods widely adopted in the natural language processing (NLP), we start to see widespread and large scale application of NLP technologies in many different areas in industry.

NLP has been replacing human work simple tasks like sentiment analysis, search completion and started to aid human beings in challenging tasks like text summerization, writing, question-answering, etc. Modern NLP tasks heavily hinge on recent progress technologies like word embedding and pre-trained models.

Text data consisting of tokens from a large vocabulary. In many text data related tasks, such as sentiment classification, we need to represent text data by numeric values. One naive way to represent the feature of a word is the one-hot word vector, whose length of the typical size of a dictionary. Such sparse representation does not capture the relations among words (i.e., meanings, lexical semantic) and are thus not considered as good features for advanced natural language processing tasks, such as language modeling. A much better alternative is to embed each word vector into a smaller dense vector (typical dimensionality ranges from 25 to 1,000) that encodes semantic meaning [Figure 10.1.1].

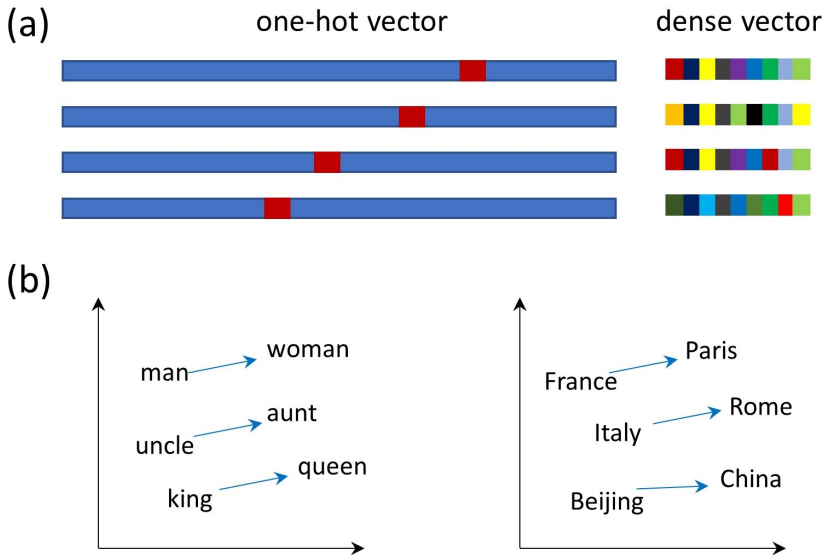


Figure 10.1.1: (a) Embedding layer maps large, sparse one-hot vectors to short, dense vectors. (b) Example of low dimensional embeddings that capture semantic meanings.

The obtained word embeddings can be fed into neural network like feed-forward network and recurrent neural network for tasks like text classification and text generation.

10.1.2 Word2Vec

10.1.2.1 *The model*

In subsubsection 10.1.3.1, we introduce a SVD based matrix decomposition method to map one-hot word vector to semantic meaning preserving dense word vector. This section, we introduce a neural networ based method. The two classical methods, called continuous bags of words (**CBOW**)[1] and **Skip-gram**[2]. Both methods employ a three-layer neural networks [Figure 10.1.2], taking a one-hot vector as input and predict the probability of its nearby words. In CBOW, the inputs are surrounding words within a context window of size c and the goal is to predict the central word (same as multi-class classification problems); in Skip-gram, the input is the single central word and the goal is to predict its surrounding words within a context window.

Denote a sequence of words $w_1, w_2, ..., w_T$ in a text, the objective of a Skip-gram model is to maximize the likelihood of observing the occurrence of its surrounding words within a context window of size c, given by

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p\left(w_{t+j}|w_t\right)$$

where we have assumed conditional independence given word w_t.

In the neural networks of Skip-gram and CBOW, we use Softmax in the output layer for classification probability, given by

$$p(w_k|w_j) = \frac{\exp(v'_k \cdot v_j)}{\sum_{i=1}^{V}\exp(v'_i \cdot v_j)},$$

where V is the size of the vocabulary, v_i is the column i of the input matrix W, and v'_i is the row i in the output matrix W'.

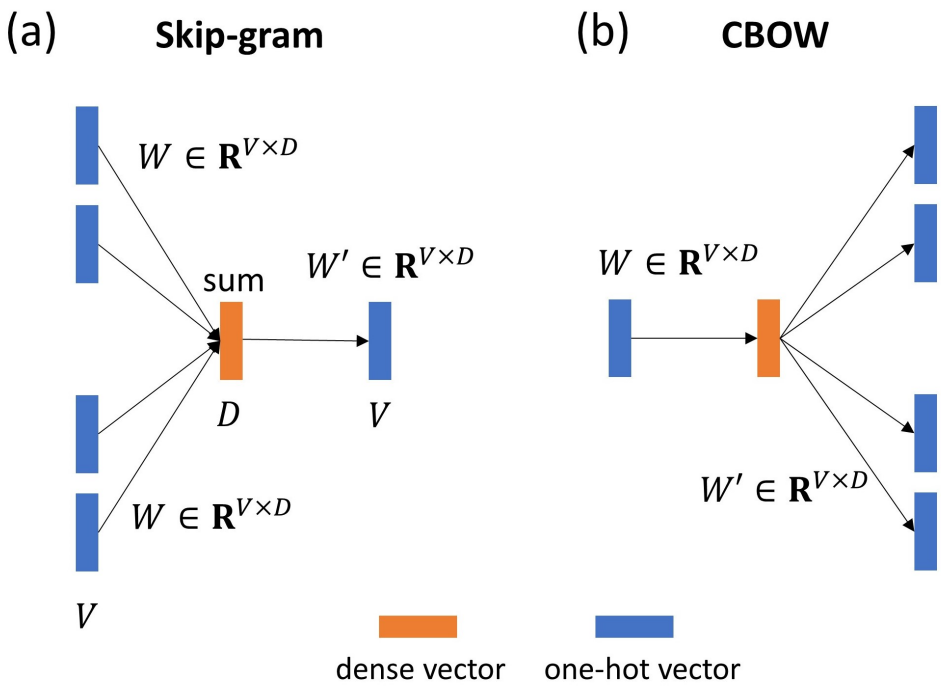


Figure 10.1.2: (a) The Skip-gram architecture that predicts surrounding words given the central word. (b) The CBOW architecture that predicts the central word given its surrounding context words. The one-hot vector has size V; the dense vector has length $D \ll V$. Also note that no nonlinearity activation is applied between input and hidden layer.

The neural network weights of the Skip-gram model are optimized to maximize the observation of a text consisting of words $w_1, w_2, ..., w_T$, which can then be written by

$$\begin{aligned}
&\arg\max_{v,v'} \sum_{t=1}^{T} \sum_{-c \le j \le c, j \neq 0} \log p\left(w_{t+j} | w_t\right) \\
&= \arg\max_{v,v'} \sum_{t=1}^{T} \sum_{-c \le j \le c, j \neq 0} \log \frac{\exp\left(v'_{t+j} \cdot v_{t)}\right)}{\sum_{w \in V} \exp\left(v'_w \cdot v_t\right)} \\
&= \arg\max_{v,v'} \sum_{t=1}^{T} \sum_{-c \le j \le c, j \neq 0} \left[v'_{t+j} \cdot v_t - \log \sum_{w \in V} \exp\left(v'_w \cdot v_t\right) \right]
\end{aligned}$$

In the CBOW model, the optimization problem becomes

$$\begin{aligned}&\arg\max_{v,v'} \sum_{t=1}^{T} \log p\left(w_t | w_{t-c}, ..., w_{t+c}\right) \\ =&\arg\max_{v,v'} \sum_{t=1}^{T} \log \frac{\exp(v'_t \cdot \sum_{-c\leq j\leq c, j\neq 0} v_j)}{\sum_{i=1}^{V} \exp(v'_i \cdot \sum_{-c\leq j\leq c, j\neq 0} v_j)}\end{aligned}$$

In both the Skip-gram and the CBOW model, each word will have two embeddings, v_i and v'_i in the input matrix W and the output matrix W', respectively. Notable, the embedding v'_i corresponds to the dense word vector that produces one-hot probability vector in the output. For these two embeddings, we can use one of them, a mixed version of them, and a concatenated one.

With the trained embeddings for each word, we can assemble then into a matrix of size $D \times V$, which is also called an Embedding layer. In applications, the one-hot word vector is fed into the embedding layer and produce the corresponding dense word vectors. From the computational perspective, we do not need to perform matrix multiplication; instead, we can view the Embedding layer as a dictionary that maps integer indices of the word to dense vectors.

In Skip-gram, the weight associated with each word receives adjustment signal (via gradient descent) from its surrounding context words. In CBOW, a central word provides signal to optimize the weights of its multiple surrounding words. Skip-gram is more computational expensive than CBOW as the Skip-gram model has to make predictions of size $O(cV)$ while CBOW makes prediction on the scale of $O(V)$. Further, because of the averaging effect from input layer to hidden layer in CBOW, CBOW is less competent in calculating effective word embedding for rare words than Skip-gram.

10.1.2.2 *Visualization*

We can visualize the word embedding space by projecting onto a 2D plane using two leading principal components [Figure 10.1.3]. The neighboring words of *apple* include *macintosh, microsoft, ibm, Windows, mac, intel, computers* as well as *wine, juice*, which capture to some extent the two common meanings in the word *apple*. This example also reveals the drawback of the word2vec approach, where we associate each token with a fixed/static embedding irrespective of context. For example, *apple* in *I like to eat an apple* vs *Apple is great company* means two different things and have the same embedding. The context-dependent embeddings will be discussed in section 10.6.

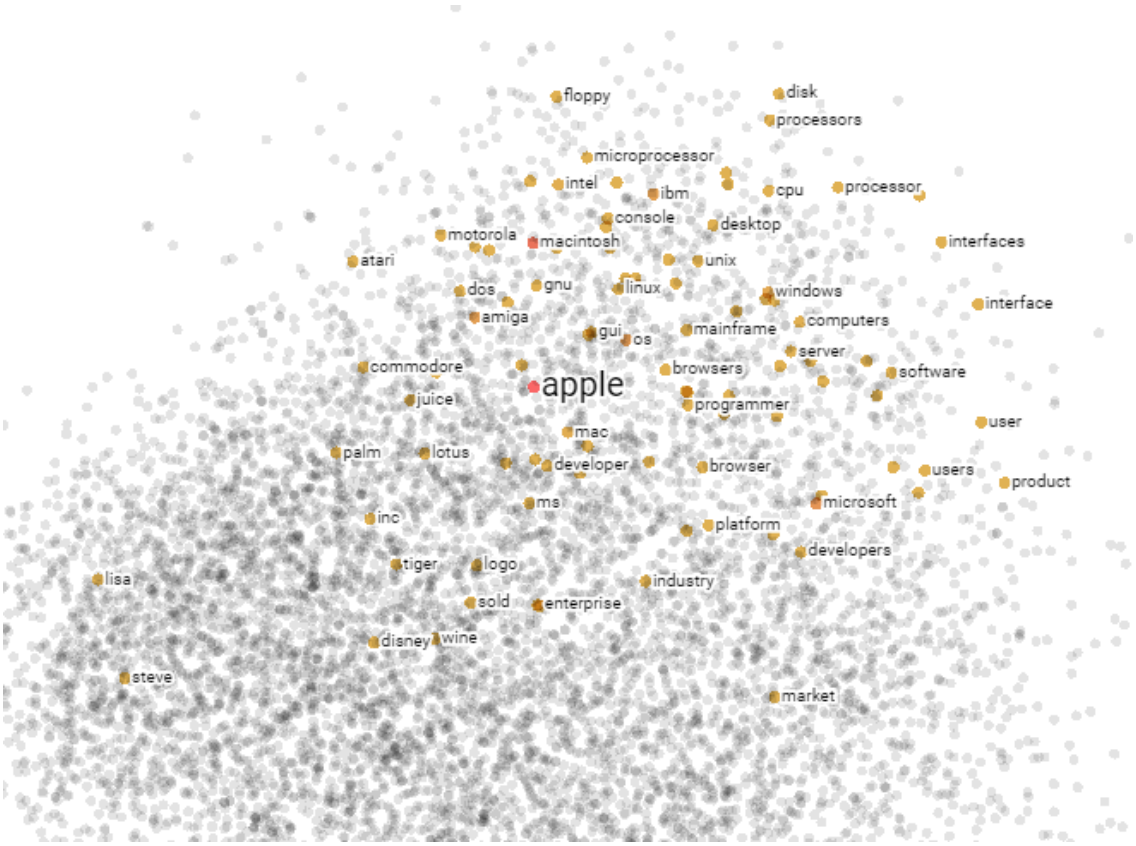


Figure 10.1.3: Visualization of neighboring words of *apple* in a 2D low-dimensional space (first two components via PCA). Image from Tensorflow projector.

10.1.3 GloVe

10.1.3.1 *SVD based word embeddings*

To efficiently carry out tasks like clustering, classification based on text data, we usually need to find numerical representation of text data such that many algorithms can process them. One naive way to represent the feature of a word as the one-hot word vector, whose length of the typical size of a dictionary. Such sparse representation does not capture the relations among words (i.e., meanings, lexical semantic, grammar requirements) and are thus not considered as a good feature for advanced natural language processing tasks, such as language modeling. A much better alternative is to embed each word vector into a smaller dense vector (typical dimensionality ranges from 25 to 1,000) that encodes semantic meaning and syntactic properties. A basic test on the ability to capture semantic and syntactic information is to be able to answer questions like

- Semantic questions like "Being is to China as Berlin is to [·]".

count	I	love	like	NLP	math
I	0	1	1	0	0
love	1	0	0	1	0
like	1	0	0	0	1
NLP	0	0	1	0	0
math	0	1	1	0	0

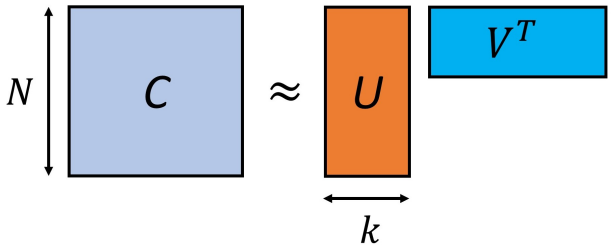


Figure 10.1.4: (left) Example of co-occurrence matrix constructed from corpus "I love math" and "I like NLP". The context window size of 2. (right) We can obtain lower-dimensional word embeddings from SVD truncated factorization of the co-occurrence matrix. Such low-dimensional embeddings captures important semantic and syntactic information in the co-occurrence matrix.

- Syntactic questions like "dance is to dancing as run is to [·]".

Here we introduce a way to obtain low-dimensional representation of a word vector that capture the semantic and syntatic relation between words by performing SVD on a co-occurrence matrix of a large corpus. Co-occurrence matrix is a big matrix whose entry encode the frequency of a pair of words occurring together within a fixed length context window. More formally, let M be a co-occurrence matrix, and we have

$$M_{ij} = \frac{\#(w_i, w_j)/n_{pair}}{\#(w_i)/n_{words} \cdot \#(w_j)/n_{words}}$$

where $\#(w_i, w_j)$ is the number of co-occurrence of words w_i and w_j within a context window, n_{pair} is the total number pairs, n_{words} is the total number of words.

Another popular matrix to capture the co-occurrence information is the the **point-wise mutual information (PMI)**. PMI entry for a word pair is defined as the probability of their co-occurrence divided by the probabilities of them appearing individually,

$$M_{ij}^{PMI} = \log \frac{p(w_i, w_j)}{p(w_i)p(w_j)} \approx \log M_{ij}.$$

The co-occurrence information captures to some extent both semantic and syntactic information. For example, terms tend to appear together either because they have related meanings (a semantic relationship, e.g., *write* and *book*) or because the grammar rule specifies so (a syntactic relation, e.g., verbs and *to*).

By using truncated SVD to decompose the co-occurrence matrix, we obtain the low-dimensional word vectors that preserve the co-occurrence information, or the semantic and syntactic relation implied by the co-occurrence information. For example, in the

low-dimensional representation, apple and pear are expected to be closer (in terms of Euclidean distance of the embedding vector) than apple and dog.

More formally, via truncated SVD, we have factorization

$$M \approx UV^T$$

where $M \in R^{N \times N}$, N is the size of the one-hot vector, $U, V \in \mathbb{R}^{N \times k}, k << N$. Columns of U are the basis vector in latent word space. Each row in V is the low dimensional representation of a word in the latent word space.

10.2 Language modeling

10.2.1 Motivation

Natural languages are communication tools between human beings and have a long history. Natural languages emerge from communication between human beings. Unlike programming languages, which have a strict set of rules to allow a human programmer to communicate instructions precisely to machines, natural languages only have a limited set of formal rules to follow.

Natural languages can be extremely flexible and sometimes ambiguous, yet still be understood by human beings. Consider the following examples [source].

- Sentences consisting of completely different tokens can express the same meaning. For example, *What is your age?* vs. *How old are you?*
- Sentences consisting of similarly tokens can have vastly different meanings? For example *You know him better than I.* vs. *You know him better than me.*
- There could be ambiguities. For example, *I hit the man with a stick* (Used a stick to hit the man) vs. *I hit the man with a stick* (I hit the man who was holding a stick).
- Punctuation can change the meaning significantly. For example, *Woman, without her man, is helpless.* vs. *Woman! Without her, man is helpless!*

Although linguists are directing efforts to modeling languages via grammars, rules, and structure of natural language, we will take alternative computational approaches that focus on learning language models from examples.

In this section, we first consider a classical n-gram statistical language approach, where we count the co-occurrence of words and model the language generation using probabilistic models (e.g., Markov chains). N-gram models focus on the superficial properties as it simply count. It further suffers from the curse of dimensionality since sentences are normally long and language vocabulary size is huge (e.g., 10^6). Over the past two decades, the neural network based language models have attracted considerable attention because these models are able to reveal deep connections between words as well as to alleviate the dimensionality challenge.

A language model can be used directly generate new sequences of text that appear to have come from the corpus. For example, an AI-writer can generate an article with given key words. Moreover, language models are an integral part of many text-related applications, where they ensure output word sequences in these tasks to have a sensible meaning or to appear from human beings. These tasks include

- Optical character Recognition, a task converting images to sentences.

- Machine translation, where one sentence from one language is translated to a sentence in another language.
- Image captioning, where a image is used as the input and the output is a descriptive sentence.
- Text summarization, where a large paragraph or a full article is converted several summarizing sentences.
- Speech recognition, where audio signal is converted words and sentences.

10.2.2 N-gram statistical model

10.2.2.1 *The model*

A statistical model aims to assign probabilities to sequences of words. The probabilistic model then can be used to sentences that are probably from human being by sampling probabilistic distribution. An n-gram statistical model is one fundamental model that estimate probabilities by counting the co-occurrence of n consecutive words. More precisely, an n-gram is a sequence consisting of N words. For example, given a sentence *The homework is due tomorrow*, it has 2-grams (or bigrams) like **the homework, homework is, is due, due tomorrow**; Similarly, it has 3-grams (or trigrams) like *the homework is, homework is due, is due tomorrow*. Counting the n-gram frequencies from a large corpus can provide information of co-occurrence probability and can be used to estimate the probability of sentences.

Formally, we denote a sequence of N words by $w_1 \dots w_n$ or w_1^n. We also use shorthand notation w_1^n to represent n words starting from w_1, i.e., $w_1^n = (w_1, w_2, \dots, w_n)$.

The probability of sequence $w_1, ..., w_n$ have the following decomposition based on conditional probability **chain rule**:

$$\begin{aligned} P\left(w_1^n\right) &= P\left(w_1\right) P\left(w_2|w_1\right) P\left(w_3|w_1^2\right) \dots P\left(w_n|w_1^{n-1}\right) \\ &= \prod_{k=1}^{n} P\left(w_k|w_1^{k-1}\right) \end{aligned}$$

Computing accurate estimations of $P\left(w_k|w_1^{k-1}\right)$ for arbitrary k is impractical for large k. Instead, we can use n-gram statistics to estimate truncated conditional probability. For example, based on bigram statistics, we can estimate $P(w_i|w_{i-1})$ and then further approximate

$$P\left(w_n|w_1^{n-1}\right) \approx P\left(w_n|w_{n-1}\right)$$

and therefore

$$P\left(w_1^n\right) \approx \prod_{k=1}^{n} P\left(w_k|w_{k-1}\right).$$

In practice, we usually go beyond the simple bigram approximation. In general, the N-gram approximation is given by

$$P\left(w_n|w_1^{n-1}\right) \approx P\left(w_n|w_{n-N+1}^{n-1}\right)$$

and

$$P\left(w_1^n\right) \approx \prod_{k=1}^{n} P\left(w_k|w_{k-N+1}^{k-1}\right).$$

Given a large corpus, the n-gram conditional probability used above can be simply obtained by counting.

For example, the bigram probability of a word $w_n = y$ given a previous word $w_{n-1} = x$, is given by

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

where $C(xy)$ is the total count of bigrams xy in the corpus and $w_{n-1}w$ enumerates all distinct bigrams.

10.2.2.2 *Evaluation*

Given a training corpus, we can construct different n-gram language models by varying different n. How can we know one is better than the other? One common way is to take a truly unseen test set and compare the probability of the test set computed from different candidate models. Whichever model assigns a higher probability to the test set is considered a better model, because it has better prediction performance.

Similar to supervised machine learning problems, n-gram language models face bias and variance trade-off given a finite-sized corpus. Taking a large n can overfit to the training corpus, as it tends to memorize specific long sequence patterns in the training corpus and gives lower probability. Using a bigram model can underfit since bigram can hardly capture the sequential relationship among words.

Although we can use raw probability as our evaluation metric, in practice, we use a variant called perplexity. The perplexity (sometimes called PP for short) 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 $W = w_1w_2 \ldots w_N$

$$\begin{aligned}\text{perplexity}(W) &= P\left(w_1 w_2 \dots w_N\right)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P\left(w_1 w_2 \dots w_N\right)}}\end{aligned}$$

where $P\left(w_1 w_2 \dots w_N\right)$ can be estimated in an K-gram model via

$$P\left(w_1^n\right) \approx \prod_{i=1}^{n} P\left(w_i | w_{i-K+1}^{K-1}\right).$$

Intuitively, perplexity is roughly the inverse probability of the test set. Therefore, the smaller the perplexity, the better the evaluation score. The perplexity is always greater than or equal to 1.0.

10.2.3 Neural language modeling

10.2.3.1 *Early developments*

The n-gram statistical language model aims to learn the joint distribution of word sequences. This approach suffers from curse of dimensionality when n becomes large. For example, consider a language with a vocabulary of size $V = 10^6$, a 10-gram model can have model parameters of V^{10}. On the other hand, natural languages tend to have long-range dependency, i.e., the occurrence probability of a word may depend on the existence of an another word multiple steps before. Therefore, a high-quality language model must require large n and, unfortunately, the curse of dimensionality becomes one inherent limitation of n-gram models.

N-gram models also fails to capture the semantic meaning of words.[3] The core of n-gram model is simply a mechanical counter of co-occurrence of words. In natural language, there are many words that are similar in their meaning as wells as their grammar rules. For example, *A cat is walking in the living room* vs. *a dog is running in the bedroom* have similar word pairs (cat, dog), (walking, running), (living room, bedroom) and use similar patterns. These similarities or word semantic meaning can be exploited to construct model with smaller parameters.

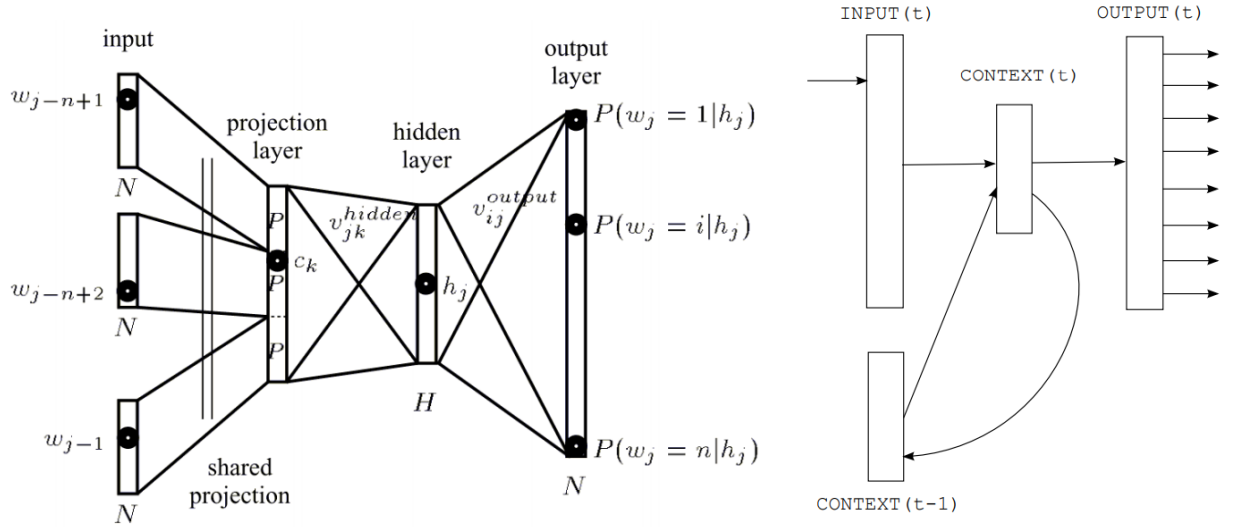
To overcome the limitation of n-gram models, Bengio et al [3] proposed neural language models, which predict and generate next word based on its context and operating at a low dimensional dense vector space (i.e., word embedding). In the early development of neural language model, there are two important attempts. One is feed-forward neural network [3] and one is recurrent neural network[4] [Figure 10.2.1].

The core idea is that by projecting words into low dimensional space (via learning and gradient descent), similar words are automatically clustered together. The curse of dimensionality is naturally addressed because of the efficient parameterization in neural networks.

In feedforward network model [Figure 10.2.1(a)], each word, together with its preceding $N-1$ words as context are projected into low-dimensional space and further predict the next word probability. Note that the context has a fixed length of N, which it is limited in the same way as in N-gram models. In recurrent network model [Figure 10.2.1(b)], context is extended via recurrent connections and context length is theoretically unlimited.

Specially, let the recurrent network input be x, hidden layer output be s and output be y. Input vector $x(t)$ is a concatenation of a word vector $w(t)$ and the previous hidden layer output $s(t-1)$, which represents the context. To summarize, we have

$$
\begin{aligned}
x(t) &= w(t) + s(t-1) \\
s_j(t) &= \text{Sigmoid}\left(W_s x(t) + b_s\right) \\
y_k(t) &= \text{Softmax}\left(W_y s(t) + b_y\right)
\end{aligned}
$$



(a) Feedforward neural netowk based language model.

(b) Recurrent neural network based language model.

Figure 10.2.1: Basic neural network based language models. Image from source.

10.2.4 Character-level language modeling

10.2.4.1 *Word classification*

In sentiment classification, we take a sequence of words as the input and output a sentiment label. In word classification, we take a sequence of characters and output a label characterizing the word.

An example application is to classify names origin via RNN in Figure 10.2.2. The overall architecture contains the following components:

- Characters are fed into the LSTM layer one by one.
- The last output from the LSTM will go to a Softmax output layer to produce classification probabilities on the language origin (Chinese, American, Japanese, etc.).

Note that we usually do not need embedding layer since the one-hot encoding on character level is already of low dimensionality.

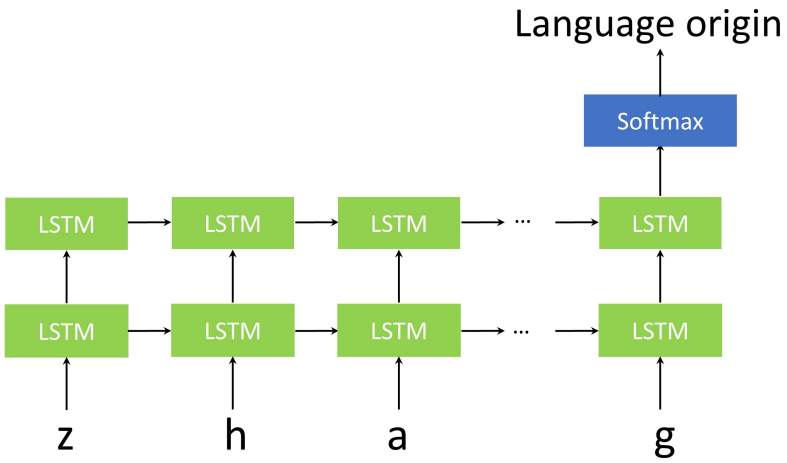


Figure 10.2.2: A RNN for character-level word classification.

10.2.4.2 *Text generation*

RNN has been used for advanced language modeling tasks, like machine translation, caption generation, and chatting robot. Before diving into these advanced applications, we cover one very basic language modeling task in this section, known as character-level language modeling. The basic goal of character-level language modeling is to predict or generate a sequence of sensible characters based on preceding characters.

One concrete application example of character-level language model is when we write an email in Gmail and type messages on a phone, we observe Gmail and our phones will suggest next character/word. Alternatively, a simple word completion program can also be made by n-gram statistical model, which basically counts n-gram frequencies. However, with RNN designed to efficiently capture the long term relation among characters, we expect RNN language modeling can do far more than a empirical n-gram statistical model.

The first step towards the language modeling task is to train a RNN that is able to predict the next character based on an input character sequence [Figure 10.2.3]. A naive n-gram model would required K^n memory, where K is the vocabulary size (say $K = 26$ for alphabetic letters). From this perspective, RNN can be viewed as an efficient parametric n-gram model.

The overall RNN architecture contains the following components:

- Characters from a training text corpus (e.g., a English novel) are fed into the LSTM layer one by one.
- Each output from LSTM will go to a Softmax output layer to produce prediction probabilities on the next letter (26 classes).
- Network parameters are updated by taking gradient descent based on a cross-entropy loss function applied to each prediction.

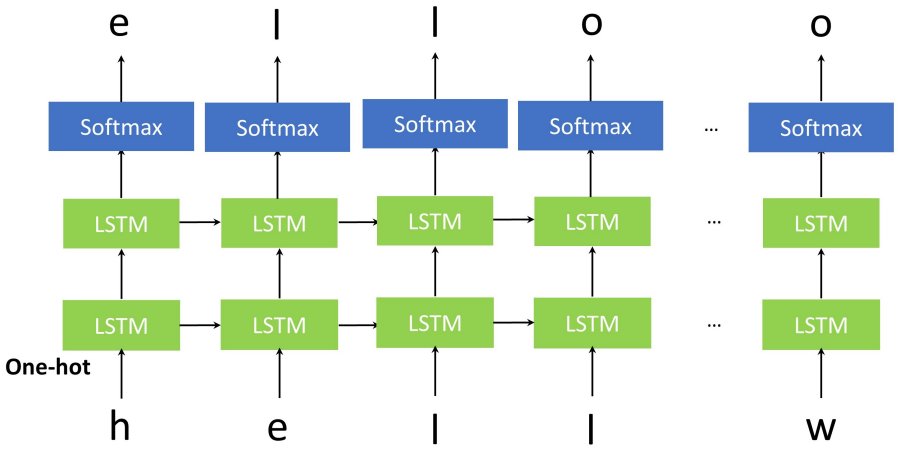


Figure 10.2.3: A RNN for character-level language model. During the training session, one-hot coded characters are directly fed into RNN and predict the next character in the word.

The stage of text generation is to use the trained RNN to sequentially predict new character based on previous predicted character.

The text generation scheme and architecture are showed in Figure 10.2.4. The actual predicted character requires sampling from the prediction probability output by the softmax layer.

There are different sampling strategies from predicted probabilities to promote the total probability of a full generated word. On one extreme, a greedy sampling strategy will take character with the largest probability. But such an approach usually results in repetitive, deterministic patterns that don't look like coherent language. A more general and flexible strategy is to introduce a temperature controlled sampling strategy, where the sampling probability for character i is given by

$$p_i^T \propto \exp(\frac{p_i^S}{T}),$$

where p_i^S is the sampling probability from softmax output. As $T \to \infty$, we get uniform sampling, and as $T \to 0$, we have greedy sampling. Compared to greedy sampling, probabilistic sampling can generate interesting language patterns that resemble human language.

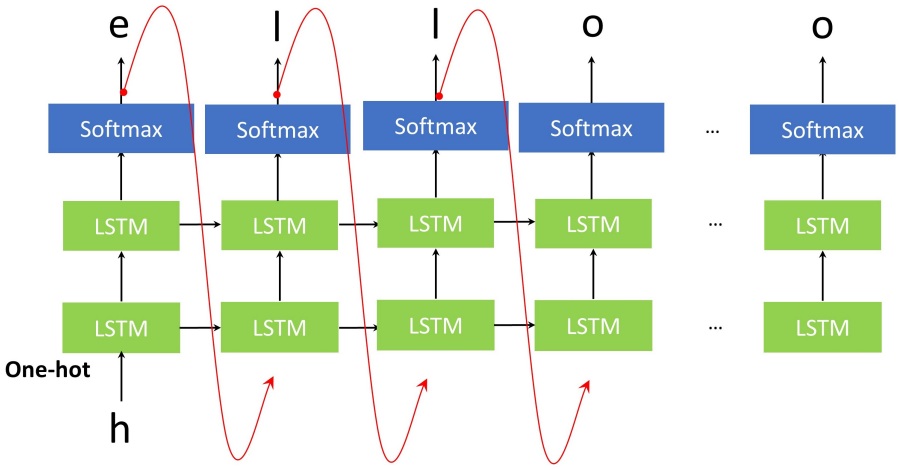


Figure 10.2.4: A RNN for character-level language model. During the word generation session, the network starts with a user input character and continues the generation process with predicted character from the previous step.

10.3 Text classification

10.3.1 Sentiment analysis

We now consider a sentiment analysis task where we are given a movie review text and we need determine if the review's sentiment is positive or negative.

Some negative example reviews are:

- this was an impulse pick up for me from the local video store . don t make the same mistake i did . this movie is **tedious unconvincingly acted** and generally **boring**. the dialogue between the young ...
- yes there are great performances here . unfortunately they happen in the context of a movie that doesn t seem to have a clue what it s doing . during the first minutes of this all the music ta...
- i saw this movie with some indian friends on christmas day . the quick summary of this **movie is must avoid** . jp dutta wrote directed produced and edited this movie and did none of these jobs well.
- this movie is **extremely boring** it tells a story of a female gas station owner and her life.**nothing exciting** ever happens . the director has really kept it real and it feels just like a camera fol...

And some positive example reviews are

- his scary and rather gory adaptation of stephen king s **great** novel features **outstandin**g central performances by dale midkiff fred gwynne who sadly died few years ago and denise crosby and some ...
- okay i ll confess . this is the movie that made me love what michael keaton could do . he does a beautiful parody of someone doing a parody of james cagney with **charm** to spare.
- directors had the same task tell stories of love set in paris . naturally some of them turned out better than others but the whole mosaic is **pretty charming** besides wouldn t it be boring if

Clearly, the existence of words like **tedious, boring** will likely be found in a negative review; similarly, the existence of words like **outstanding, charm** will likely be found in a positive review. Our goal is to define a neural network that can automatically establish such relation.

An initial analysis is to identify these words that are hallmarks of positive and negative reviews. Our method is to first count the word frequency for positive reviews and negative reviews, respectively. The most frequent words in both positive and negative review are actually neural stop words like *the, I,* etc..

Let w_i^P and w_i^N be the frequency of word i shown in positive and negative reviews respectively. We instead of show the words i with highest ratio of w_i^P / w_i^N and lowest ratio of w_i^P / w_i^N with at least 100 counts. They are expected to be most distinguishing words that determine the sentiment of review.

The result in the following table agrees with our expectation that words like *superb* and *worst* are the key words to determine the sentiment.

Positive	**Negative**
superb	worst
wonderful	awful
fantastic	horrible
excellent	crap
amazing	worse
powerful	terrible
favorite	mess
perfect	stupid
brilliant	dull

Clearly, only words with ratio away from 1 are useful features to distinguish the sentiment of a review. The first step of feature engineering is to keep only words that satisfy ratio requirement. These words are then fed to embedding layer and sum up to dense vector, as a bagged feature embedding vector [Figure 10.3.1]. The bagged embedding is then fed into a feed-forward module to predict labels.[1]

1 One the most efficient implementation of this embedding bagging approach is https://github.com/facebookresearch/fastText.

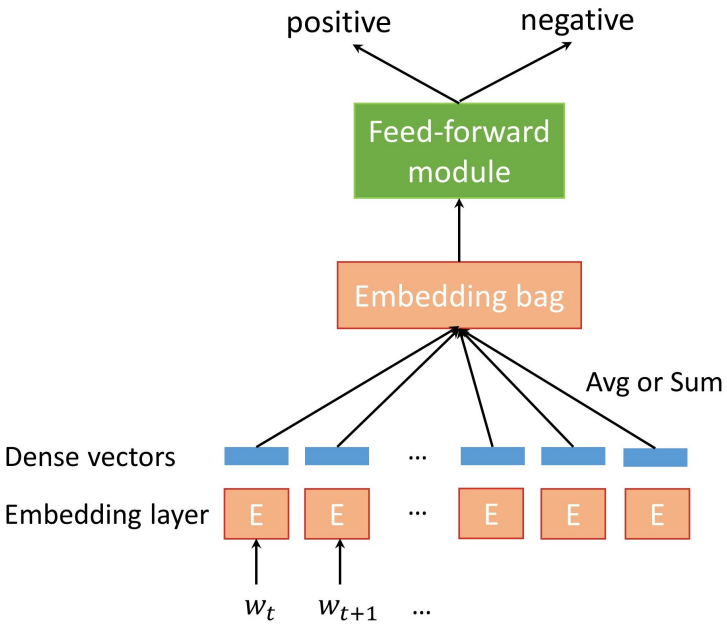


Figure 10.3.1: A feed-forward neural network architecture for sentiment analysis.

However, this embedding bagging approach throws away order information and could run into difficulties when dealing with complex sentences like [5]

- The movie is not very good, but i still like it.
- The movie is very good, but i still do not like it.
- I do not like it, but the movie is still very good.

Such drawback can be alleviated by feeding embeddings into the recurrent neural network.

10.3.2 Sentence classification via CNN

The primary playgrounds of CNNs are image analysis and understanding. The local connectivity in CNN suits spatial locality in images and the multi-layer of filters enable progressive extraction of high-levels features from low-level raw pixel data [subsubsection 8.6.2.1]. The ability of CNN to extract high-feature from local low-level features also sparks novel application in language related application, such as sentence classification[6].

We first look at how CNN filters can act as feature extractor for sentences. Consider a sentence consisting of N words, with each word represented by a k-dimensional vector $x_i \in \mathbb{R}^k$. In general x_i are low-dimensional dense word embeddings [section 10.1]. If we view a sentence as a stack of row vectors of its constituent words, then the representation of a sentence is similar to an image with size $N \times k$. If we apply a convolutional filter

with kernel size $h \times k$, then the output will be a one-dimensional feature vector of length $N - h + 1$ [Figure 10.3.2].

Mathematically, the CNN filter extracts a new feature vector from a word sequence of window h given by

$$c_i = f\left(\sum_{j=1}^{h} W x_{i+j-1} + b\right),$$

where W is the kernel matrix, b is the bias, and f is a non-linear activation function. In models proposed in [6], CNN filters have different window sizes $h = 3, 4, 5$, with 100 for each of them. Intuitively, these CNN filters are extracting features from 3-grams, 4-grams, and 5-grams. All these CNN extracted features are then pooled and combined to feed into feed-forward layers for sentence classification [Figure 10.3.3].

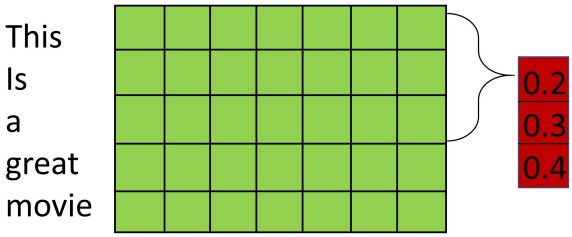


Figure 10.3.2: Demonstration of a CNN filter applying to a sentence to produce a one-dimensional feature vector.

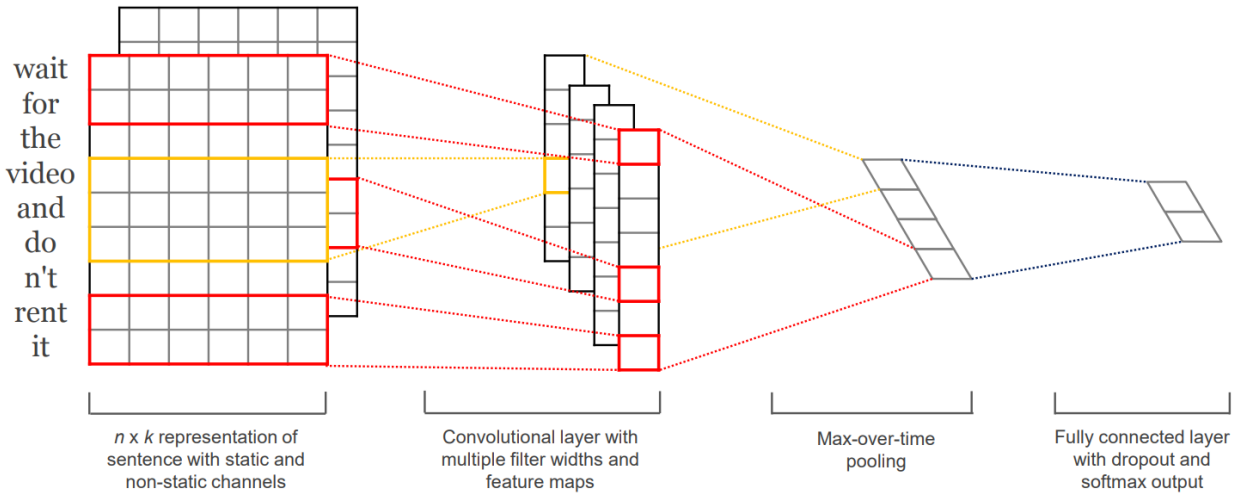


Figure 10.3.3: CNN for sentence classification proposed in [6].

10.4 Sequence-to-sequence modeling

10.4.1 Encoder decoder model

In Sequence-to-sequence (seq2seq) modeling, we are seeking a mapping that takes a sequence $X_1, X_2, ..., X_N$ as the input and outputs another sequence $Y_1, Y_2, ..., Y_M$ that meet certain requirements. In general, X_i and Y_j can be a character, a words, or a real-valued vector, an image, etc, and generally $N \neq M$.

Many fascinating AI real-world applications fall into the seq2seq category. Examples include

- machine translation, where a word sequence in one language is mapped to a word sequence in another language;
- speech recognization, where a time series of audio signal is mapped to a word sequence;
- video captioning, where a video, or a sequence of images, is mapped to a word sequence;
- text summarization, where a long sequence of words is mapped to a short sequence of words (i.e., a summary).

One challenge in a seq2seq modeling problem is that usually it cannot be decomposed to a smaller-scale mapping problem between one sequence unit to another sequence unit. For example, in translation task, one word in one language could be translated to one word, or multiple words, or even not translated depending on the context. Further, usually no synchrony exist between sequence X and sequence Y. Take machine translation for another example. Suppose in a word sequence X_i precedes word X_j, and in the translated word sequence, they also have clear correspondence to Y_k and Y_l, respectively. By no synchrony, we mean Y_k might come after Y_l due to language grammar reasons. Therefore, in the seq2seq modeling, sequences must be processed as a whole unit. Note that such architecture can include multiple hidden layers as well. A example of synchrony issue in machine translation is showed in Figure 10.4.1.

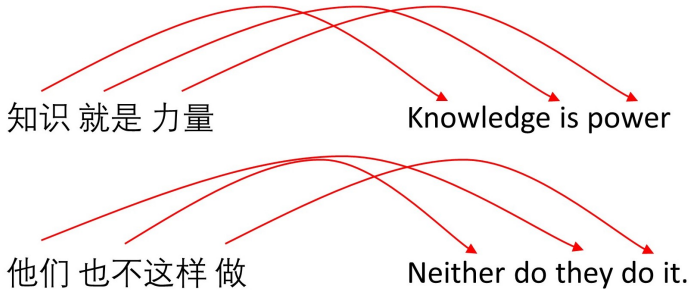


Figure 10.4.1: Seq2seq modeling for language transformation. The input and output sequences might have different lengths, and not in synchrony.

The most basic implementation of a sequence-to-sequence model is via a encoder-decoder network consisting of two RNNs, called encoder and decoder, respectively [Figure 10.4.2]. The encoder-decoder framework stems from efforts in natural language model[7]. The encoder reads a sequence of inputs and outputs the hidden vector at the end of the sequence, known as **the context vector**; the decoder takes the context vector as the initial hidden state input and then sequentially generates a sequence. After training, the context vector is expected to store all relevant, critical information of the input sequence such that the decoder can produce sequences meet certain requirements.

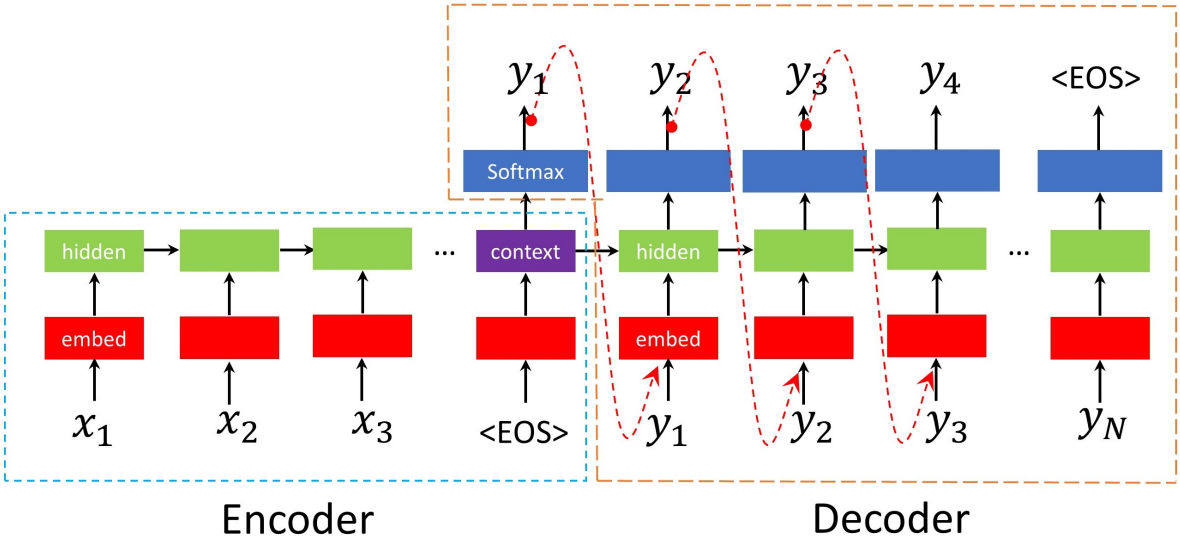


Figure 10.4.2: The Encoder-decoder architecture for seq2seq language modeling. The input sequence is fed into the encoder RNN and terminated by an explicit <EOS> symbol. Then the decoder RNN starts with the context vector and the final prediction of the encoder to generate an output sequence until an explicit <EOS> symbol is produced.

In a video captioning application, this encoder-decoder model will take image sequence as the input, with the usage of convolutional layers replacing embedding layers to extract image feature, and will produce a sequence of words to generate words as captions.

10.4.2 Attention mechanism

The performance of a basic encoder-decoder model [Figure 10.4.2] relies on the **context vector** to store all relevant, critical information of the input sequence. It is not hard to notice that this encoder-decoder network suffers when the input sequences are very long: difficulties arise in both in the encoding part - difficult to encode long sequences in a single vector without losing information, and in the decoding part - difficult to decode a single information vector to a long sequence.

Attention mechanism is inspired how human beings focus on relevant information when performing visual and language tasks[8, 9]. Consider a human translates a sentence. When he translates each word in the sentence, he tends to focus on its related context rather than focusing on the entire sentence. Similarly, when a human is watching closely some part of a object, he tends to focus on the part of interest.

In the context of encoder-decoder, all hidden states carry information and some information would get diluted in the subsequent context vector, especially when the sequence is long. A encoder-decoder architecture with attention mechanism have following features to resolve the difficulties of encoding and decoding [Figure 10.4.3]:

- During the encoding phase, all hidden states, rather than the final one, are saved to construct different context vectors via linear combination for the decoding stage.
- During the decoding phase, relevant context vectors are constructed and fed into the each hidden states in the decoder, alleviating the difficulty to decode all information from one starting context vector.

In summary, the encoder encodes a library of hidden vectors as the building blocks of different context vectors for each output in the decoding stage. The attention mechanism resembles a information lookup system that constructs and feeds the most important and relevant context vector to help the decoder part to do their job. From information flow point of view, the primary purpose attention is to calculate a weight vector to amplify signals that are most relevant in the input sequence and to de-emphasize the part that is irrelevant. The higher weights will enable more relevant hidden state information fed into the the decoder hidden layer.

Now we consider a more concrete implementation of attention mechanism from [8]. Let $(x_1, x_2, ..., x_m)$ and $(y_1, y_2, ..., y_n)$ denote the input and output sequences. The encoder uses bidirectional RNN to encode input sequences to a library of hidden states. Particularly, let $\overrightarrow{h}_i$ denote the hidden state in the forward pass and $\overleftarrow{h}_i$ the hidden state in the backward pass. A hidden state is the concatenation $h_i = cat(\overrightarrow{h}_i, \overleftarrow{h}_i)$. In the decoder part, let s_t be the hidden state information that just precedes the output y_t. s_t is given by

$$s_t = f(s_{t-1}, y_{t-1}, c_i),$$

where c_i is the context vector constructed as a weighted combination of hidden information $\sum_{i=1}^{M} w_i(t) h_i$. Note that the weights is index-dependent, derived from

$$\begin{aligned} e_i(t) &= g(h_i, s_{t-1}) \\ w_i(t) &= \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}. \end{aligned}$$

The primary purpose of g is to capture the relevance or correlation between the decoder hidden state and the encoder hidden state. Typical choices for g functions include

$$\begin{aligned} g\left(h_i, s_{t-1}\right) &= h_i^T s_{t-1} \\ g\left(h_i, s_{t-1}\right) &= h_i^T W_g s_{t-1} \\ g\left(h_i, s_{t-1}\right) &= v_g^T \tanh\left(W_g [h_i; s_{t-1}]\right) \end{aligned}$$

where W_g and v_g are learnable parameters.

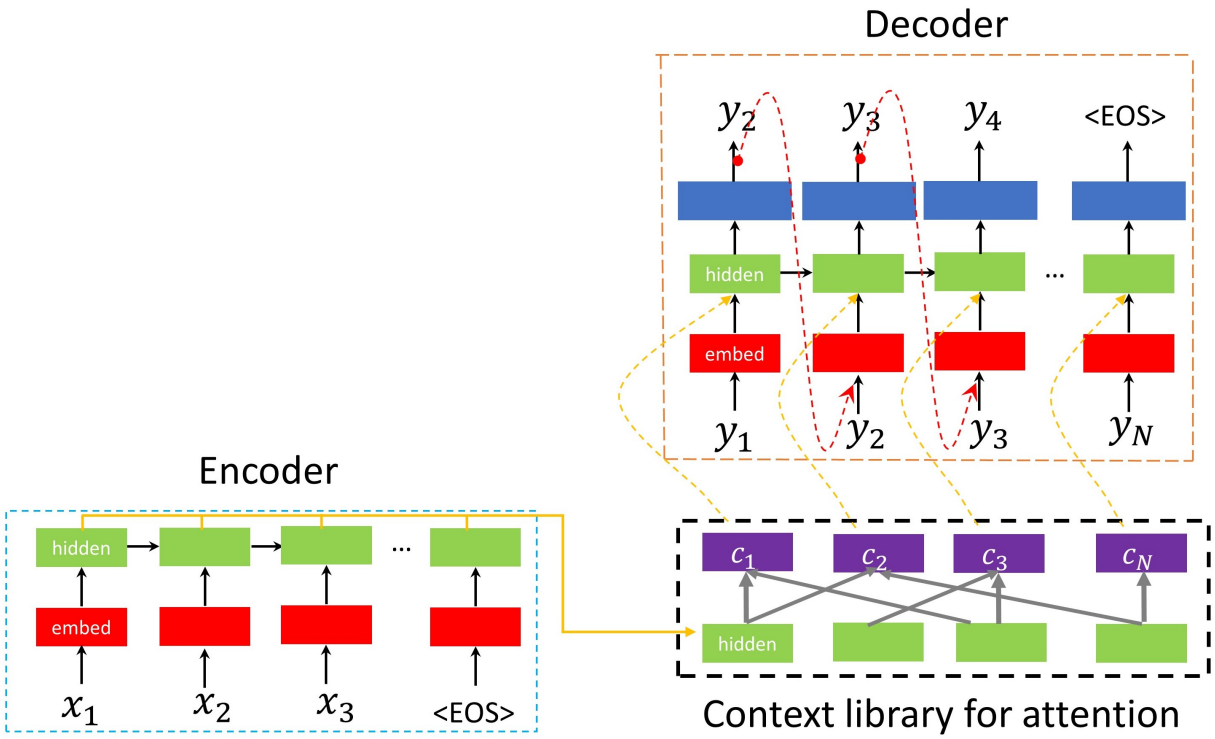


Figure 10.4.3: The Encoder-decoder architecture with attention mechanism for seq2seq modeling. During the encoding phase, all hidden states, rather than the final one, are saved to construct different context vectors via linear combination for the decoding stage. During the decoding phase, relevant context vectors are constructed and fed into the each hidden states in the decoder.

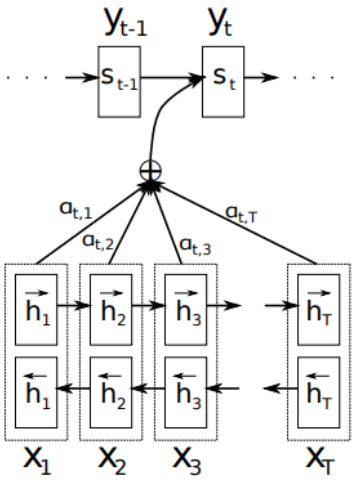


Figure 10.4.4: A bidirectional RNN encoder system with attention mechanism. [8]

10.4.3 Google's Neural Machine Translation System

A production level application of the attention-based seq2seq framework is the Google's Neural Machine Translation System [10].

Our model consists of a deep LSTM network with 8 encoder and 8 decoder layers using residual connections as well as attention connections from the decoder network to the encoder. Simply stacking more LSTM layers will lead to performance degradation, probably due to exploding and vanishing gradient in the vertical direction. Residual connection can greatly improve the gradient flow in the backpropogation process. As translation usually requires information across the whole source side, bi-directional connection is employed. However, to enable maximum possible parallelization, bi-directional connections are only used in the first layer in the encoder.

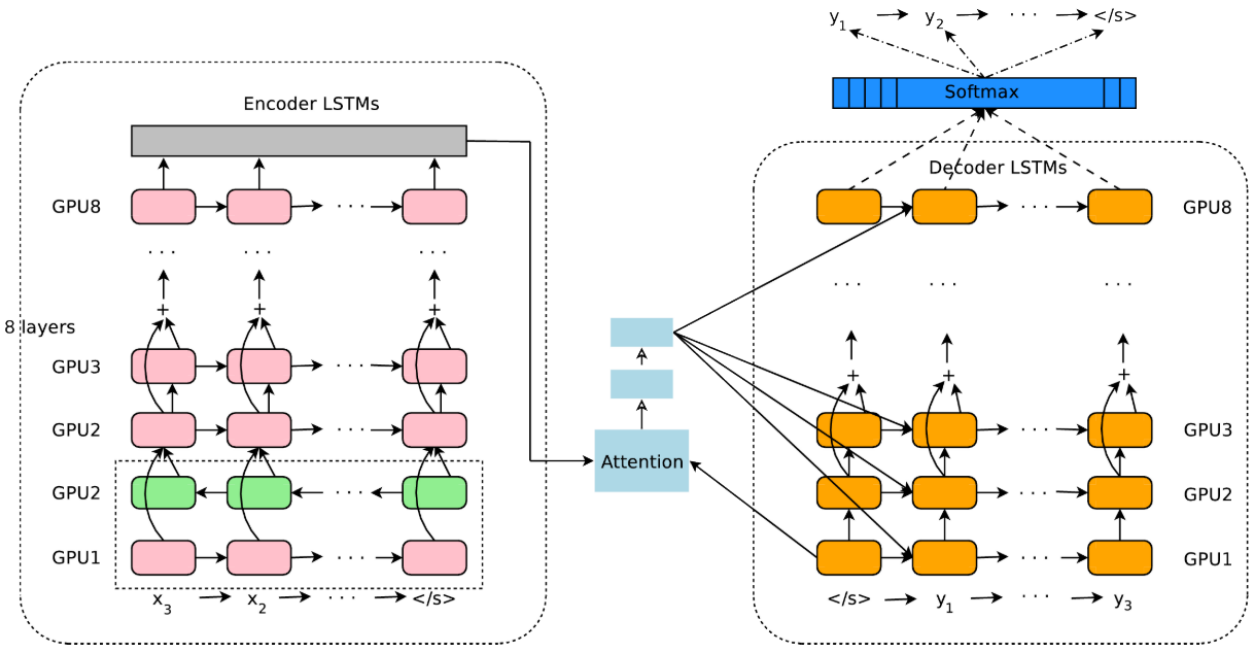
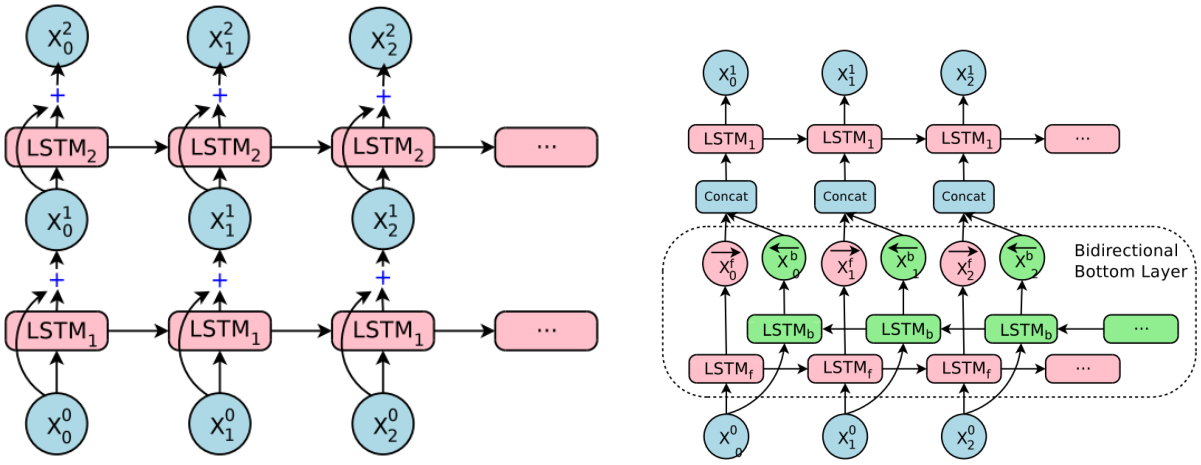


Figure 10.4.5: The model architecture of Google's Neural Machine Translation system, which consists of an encoder module (left), an attention module (middle), a decoder module (right). The bottom encoder layer is bi-directional and other layers are uni-directional. Residual connections are used from the third layer in the encoder and in all layers in the decoder. There are total 8 LSTM layers in both encoder and decoder and the model is partitioned into multiple GPUs to speed up training. Image from [10].



(a) stacked LSTM layers with residual connections.

(b) The structure of bi-directional connections in the first layer of the encoder.

Figure 10.4.6: The residual connection and bi-directional LSTM structure in the model.Image from [10].

Now we look into the mathematical detail. Let (X, Y) be a source and target sentence pair. Let $X = (x_1, x_2, x_3, \ldots, x_M)$ be the sequence of M tokens in the source sentence and let $Y = y_1, y_2, y_3, \ldots, y_N$ be the sequence of N tokens in the target sentence. The encoder module transforms the M input tokens into M hidden states $(\boldsymbol{x_1}, ..., \boldsymbol{x_M})$:

$$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M = \text{Encoder}\left(x_1, x_2, x_3, \ldots, x_M\right).$$

With the residual connection, the update of hidden states $\boldsymbol{h}$ and cell state $\boldsymbol{c}$ have the following update procedures throughout the stacked LSTM encoder (from layer i to layer $i+1$).

$$\begin{aligned}
\mathbf{c}_t^i, \mathbf{h}_t^i &= \text{LSTM}_i(\mathbf{c}_{t-1}^i, \mathbf{h}_{t-1}^i, \mathbf{x}_t^{i-1}) \\
\mathbf{x}_t^i &= \mathbf{h}_t^i + \mathbf{x}_t^{i-1} \\
\mathbf{c}_t^{i+1}, \mathbf{h}_t^{i+1} &= \text{LSTM}_{i+1}(\mathbf{c}_{t-1}^{i+1}, \mathbf{h}_{t-1}^{i+1}, \mathbf{x}_t^i)
\end{aligned}$$

The computation of the context vector $\boldsymbol{a}_i$ for ith step in the decoder is given by the following step

$$\begin{aligned}
s_t &= \text{Attention}\left(\mathbf{y}_{i-1}, \mathbf{x}_t\right) \quad \forall t, \quad 1 \leq t \leq M \\
w_t &= \frac{\exp(s_t)}{\sum_{t=1}^{M} \exp(s_t)} \quad \forall t, \quad 1 \leq t \leq M \\
\mathbf{a}_i &= \sum_{t=1}^{M} w_t \cdot \mathbf{x}_t
\end{aligned}$$

where the Attention function is implemented as a feed-forward neural network with one hidden layer.

As the softmax layer in the decoder output the token probability based on previous inputs, the whole model approximates the conditional probability in the following way

$$\begin{aligned} P(Y|X) &= P\left(Y|\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_{\mathrm{M}}\right) \\ &= \prod_{i=1}^{N} P\left(y_i|y_0, y_1, y_2, \ldots, y_{i-1}; \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_{\mathbf{M}}\right) \end{aligned}$$

The optimization on model parameter θ is achieved by performing gradient descent to maximizes a mixed likelihood, one is ordinary likelihood and one is reward-weighted likelihood

$$L_{\text{Mixed}}(\theta) = \alpha L_{\text{ML}}(\theta) + L_{\text{RL}}(\theta)$$

where α is a scalar, Y^* is the ground truth output sequence and

$$L_{\text{ML}}(\theta) = \sum_{i=1}^{N} \log P_\theta\left(Y^{*(i)}|X^{(i)}\right)$$

and

$$L_{\text{RL}}(\theta) = \sum_{i=1}^{N} \sum_{Y \in \mathcal{Y}} P_\theta\left(Y|X^{(i)}\right) r\left(Y, Y^{*(i)}\right).$$

The reward is related to evaluation score in the translation task.

10.5 Transformers

10.5.1 Self-attention and the Transformer

One common theme for many deep learning architectures is to learn the complex relationship among features.

Fully-connected feed-forward neural network have connections between all input features and each hidden units. By increasing the number of hidden units or number of hidden layers, feed-forward neural networks can express arbitrarily complex continuous functions [Theorem 8.1.1]. However, feed-forward neural network cannot handle input of varying sizes.

CNN can be viewed as a special type of feed-forward neural network with connections among spatially close units. By enlarging kernel size and increasing the number of layers, CNN can also be used to capture relationship among spatially distant feature.

RNN capture relationship among input features by its recurrent structure. Although vanilla RNN units experience gradient vanishing or explosion issues for long input sequences. The LSTM and GRU variants have been proposed to address these issues by extra gate control mechanisms.

Self-attention, also known as **intra-attention**, is a recent powerful techniques that enables learning the representation of a varying length of input dynamically dependent on context.

In feed-forward neural network, the learned representation of an input sequence is statically dependent on its context since the weight associated with the connections are fixed.

Self-attention mechanism can be viewed as a feed-forward neural network with dynamic connections, whose connectivity between input units to hidden units are depending on the input itself.

$$Q = W_Q X \in \mathbb{R}^{D_Q \times N}$$
$$K = W_K X \in \mathbb{R}^{d_K \times N}$$
$$V = W_V X \in \mathbb{R}^{d_V \times N}$$

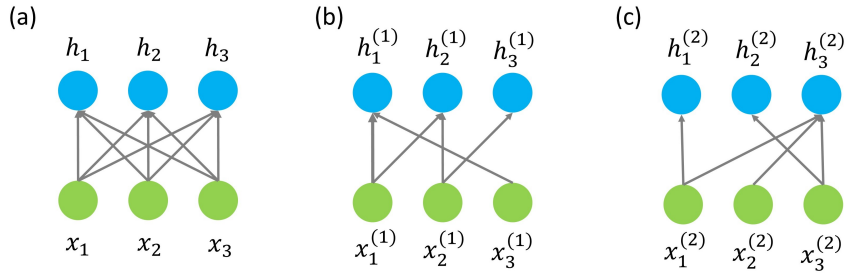


Figure 10.5.1: Intuition of self-attention mechanism in learning representations $h_1, h_2, h_3 \in \mathbb{R}^H$ for a sequence $x_1, x_2, x_3 \in \mathbb{R}^D$ (a) A full-connected layer without attention mechanism. (b, c) Self-attention mechanism intuitively enables connectivity between input sequences and hidden sequences to be dynamically depend on the input sequence itself.

$$
\begin{aligned}
\boldsymbol{h}_i &= \text{att}\left((K, V), \boldsymbol{q}_i\right) \\
&= \sum_{j=1}^{N} \alpha_{ij} \boldsymbol{v}_j \\
&= \sum_{j=1}^{N} \text{softmax}\left(s\left(\boldsymbol{k}_j, \boldsymbol{q}_i\right)\right) \boldsymbol{v}_j
\end{aligned}
$$

$$
H = V \,\text{softmax}\left(\frac{K^{\mathrm{T}} Q}{\sqrt{d_3}}\right)
$$

If we take Q, K to be zero such that all keys and queries are the same zero. Then the output will be $H = V = W_V X$, which reduces to feed-forward neural network with static connection.

Attention-based encoder-decoder model [Figure 10.4.3] relieve the information bottleneck issue in the original encoder-decoder model [Figure 10.4.2]. Recently, a revolutionary model, known as transformer model [11], made further improvements on seq2seq. First, transformer model does not rely on a RNN structure, which open opportunities for parallel computing. Second, transformer employs self-attention mechanism. Classical attention mechanism captures the relationship between input sequence and output sequence, but not relationship among input sequence or output sequence; self-attention mechanism offers a route to capture relationship among input and output sequence.

Since the advent of the first transformer [11], many variants have been developed by tech giants. Those high performance transformer models include BERT[12], GPT[13], GPT-2 [14], Transformer-XL [15], XLNet [16], and XLM [17].

10.5.2 Vanilla transformer

10.5.2.1 *Overall structure*

Transformer [11] is one of most successful architectures in tackling challenging seq2seq NLP tasks like translation, question-answering, etc.

Traditionally, seq2seq tasks heavily use RNN based encoder-decoder architectures, plus attention mechanisms, to transform one sequence into another one. Transformer, on the other hand, does not rely on any recurrent structure and is able to process all tokens in a sequence at the same time.

On a high level, transformer resembles the encoder-decoder architecture [Figure 10.5.2], where the encoder converts an input sequences into low-dimensional embeddings and the decoder outputs an output sequence probabilities. The decoder takes input sequence embeddings and previous output token as the input. The sequential information, original stored in recurrent network structure, is now stored in position encoding added at the entry point of the encoder and decoder modules.

Attention mechanisms are heavily used in the transformer architecture to learn contextualized embeddings and overcome the limitation of recurrent neural network in learning long-term dependencies (e.g., seq2seq model with attention [subsection 10.4.2]).

The encoder module on the left [Figure 10.5.2] consists of blocks that are stacked on top of each other to obtain the embeddings that retain rich information in the input. Multi-head self-attentions are used in each block to enable the extraction of contextual information into the final embedding. Similarly, the decoder module on the right also consists of blocks that are stacked on top of each other to obtain the embeddings. Two types of multi-head attentions are used in each block, one is self-attention to capture contextual information among output sequence and one is to encoder-decoder attention to capture the dynamic information between input and output sequence.

In the following, we will present detailed analysis on each component of transformer architecture.

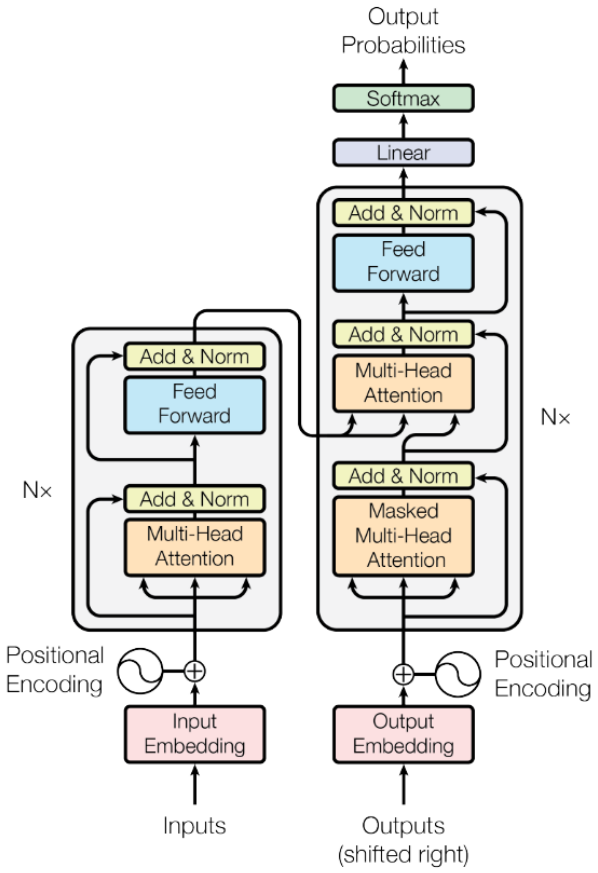


Figure 10.5.2: The transformer architecture [11].

10.5.2.2 *Input output conventions*

In the transformer architecture, we need to deal with different types of sequences

- Input sequence $s = (i_1, i_2, ..., i_p, ..., i_n), i_p \in \mathbb{N}$ and input position sequence $s^p = (1, 2, ..., n)$.
- Output sequence $o = (o_1, o_2, ..., o_p, ..., o_m), o_p \in \mathbb{N}$ and output position sequence $o^p = (1, 2, ..., m)$. Output sequence is the input to the decoder.
- Target sequence $t = (t_1, t_2, ..., t_p, ..., t_m), t_p \in \mathbb{N}$. Input sequence and target sequence form a pair in the training examples.

For example, consider a translational task with input and target sequence given by 'Ich möchte eine Flasche Wasser' and 'I want a bottle of water'. In the training, we can get a typical input sequence, target sequence, and output sequence in the following form

- $s = (Ich, mchte, eine, Flasche, Wasser, PAD, PAD)$
- $t = (I, want, a, bottle, of, water, EOS, PAD)$

- $o = (SOS, I, want, a, bottle, of, water, EOS)$

The output sequence is the right-shifted target sequence with a starting token. Because sequence data is fed into transformer via mini-batches, all input sequence in the same mini-batch will be pad to the maximum length of its batch.

10.5.2.3 *Multihead attention with marks*

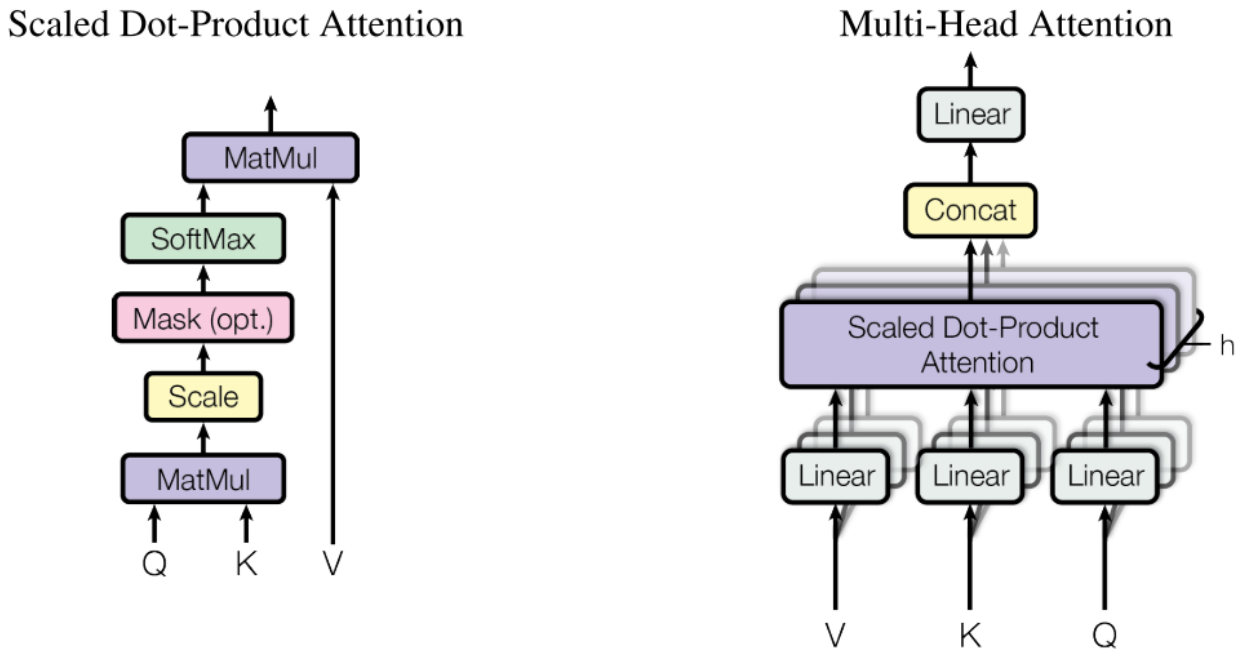


Figure 10.5.3: The multi-head attention architecture [11].

Multi-head attention mechanism plays important role in both encoder and decoder side; particularly, they are used in three places:

- In the encoder module, Multi-head attention is used to construct embedding for each token that depends of its context (i.e., self-attention of the input sequence).
- In the decoder module, Multi-head attention is used to construct embedding for each token that depends of its *seen* output context (i.e., masked self-attention of the output sequence).
- From the encoder module to the decoder module, Multi-head attention is used to construct embedding for each output token that depends of its *input* context (i.e., attention between input and output sequences).

Given a query matrix $Q \in \mathbb{R}^{n \times d_{model}}$, a key matrix $K \in \mathbb{R}^{m \times d_{model}}$, and a value matrix $V \in \mathbb{R}^{m \times d_{model}}$, the multi-head ($h$ heads) attention associated with (Q, K, V) is given by

$$\text{MultiHeadAttention}\,(Q, K, V) = Concat\,(head_1, \cdots, head_h)\, W^O$$

where $head_i \in \mathbb{R}^{n \times d_v}$

$$head_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

with $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}, W^O \in \mathbb{R}^{h \times d_v \times d_{model}}$. In general, we require $d_k = d_v = d_{model}/h$ such that the output of $\text{MultiHeadAttention}\,(Q, K, V)$ has the dimensionality of $n \times d_{model}$.

The attention among (Q, K, V) is given by

$$\text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) = \text{softmax}\left(\frac{QW_i^Q\left(KW_i^K\right)^T}{\sqrt{d_k}}\right) VW_i^V.$$

Note that the softmax applies to each row such that the $\text{softmax}(\cdot)$ produces a weight matrix $w^{att} \in \mathbb{R}^{n \times m}$, with each row summing up to unit 1.

Explicitly, we have

$$\begin{bmatrix} w_{11}^{att} & w_{12}^{att} & \cdots & w_{1m}^{att} \\ w_{21}^{att} & w_{22}^{att} & \cdots & w_{2m}^{att} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^{att} & w_{n2}^{att} & \cdots & w_{nm}^{att} \end{bmatrix} \begin{bmatrix} [VW^V]_1 \\ [VW^V]_2 \\ \vdots \\ [VW^V]_m \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^m w_{1j}^{att}[VW^V]_j \\ \sum_{j=1}^m w_{2j}^{att}[VW^V]_j \\ \vdots \\ \sum_{j=1}^m w_{nj}^{att}[VW^V]_j \end{bmatrix}$$

where $[VW^V]_i$ is the ith row vector of value matrix VW^V.

Sometimes, for each query associated with a symbol, we like to only allow a subset of keys and values to be queried via a binary **mask** $mask \in \{0, 1\}^m$, where 1 indicates exclusion of the key. We can set the values in the intermediate matrix $QW_i^Q(KW_i^K)^T$ to be negative infinity if this column index is associated with mask value 1.

Remark 10.5.1. On the scalability of Self-Attention At this point, it is apparent that the memory and computational complexity required to compute the attention matrix is quadratic in the input sequence length, i.e., $N \times N$. In particular, the $QK^\top$ matrix multiplication operation alone consumes N^2 time and memory. This restricts the overall utility of self-attentive models in applications which demand the processing of long sequences. In subsequent sections, we discuss methods that reduce the cost of self-attention.

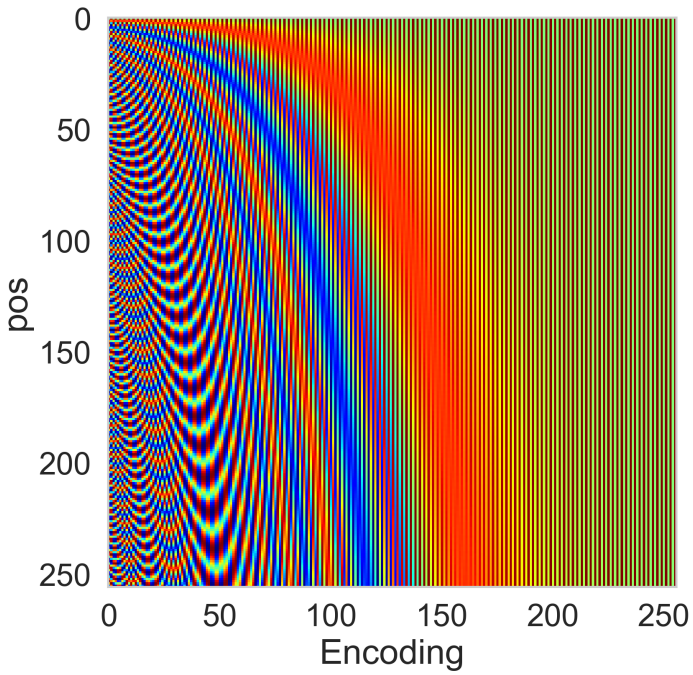


Figure 10.5.4

10.5.2.4 *Position encoding*

For the input sequence s, we first project s into a dense word embedding space via $WE(s) \in \mathbb{R}^{n \times d_{model}}$. The word embedding $WE(s)$ does not encode positional information in the sequence, so we utilize a position encoding PE maps an integer index representing the position of the token in the sequence, $s^p = (1, 2, ..., n)$ to n dense vector with same dimension d_{model}, i.e., $\text{PE}(s^p) \in \mathbb{R}^{n \times d_{model}}$. Notably, given the token position $i \in \{1, ..., n\}$ $\text{PE}(i) \in \mathbb{R}^{d_{model}}$ is given by

$$[\text{PE}(i)]_j = \begin{cases} \sin\left(\frac{i}{10000^{2j/d_{model}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{2j-1/d_{model}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

Intuitively, each dimension of the positional encoding corresponds to a sine wave of different wavelengths ranging from 2π to $10000 \cdot 2\pi$. The position encoding can take other function forms as well. In theory, the purpose of the position encoding is to preserve position information by mapping different position to different values. The position encoding mapping can also be learned from data.

10.5.2.5 *Encoder anatomy*

Given an input sequence represented by integer sequence $s = (i_1, ..., i_p, ..., i_n), i_p \in \mathbb{N}$, *e.g.*, $s = (3, 5, 30, 2, ..., 21)$, the goal of the encoder module [Figure 10.5.2] is to map s to n dense vectors that captures semantic, position, and contextual information. For the input sequence s, we first project s into a dense word embedding space via $WE(s) \in \mathbb{R}^{n \times d_{model}}$. The word embedding $WE(s)$ does not encode positional information in the sequence, so we utilize a position encoding PE maps an integer index representing the position of the token in the sequence, $s^p = (1, 2, ..., n)$ to n dense vector with same dimension d_{model}, i.e., $\text{PE}(s^p) \in \mathbb{R}^{n \times d_{model}}$. Notably, given the token position $i \in \{1, ..., n\}$ $\text{PE}(i) \in \mathbb{R}^{d_{model}}$ is given by

$$[\text{PE}(i)]_j = \begin{cases} \sin\left(\frac{i}{10000^{2j/d_{model}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{2j-1/d_{model}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

Intuitively, each dimension of the positional encoding corresponds to a sine wave of different wavelengths ranging from 2π to $10000 \cdot 2\pi$. The position encoding can take other function forms as well. In theory, the purpose of the position encoding is to preserve position information by mapping different position to different values. The position encoding mapping can also be learned from data.

The whole computation in the encoder module can be summarized in the following.

Definition 10.5.1 (computation in encoder module). *Given an input sequence represented by integer sequence $s = (i_1, ..., i_p, ..., i_n)$ and its position $s^p = (1, ..., p, ..., n)$. The encoder module takes s, s^p as inputs and produce $e_N \in \mathbb{R}^{n \times d_{model}}$.*

$$\begin{aligned} e_0 &= \text{WE}(s) + \text{PE}(s^p) \\ e_1 &= \text{EncoderLayer}(e_0) \\ e_2 &= \text{EncoderLayer}(e_1) \\ &\cdots \\ e_N &= \text{EncoderLayer}(e_{N-1}) \end{aligned}$$

where $e_i \in \mathbb{R}^{n \times d_{model}}$, $\text{EncoderLalyer} : R^{n \times d_{model}} \to R^{n \times d_{model}}$ is an encoder sub-unit, N is the number of encoder sub-units. Specifically, this encoder sub-unit can be decomposed into following calculation procedures

$$\begin{aligned} e_{mid} &= \text{LayerNorm}(e_{in} + \text{MultiHeadAttention}(e_{in}, e_{in}, e_{in}, padMask)) \\ e_{out} &= \text{LayerNorm}(e_{mid} + \text{FFN}(e_{mid})) \end{aligned}$$

where $e_{mid}, e_{out} \in \mathbb{R}^{n \times d_{model}}$,

$$FFN(e_{mid}) = \max(0, e_{mid}W_1 + b_1)W_2 + b_2,$$

with $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}, W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}, b_1 \in \mathbb{R}^{d_{ff}}, b_2 \in \mathbb{R}^{d_{model}}$, *and the padMask excludes padding symbols in s.*

10.5.2.6 *Decoder anatomy*

In the decoder side, we are similarly given an output sequence represented by integer sequence $o = (o_1, ..., o_p, ..., o_m), o_p \in \mathbb{N}$, *e.g.*, $o = (5, 10, 30, 2, ..., 21)$. The decoder module [Figure 10.5.2] aims to converts an output sequence o, combining with resulting embedding e_N in the encoder module [Definition 10.5.1] to its corresponding probabilities over the vocabulary. The output probability can be used to compute categorical loss that drives the learning process of the encoder and the decoder.

The whole computation in the encoder module can be summarized in the following.

Definition 10.5.2 (computation in decoder module). *Given an input sequence represented by integer sequence* $o = (o_1, ..., o_p, ..., o_n)$ *and its position* $o^p = (1, ..., p, ..., m)$. *The encoder module takes* o, o^p *as inputs, combines resulting embedding* e_N *from the encoder, and produce* $d_N \in \mathbb{R}^{n \times d_{model}}$ *and probabilities over the vocabulary.*

$$\begin{aligned} d_0 &= \text{WE}(o) + \text{PE}(o^p) \\ d_1 &= \text{DecoderLayer}(d_0, e_N) \\ d_2 &= \text{DecoderLayer}(d_1, e_N) \\ &\dots \\ d_N &= \text{DecoderLayer}(d_{N-1}, e_N) \\ output\ prob &= \text{Softmax}(d_N W) \end{aligned}$$

where $d_0 \in \mathbb{R}^{m \times d_{model}}$, $\text{DecoderLayer} : R^{m \times d_{model}} \to R^{m \times d_{model}}$ *is a decoder sub-unit, N is the number of decoder sub-units,* $W \in \mathbb{R}^{d_{model} \times |V|^O}$. *Specifically, this decoder sub-unit can be decomposed into following calculation procedures*

$$\begin{aligned} d_{mid1} &= \text{LayerNorm}(d_{in} + \text{MaskedMultiHeadAttention}(d_{in}, d_{in}, d_{in}, padMask)) \\ d_{mid2} &= \text{LayerNorm}(d_{mid1} + \text{MaskedMultiHeadAttention}(d_{mid1}, e_N, e_N, padMask|seqMask)) \\ d_{out} &= \text{LayerNorm}(d_{mid2} + \text{FFN}(d_{mid2})) \end{aligned}$$

where $d_{mid1}, d_{mid2}, d_{out} \in \mathbb{R}^{m \times d_{model}}$,

$$FFN(d_{out}) = \max(0, d_{mid} W_1 + b_1) W_2 + b_2,$$

with $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}, W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}, b_1 \in \mathbb{R}^{d_{ff}}, b_2 \in \mathbb{R}^{d_{model}}$.

Note that there are two type of attention in the decoder module, one is self-attention among the output sequence itself and one is attention between encoder output and decoder output, i.e., **encoder-decoder attention**. The encoder-decoder attention uses a mask that excludes padding symbol in the input sequence. The queries come from the output of previous sub-unit and keys and values come from the final output of the encoder module. This allows decoder sub-units to attend to all positions in the input sequence.

The decoder self-attention uses a mask that excludes padding symbol and future symbol, which can be computed via logical OR between *padMask* and *seqMask*. *seqMask* for a symbol at position i is a binary vector $seqMask_i \in \mathbb{R}^m$ whose value is 1 at position equal or greater than i. Therefore, for a sequence of length m, the complete *seqMask* would be a upper triangle matrix value 1.

10.5.3 Transformer-XL

10.5.4 Reformer

10.6 Pretrained language models

10.6.1 Introduction

In natural language processing, although we have seen many successful end-to-end systems , they usually require large scale training examples and the systems require a complete retrain for a different task. Alternatively, we can first learn a good representation of word sequences that not task-specific but would be likely to facilitate downstream specific tasks. Learning a good representation in prior from broadly available unlabeled data also resembles how human perform various intelligent tasks. In the context of natural languages, a good representation should capture the implicit linguistic rules, semantic meaning, syntactic structures, and even basic knowledge implied by the text data.

With a good representation, the downstream tasks can be significantly sped up by fining training the system on top of the representation. Therefore, the process of a learning a good representation from unlabeled data is also known as pre-training a language model.

Pre-training language models have several advantages: first, it enable the learning of universal language representations that suit different downstream tasks; second, it usually gives better performance in the downstream tasks after fine-tuning on a target task; finally, we can also interpret pre-training as a way of regularization to avoid overfitting on small data set.

In section 10.1, we discussed different approaches (e.g., Word2Vec, GloVe) to learning a low-dimensional dense vector representation of word tokens. One significant drawback of these representations is context-independent or context-free static embedding, meaning the embedding of a word token is fixed no matter the context it is in. By contrast, in natural language, the meaning of a word is usually context-dependent. For example, in sentences *I like to have an apple since I am thirty* vs. *I like to have an Apple to watch fun movies*, the word *apple* mean the fruit apple and the electronic device, respectively.

There has been significant efforts directed to learning contextual embedding of word sequences. A contextual embedding encoder usually operates at the sequence level. As shown in Figure 10.6.2, given a non-contextual word embedding sequence $x_1, x_2, \cdots, x_T$, the contextual embeddings of the whole sequence are obtained simultaneously via

$$[\mathbf{h}_1, \mathbf{h}_2, \cdots, \mathbf{h}_T] = f_{\text{enc}}(x_1, x_2, \cdots, x_T)$$

where $f_{\text{enc}}(\cdot)$ is neural contextual embedding encoder.

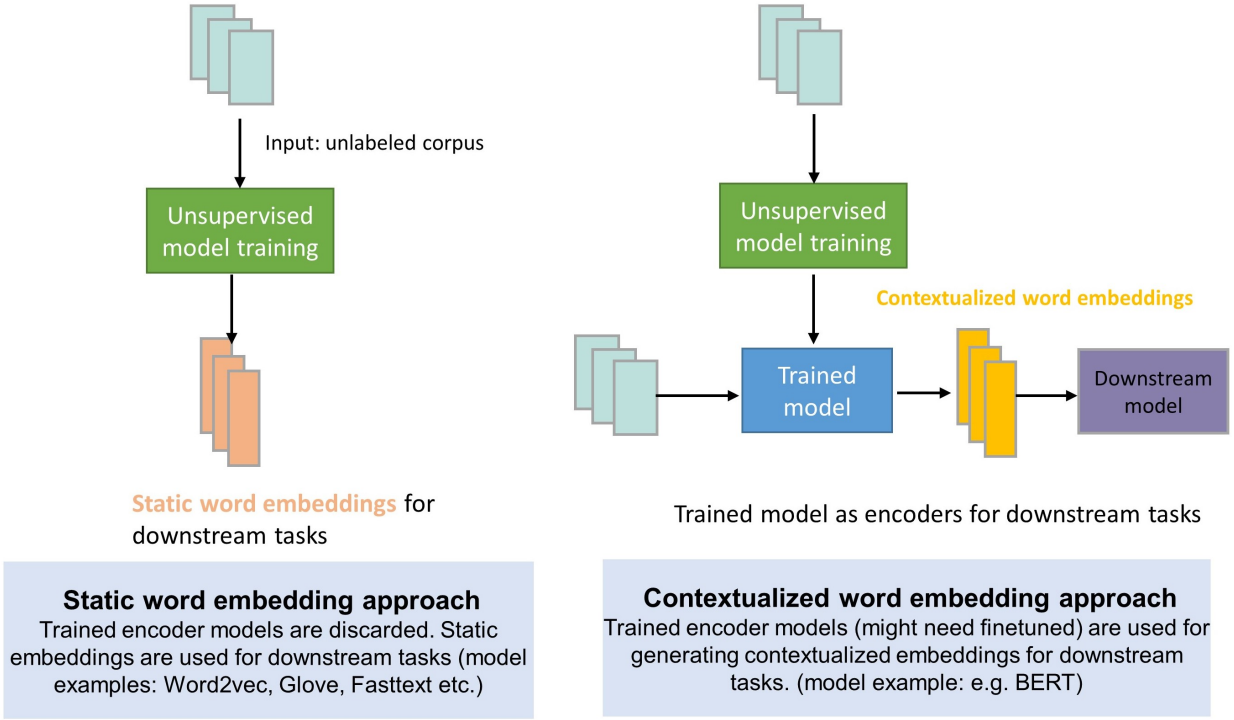


Figure 10.6.1: Static word embedding approach vs. contextualized word embedding approach.

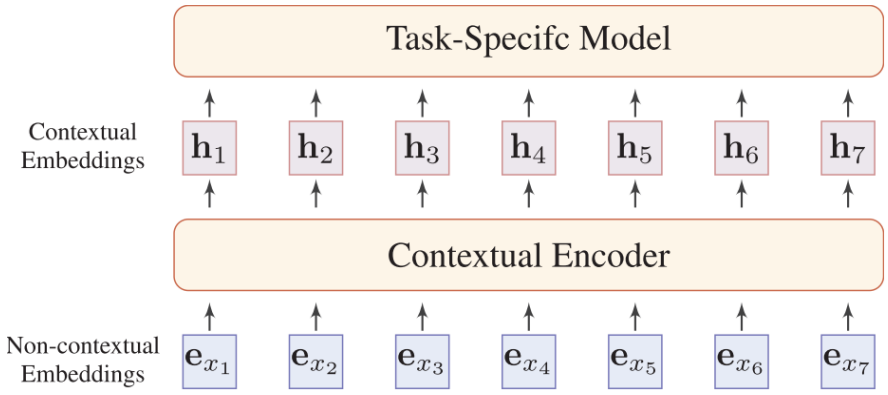


Figure 10.6.2: A generic neural contextual embedding encoder. Image from [18].

Given large-scale unlabeled data, the most common way to learn a good representation is via self-Supervised learning. The key idea of self-Supervised learning is to predict part of the input from other parts in some form (e.g., add distortion). By minimizing prediction task loss or other auxiliary task losses, the neural network learns good presentations that can be used to speed up downstream tasks.

Since the advent of the most successful pretrained language model BERT[12], many follow-up research found that the performance of the pretrained model in downstream tasks highly depend on the self-supervised tasks in the pretraining stage. If the downstream tasks are closely related to self-supervised tasks, a pretrained model can offer significant performance boost. And the fine tuning process can be understood as a process that further improves features relevant to downstream tasks and discards irrelevant features.

10.6.2 BERT

10.6.2.1 *The model*

BERT, Bidirectional Encoder Representations from Transformers[12], is one of the most successful pre-trained language model. BERT relies on a Transformer (the attention mechanism that learns contextual relationships between words in a text). A basic Transformer consists of an encoder to read the text input and a decoder to produce a prediction for the task. since BERT's goal is to generate a language representation model, it only needs the encoder part.

There are two tasks to pretrain the network: masked language modeling (Marked LM) and next sentence prediction (NSP).

In the Masked LM, some percentage of randomly sampled words in a sequence are masked, i.e., being replaced by a [MASK] token. The task is to predict only the masked words, based on the context provided by the other non-masked words in the sequence.

In the NSP, the network is trained to understand relationship between two sentences. A pre-trained model with this kind of understanding is relevant for tasks like question answering and natural language Inference.. During training the model gets as input pairs of sentences and it learns to predict if the second sentence is the next sentence in the original text as well. Specifically, when choosing the sentences pair for each pre-training example, 50% of the time, the second sentence is the actual next sentence of the first one, and 50% of the time, it is a random sentence from the corpus. By doing so, it is capable to teach the model to understand the relationship between two input sentences and thus benefit downstream tasks that are sensitive to this information, such as Question Answering and Natural Language Inference.

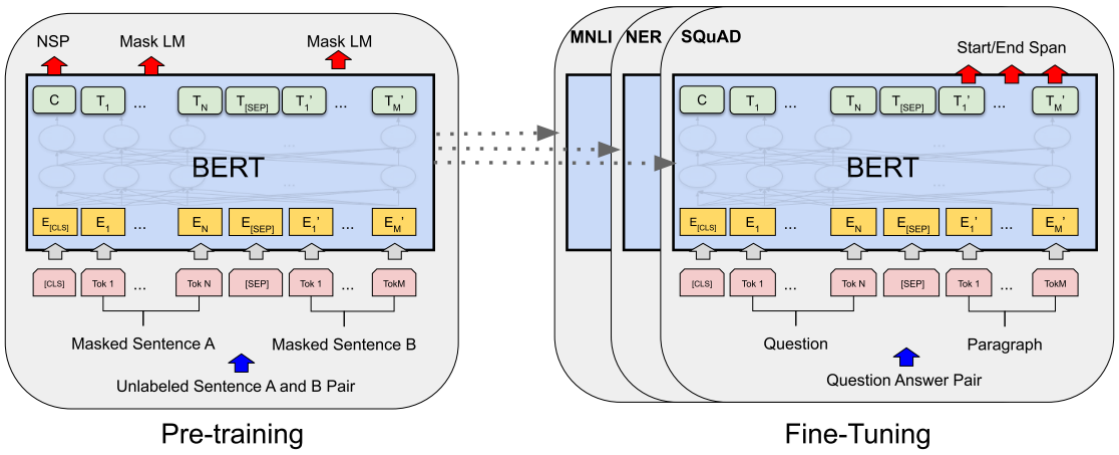


Figure 10.6.3: BERT pre-training and downstream task fine tuning framework. Image from [12].

The input to the encoder for BERT is a sequence of tokens, which are first converted into vectors and then processed in the neural network. The encoding of each word in a sequence consisting of following components:

- **Token embeddings**, which is the ordinary dense word embedding. Note that A [CLS] token is added to the input word tokens at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- **Segment embeddings**, which is a marker 0 or 1 indicating if sentence A precedes sentence B.
- **Positional embeddings**, which encodes information of position in the sentence.

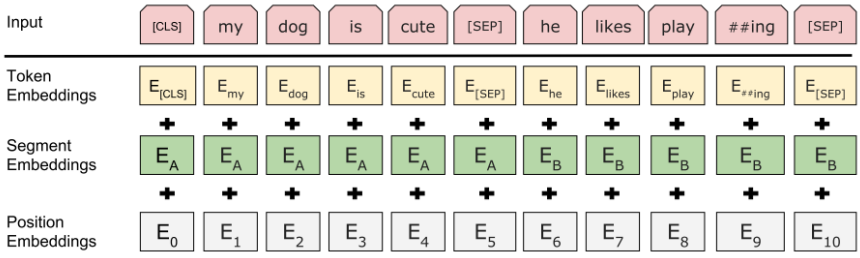


Figure 10.6.4: Input embedding in BERT, which consists of token embedding, segmenet embedding and positional embedding. Image from [12].

10.6.3 Fine tuning procedures

10.7 Other BERT family members

10.7.1 ALBERT

BERT have achieved marked success in tackling challenging NLP tasks such as machine translation and question answering. In a end-to-end system, BERT can used as an encoder, which connects back-end task-specific modules (e.g., classifiers). By fine-tuning the BERT, the system can usually achieve satisfactory results even under limited resources setting (e.g., small training set). However, BERT is huge model, the base version has 108M parameters, which prohibits its application in devices with limited memory and computation power. There have been constant efforts to develop smaller versions of BERT without significant compromise on its performance.

ALBERT[19], standing for A Lite BERT, is one of the recent achievement that reduces the model parameter number considerably and at the same time improves performance. In ALBERT , there are three major improvements on the model architecture and the pretraining process. The first two improvements on the model architecture are

- **Factorized embedding parameterization**. In the original BERT, tokens (represented as one-hot vectors) are directly projected the hidden space with dimensionality $H = 768$ or 1024. For large vocabulary size V at the scale of 10,000, the projection matrix W has parameters HV. One way to reduce parameter size is to factorize W into two lower rank matrices. This is equivalent to two-step projection:
  - First project one-hot vector to embedding space of dimensionality E, say $E = 128$;
  - Second project the embedding space to the hidden space of dimensionality H.

  The two-step projection only requires parameters $VE + EH$ and the reduction is notable when $E \ll H$.
- **Cross-layer parameter sharing**. In the original BERT, each encoder sub-unit (which has an attention module and a feed-forward module) has different parameters. In ALBERT, these parameters are shared in different sub-units to reduce model size and improve parameter efficiency. The authors also mentioned that there are alternative parameter sharing strategies, for example, only sharing feed-forward network parameters or only sharing attention module parameters. could also be

The effect of embedding factorization and parameter sharing can be illustrated in the comparison [Table 10.7.1].

Model		Parameters	Layers	Hidden	Embedding	Parameter-sharing
BERT	base	108M	12	768	768	False
	large	334M	24	1024	1024	False
ALBERT	base	12M	12	768	128	True
	large	18M	24	1024	128	True
	xlarge	60M	24	2048	128	True
	xxlarge	235M	12	4096	128	True

Table 10.7.1: Comparison of model parameter size for different configurations of BERT and ALBERT models.

The third improvement in ALBERT over BERT is a new loss in next sentence prediction task. In ALBERT, a new sentence-order prediction (SOP) loss to model inter- sentence coherence, which demonstrated better performance in multi-sentence encoding tasks.

10.7.2 XL-Net

XLNet: Generalized Autoregressive Pretraining for Language Understanding [16]

10.7.3 RoBERTa

[20]

10.7.4 Computational efficient architectures

Reformer [21], Longformer [22]

10.7.4.1 *Overview*

Although the Bert model is very powerful, the time and space complexity of the two-way attention increases in a trend of N^2, where the N is the length of the sequence. To ensure reasonable performance, the maximum length of original BERT model is limited to 512.

Under this restriction, one workaround when dealing with longer sequences is to cut the long sequence into shorter sequences (maximum 512 tokens per sequence) and feed them into BERT to obtain separate embeddings. The final output is the concatenation of all embedding.

This workaround loses the dependence between different sequences and it is desirable to have an architecture that can directly handle long sequences in efficient ways.

In this section we discuss a couple of strategies employed to address the efficiency issue in the original BERT.

- Reformer[21] uses local sensitive hashing to solve performance problems. The motivation is that attention works on top-N rather than all.
- The adaptive width uses a dynamic window method to solve the problem. The motivation is that attention may only attend the nearest context.
- Longformer[22] uses a more direct way to transform attention. That is the combination of local attention and global attention. Local attention is used to capture local information and is generally used in the bottom layer.

10.7.4.2 *Reformer*

10.7.4.3 *Longformer*

10.7.5 Model distillation

10.7.5.1 *Patient knowledge distillation*

Although BERT has achieved notable success in addressing NLP challenges, its large model size and costly computation prohibits its wide adoption in industrial application. One approach to overcoming the limitation is knowledge distillation [23], a technique that enables transfer knowledge from a large, complex network to a light-weighted, small network.

Mathematically, let the original large model be the teacher and be represented by a function mapping $f(x;\theta)$ with parameter θ, where x is the input. Knowledge distillation aims to learn a new set of parameters θ' for a shallower student network $g\left(x;\theta'\right)$, such that the student network achieves comparable performance to the teacher. The student model is a light-weighted, small network, with much lower computational cost. Knowledge distillation therefore bridges the gap between complex model and practical applications.

In the context of BERT sequence modeling, consider $\left\{x_i, y_i\right\}_{i=1}^N$ are N training samples, where x_i is the i -th input instance for BERT, and y_i is the corresponding ground-truth label. Denote the contextualized embedding from BERT by

$$h_i = \text{BERT}\left(x_i\right) \in \mathbb{R}^d.$$

A probability distribution on y_i is approximated by feeding h_i to a Softmax layer, which gives $\hat{y}_i = P\left(y_i \mid x_i\right) = \text{softmax}\left(Wh_i\right)$.

Let the student model be a BERT style encoder but with much fewer layers.

In the first attempt to knowledge distillation[24], we encourage the probabilistic predictions between the student and the teacher model to agree with each other. This can be achieved by minimizing cross-entropy loss, with the target label coming from both the prediction of the teacher model and the actual ground truth. Taken together, the final knowledge distillation loss function is given by

$$\begin{aligned}
L_{KD} &= (1-\alpha) L_{CE}^s + \alpha L_{DS} \\
L_{DS} &= -\sum_{i \in [N]} \sum_{c \in C} \left[P^t\left(y_i = c \mid x_i; \hat{\theta}^t\right) \log P^s\left(y_i = c \mid \mathbf{x}_i; \theta^s\right) \right] \\
L_{CE}^s &= -\sum_{i \in [N]} \sum_{c \in C} \left[\mathbb{1}\left[\mathbf{y}_i = c\right] \log P^s\left(\mathbf{y}_i = c \mid \mathbf{x}_i; \theta^s\right) \right]
\end{aligned}$$

where L_{KS} is the distillation loss, α is a scalar, L_{DS} is the student-teacher distance loss, and L_{CE} is the student cross-entropy loss. The superscripts t and s denote teacher and student model, respectively.

A further step to distill knowledge from the teacher to the student is to encourage student learning not just on the last classification outcome, but also on intermediate outputs. The authors proposed two patient distillation strategies [Figure 10.7.1]:

- PKD-Skip, where the student learns from every k layers of the teacher;
- PKD-Last, where the student learns from the last k layers of the teacher.

Denote the intermediate hidden state output by

$$h_i = \left[h_{i,1}, h_{i,2}, \ldots, h_{i,k}\right] = \text{BERT}_k\left(x_i\right) \in \mathbb{R}^{k \times d}$$

The additional training loss introduced by the patient teacher is defined as the mean-square loss between the normalized hidden states:

$$L_{PT} = \sum_{i=1}^{N} \sum_{j=1}^{M} \left\| \frac{h_{i,j}^s}{\left\| h_{i,j}^s \right\|_2} - \frac{h_{i,I_{pt}(j)}^t}{\left\| h_{i,I_{pt}(j)}^t \right\|_2} \right\|_2^2$$

where M denotes the number of layers in the student network, N is the number of training samples, and the superscripts s and t in $\mathbf{h}$ indicate the student and the teacher model, respectively. Note that We denote the set of intermediate layers to distill knowledge from as I_{pt}. Take distilling from BERT_{12} to BERT_{6} as an example. For the PKDSkip strategy, $I_{pt} = \{2, 4, 6, 8, 10\}$; and for the PKD-Last strategy, $I_{pt} = \{7, 8, 9, 10, 11\}$.

Combined with the knowledge distillation loss introduced previously, the final loss function is given by

$$L_{PKD} = (1 - \alpha)L_{CE}^{s} + \alpha L_{DS} + \beta L_{PT}.$$

where β is another hyper-parameter that weights the importance of the features for distillation in the intermediate layers.

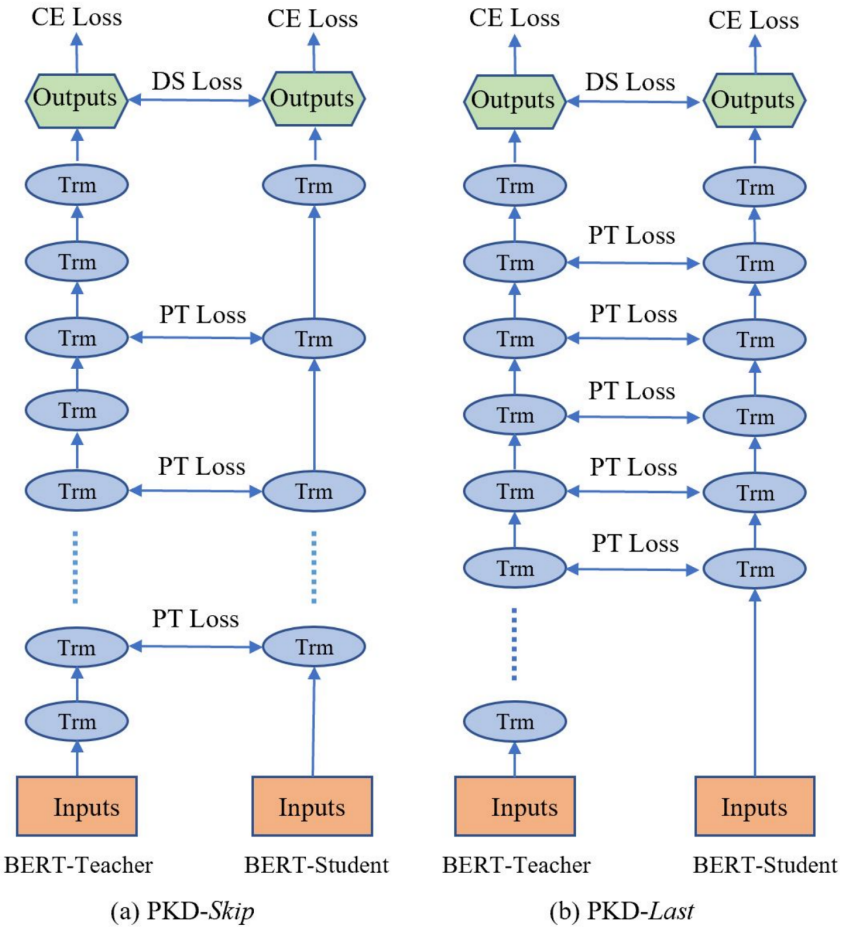


Figure 10.7.1: The two proposed Patient Knowledge Distillation approaches to distill knowledge from teacher model to student model. Trm stands for Transformer. The total loss consists of cross-entropy loss on the classification outcome and the intermediate output mean-square loss. Image from [24].

10.8 Notes on Bibliography

General books on Natural language processing include [25][26][27]. Practical books include [25][27].

For a comprehensive review on deep learning methods for text classification, see [28].

BIBLIOGRAPHY

1. Mikolov, T., Chen, K., Corrado, G. & Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
2. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. & Dean, J. *Distributed representations of words and phrases and their compositionality* in *Advances in neural information processing systems* (2013), 3111–3119.
3. Bengio, Y., Ducharme, R., Vincent, P. & Jauvin, C. A neural probabilistic language model. *Journal of machine learning research* **3,** 1137–1155 (2003).
4. Mikolov, T., Karafiát, M., Burget, L., Cernocký, J. & Khudanpur, S. *Recurrent neural network based language model.* in *INTERSPEECH* (eds Kobayashi, T., Hirose, K. & Nakamura, S.) (ISCA, 2010), 1045–1048.
5. Dong, L., Wei, F., Liu, S., Zhou, M. & Xu, K. A statistical parsing framework for sentiment classification. *Computational Linguistics* **41,** 293–336 (2015).
6. Kim, Y. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).
7. Sutskever, I., Vinyals, O. & Le, Q. V. *Sequence to sequence learning with neural networks* in *Advances in neural information processing systems* (2014), 3104–3112.
8. Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
9. Luong, M.-T., Pham, H. & Manning, C. D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).
10. Wu, Y. *et al.* Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).
11. Vaswani, A. *et al. Attention is all you need* in *Advances in neural information processing systems* (2017), 5998–6008.
12. Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
13. Radford, A., Narasimhan, K., Salimans, T. & Sutskever, I. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf* (2018).

14. Radford, A. *et al.* Language models are unsupervised multitask learners. *OpenAI Blog* **1,** 9 (2019).
15. Dai, Z. *et al.* Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).
16. Yang, Z. *et al. Xlnet: Generalized autoregressive pretraining for language understanding* in *Advances in neural information processing systems* (2019), 5754–5764.
17. Lample, G. & Conneau, A. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291* (2019).
18. Qiu, X. *et al.* Pre-trained Models for Natural Language Processing: A Survey. arXiv: 2003.08271 (2020).
19. Lan, Z. *et al.* ALBERT: A Lite BERT for Self-supervised Learning of Language Representations, 1–17. arXiv: 1909.11942 (2019).
20. Liu, Y. *et al.* Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
21. Kitaev, N., Kaiser, Ł. & Levskaya, A. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451* (2020).
22. Beltagy, I., Peters, M. E. & Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).
23. Hinton, G., Vinyals, O. & Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
24. Sun, S., Cheng, Y., Gan, Z. & Liu, J. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355* (2019).
25. Manning, C. D. & Schütze, H. *Foundations of statistical natural language processing* (MIT press, 1999).
26. Jurafsky, D., Martin, J., Norvig, P. & Russell, S. *Speech and Language Processing* ISBN: 9780133252934 (Pearson Education, 2014).
27. Goldberg, Y. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies* **10,** 1–309 (2017).
28. Minaee, S. *et al.* Deep Learning Based Text Classification: A Comprehensive Review. *arXiv preprint arXiv:2004.03705* (2020).