Deep Learning for Natural Language Processing

IRS ML Section[⋆]

Hwa Chong Institution

1 Introduction to Natural Language Processing

1.1 What is NLP?

In its most general form, NLP is the study of allowing machines to understand language as it is used by humans. Note that this document will largely concern the use of deep learning to extract the meaning of text, which is the focus of most current NLP research.

1.2 Why do we need NLP?

Natural language analysis. Much of human-generated data is in the form of text - after all, tens of billions of text messages are sent every day, in addition to the myriad text-based social media posts made by hundreds of millions of users worldwide. As with other fields of data analysis, when data is generated, people would like to draw insights from it. Since the sheer volume of textual data precludes manual processing, data scientists have naturally turned to computers to analyse it.

Natural language generation. Several language-related tasks involve not just analysing human language but also generating it. The latter tends to be even more time-consuming for humans, hence the desire to automate it as well. As it turns out, natural language generation is typically also more difficult than pure language analysis from a computational perspective.

Human-computer interaction. By endowing computers with the ability to understand (or at least process) human language, we can and have significantly streamlined the way we use computers. Thanks to applications of NLP in human-computer interaction, most owners of modern smartphones now possess convenient alternatives to button-pushing, in the form of Siri or Google Assistant.

1.3 Some applications of NLP

- Sentiment analysis
- Speech recognition

[⋆] Written by Ho Wing Yip and Marcus Wee

- Fake news detection
- Machine translation
- Text summarization
- Question answering

2 Linguistic Preprocessing

Before applying machine learning to language data - or for that matter, any kind of data - it often helps to process the data beforehand. Preprocessing language data usually serves to reduce it down to its most salient features, decreasing the complexity of the problem we are trying to solve. In this section, we detail a number of techniques for doing so.

2.1 Punctuation removal

To allow our models to analyse our data as easily as possible, we aim to distil the language down to its bare meaning, removing all extraneous noise. Consider the following un-punctuated sentence:

I prefer not to eat meat it just doesn't taste fantastic and environmentalists say it's bad for the planet anyway

While it obviously doesn't read very elegantly, one can easily discern what the person is trying to say. To machine learning models that aim to infer the meaning of a piece of text, punctuation is thus little more than a distraction most of the time. Hence, punctuation is typically removed during preprocessing.

2.2 Lowercase conversion

In a similar vein, whether a word is in uppercase or lowercase is often unimportant in NLP, though this varies between tasks. As such, we convert text to lowercase where applicable.

2.3 Tokenization

For humans, an intuitive starting point for analysing text is to split it into its constituent sentences, phrases, words and perhaps even characters. The same principle is applied in NLP, where processing a piece of text begins by splitting it into its linguistic building blocks, or *tokens*. This splitting process is termed *tokenization*.

Typically, language models operate on more granular tokens such as words, subwords or characters. Such tokens tend to recur more often than "coarser" tokens like sentences, making for more consistent and widely applicable analysis.

Word-level tokenization:
`Mary / had / a / beautiful / little / lamb / with / sparkling / eyes`

Subword-level tokenization:
`Mary / had / a / beauti / ful / little / lamb / with / sparkl / ing / eyes`

Character-level tokenization:
`M / a / r / y / h / a / d / a / b / e / a / u / t / ...`

Fig. 1: Various types of tokenization

2.4 Stopword removal

In a similar vein to punctuation removal, one often finds that certain words are of little use in determining the meaning of a piece of text. Consider the following sentence:

I prefer not to eat meat – it just doesn't taste fantastic, and environmentalists say it's bad for the planet anyway.

Now, let us remove some words:

I prefer not eat meat doesn't taste fantastic environmentalists say bad for planet

Despite the lack of some prepositions like 'to' and pronouns like 'it', the meaning of the text remains relatively clear - can you still infer the two reasons why the speaker does not wish to eat meat?

Unimportant words like the ones omitted above are known as *stopwords*, and are commonly removed during preprocessing. Note that stopwords may vary between tasks - for instance, when determining the topic of a piece of text, it is usually safe to ignore the word 'not', as the subject matter being discussed is unlikely to depend on it. In contrast, if one wishes to examine the sentiment of the text, omitting 'not' can cause one's prediction to be the complete opposite of the actual opinion expressed. It is thus important to be discerning when applying NLP preprocessing techniques, for there is no one-size-fits-all preprocessing solution for all tasks.

Most NLP toolkits like NLTK and spaCy implement a list of common stopwords for several languages. You may also choose to implement your own stopword list if you find it comprehensive enough.

2.5 Stemming and lemmatization

Stemming and lemmatization are two ways to reduce various inflections of the same root word (e.g. 'walks', 'walking', 'walked') to a common form (e.g. 'walk'). This can reduce the size of our vocabulary - and thus the complexity of our task - without sacrificing semantic information, as the meaning of words typically remains the same no matter which grammatical form they are in.

Stemming is the process of removing or replacing common word affixes; in English, examples include '-ing', '-ed', and '-ful'. The stemmed word need not be a valid word in the language. It is important to note that stemming is performed only on **single words** with **no regard for context**, making it a rather naive approach. The most popular stemming algorithm is Porter's algorithm, a full description of which can be found at http://snowball.tartarus.org/algorithms/porter/stemmer.html.

While stemming is usually an acceptable rudimentary solution, such an approach is too simplistic to deal with the fickle variety present in languages like English. As an example, consider the fact that the root form of 'better' is not 'bett' but 'good'. Thus, when possible, we turn to a more complete method – lemmatization.

Lemmatization converts words to their root form or *lemma*, taking into account **context** and **meaning**. Consider the word 'meeting' – depending on context, it could be either a noun in its root form, or the '-ing' form of the verb 'meet'. An ideal lemmatizer uses context to select the appropriate lemma. Many popular NLP packages like NLTK and spaCy have lemmatizers built in.

TODO: Provide sample code for preprocessing libraries (nltk, Spacy)

3 Word Encoding

3.1 One-hot encoding

Now that we've preprocessed our data, we need to make it machine-readable. Each word, or *token*, must thus be converted into a numerical form. Since the set of possible words is discrete, we represent each word by a one-hot-encoded vector. Denoting the set of all distinct words in our data (i.e. our *vocabulary*) by V, word i is represented by a $|V|$-element vector containing a 1 in position i and zeroes everywhere else.

Example. Suppose our vocabulary consists of the five words 'apple', 'banana', 'i', 'like' and 'no'. We could convert them to one-hot vectors as follows:

Index	**Word**	**One-hot representation**
1	apple	$[1, 0, 0, 0, 0]$
2	banana	$[0, 1, 0, 0, 0]$
3	i	$[0, 0, 1, 0, 0]$
4	like	$[0, 0, 0, 1, 0]$
5	no	$[0, 0, 0, 0, 1]$

Fig. 2: A possible one-hot representation of a 5-word vocabulary

Each document then becomes a sequence of vectors. This can be represented by a matrix consisting of the one-hot representations of all words in the document simply stacked on top of each other. Fig. 3 illustrates this for the sentence “i no like banana”.

$$\begin{array}{r} \text{i} \\ \text{no} \\ \text{like} \\ \text{banana} \end{array} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 3: The matrix representation of “i no like banana”. Each row is the one-hot representation of its associated word.

Note: It is also useful to have an *out-of-vocabulary (OOV) token* to represent words not in the vocabulary. Such words may be encountered in test data, and in real-world model use.[1]

3.2 Word embedding

Since we represent words as vectors, it makes sense to use the angle between word vectors to measure their similarity; precisely, we quantify the similarity by $\cos\theta$, where $0^\circ \leq \theta \leq 180^\circ$ is the angle between the vectors. The intuition behind this is as follows: if $0^\circ \leq \theta < 90^\circ$, the word vectors point in similar directions, meaning that their similarity should be positive. If $\theta = 90^\circ$, the word vectors are orthogonal and thus “unrelated”, implying zero similarity. If $90^\circ < \theta \leq 180^\circ$, the word vectors point in “opposite” directions, and should thus have negative similarity. One can easily verify that $\cos\theta$ is positive when $0^\circ \leq \theta < 90^\circ$, zero when $\theta = 90^\circ$, and negative when $90^\circ < \theta \leq 180^\circ$, adhering to the properties we expect of a similarity measure.[2]

From this perspective, however, one finds that our one-hot representation encodes no information about word similarity. All our words are orthogonal and would thus be considered unrelated to each other, which is obviously not accurate! As such, we use a process called *embedding* to convert word vectors into a semantically richer form that encodes similarity information.

[1] In TensorFlow, OOV tokens are typically represented as a word with index 0. The vocabulary size increases by one, and one-hot encoding proceeds as usual, except with the first word having a 1 in position 2, the second word having a 1 in position 3, and so on. Note that the programmatic *index* of the 1 remains the same, as indexing in Python is zero-based.

[2] To be precise, $\cos\theta$ describes the magnitude and direction of the projection of the normalized word vectors in the direction of each other, thus capturing the idea of similarity.

We embed the one-hot vectors by linearly projecting them to an *embedding space*, where similar words should correspond to vectors pointing in similar directions. The linear projection is performed by multiplying the one-hot vectors by an *embedding matrix*, denoted by E. The dimension of the embedding space is a hyperparameter, which we shall call e.[3] Since our vocabulary has size $|V|$, each one-hot vector will have $|V|$ elements. Hence, E is an $e \times |V|$ matrix.

It is important to note that the model **learns** the embedding matrix, so as to generate the best possible word representations for the task at hand. There also exist pretrained embedding weights learned for a single task that have proven effective for several other applications.

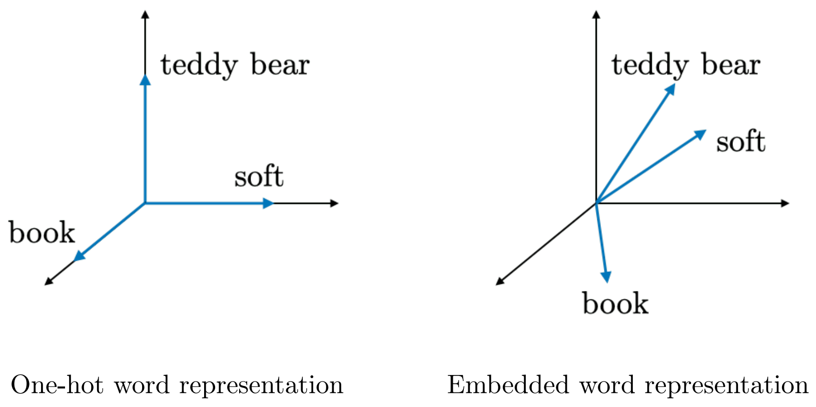

Fig. 4: Illustration of one-hot vs. embedded word representations. Image credit: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Learning embeddings. Normal layer weight learning through backpropagation, word2vec, skip-gram, negative sampling, GloVe, etc.

4 NLP Models

4.1 Recurrent neural networks

Recurrent neural networks (RNNs) are a type of neural network for sequence processing. This makes them appropriate for NLP, as text can be represented as a sequence of words or characters. Note that in virtually all deep learning frameworks, RNNs are implemented as individual layers that can be combined

[3] Typically, e is on the order of 10^1 to 10^2.

with other types of layers to create a full network. It is thus more appropriate to discuss RNNs as a kind of layer rather than a network in itself.

An RNN layer receives a sequence of vectors $\{x_1, x_2, ..., x_n\}$ as input, and processes each element or *timestep* in order. In the context of NLP, each timestep is usually an embedded word vector. After processing the t-th timestep x_t, the layer generates a hidden output vector h_t and (actual) output vector y_t, taking into account both the current timestep x_t, and either the previous hidden output h_{t-1} or the previous actual output y_{t-1}.

A basic RNN layer known as an *Elman network* can be described by two equations:

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \tag{1a}$$

$$y_t = \sigma_y(W_y h_t + b_y) \tag{1b}$$

where W_h, U_h and W_y are weight matrices, b_h and b_y are bias vectors, and σ_h and σ_y are activation functions (typically sigmoid and tanh respectively). An alternative formulation known as a *Jordan network* replaces equation (1a) by

$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h) \tag{2a}$$

Shortcomings of RNNs. While RNNs used to enjoy relative ubiquity for sequence processing tasks, a number of issues began to surface – namely, the inability to parallelize RNN layer computation, and the vanishing and exploding gradient problems.

The parallelization issue arises from the fact that computing y_t requires h_{t-1} or y_{t-1}. Since the current output depends on all previous outputs, the output for all timesteps in the sequence cannot be computed simultaneously. Thus, the computation cannot be fully parallelized, making it slower. This is in contrast to other architectures like CNNs, where the same filter can be applied to multiple regions of an image at once, allowing the layer to compute its whole output at the same time.

The vanishing and exploding gradient problems are arguably more severe, since they concern not just computation speed but the network's ability to learn. RNNs are trained using *backpropagation through time* (BPTT). Roughly speaking, BPTT "unrolls" the network and computes the error at each timestep E_t, before calculating the sum of errors $E := \sum_{t=1}^{n} E_t$ and updating the weights through normal backpropagation according to $\partial E/\partial W$.

It turns out that the calculation of $\partial E/\partial W$ involves repeated multiplication by the same matrices, and that a repeated product of matrices can explode to zero or infinity just like a repeated product of real numbers. This causes the value of $\partial E/\partial W$ to become exponentially large ("exploding") or exponentially small ("vanishing").[4]

[4] A detailed mathematical analysis can be found at https://proceedings.mlr.press/v28/pascanu13.pdf.

In practice, vanishing or exploding gradients mean that the network either excessively takes into account inputs from many timesteps ago, or begins to forget them as more timesteps are processed. This is worrying for NLP, as we want our network to be able to properly model relationships between words irrespective of the distance between them. A number of mitigations for these issues have thus been developed, the most well-known of which is the long short-term memory (LSTM) architecture.

LSTM utilizes an additional “cell state” c_t that, in theory, should allow more information to flow from earlier timesteps to later timesteps unperturbed. Despite this, even LSTM suffers from the vanishing gradient problem, preventing it from learning long-term dependencies between inputs more than 1,000 timesteps apart. Eventually, a major architectural shift away from RNNs arrived in the form of transformers, which resolve the vanishing and exploding gradient problems associated with processing sequences in order.

4.2 Transformers

Transformers are *encoder-decoder* networks which receive sequential input and generate sequential output. Transformers use a mechanism called *attention* which does not process the input sequence in any particular order, and thus allows the model to draw information from any timestep. This allows relationships between words to be modelled regardless of how far apart the words are.

In the attention mechanism, each token x_i in the input sequence is multiplied by three weight matrices: the query weights W_Q, the key weights W_K, and the value weights W_V. The resulting vectors q_i, k_i and v_i are known as the query, key and value vectors respectively. The *attention weight* between two tokens x_i and x_j is given by $q_i \cdot k_j$, and it represents how much token i “attends” to token j; i.e. how relevant token i is to token j. In practice, the query, key and value vectors are stacked into matrices, denoted by Q, K and V, and the attention computation is expressed as

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

where d_k is the length of the key and query vectors.[5] It is suspected that when d_k is large, the dot products $q_i \cdot k_j$ also become large, pushing the softmax function into regions where its gradients are small. To counteract this effect, the scaling factor of $\frac{1}{\sqrt{d_k}}$ is used.

Each set of matrices (W_Q, W_K, W_V) is called an attention head, and each layer in a transformer model contains multiple attention heads. This allows each layer to model several definitions of attention. The encoder attention layers calculate

[5] Two main variants of attention exist: *scaled dot-product attention* and *additive attention*. The former is described here, and is more efficient as it can be implemented as a single matrix computation.

attention on the input sequence, and the decoder layers calculate attention on the sequential output of the encoder layers. For the latter, an *attention mask* is implemented, which prevents the decoder from "cheating" by using information from the current or future timesteps when predicting on the current timestep.

Transformers have achieved higher accuracies than RNNs on several sequence processing tasks both related and unrelated to NLP. Recently, transformers have even begun to outperform CNNs on object detection. However, transformers' state-of-the-art accuracy comes at the expense of vastly increased computational cost.

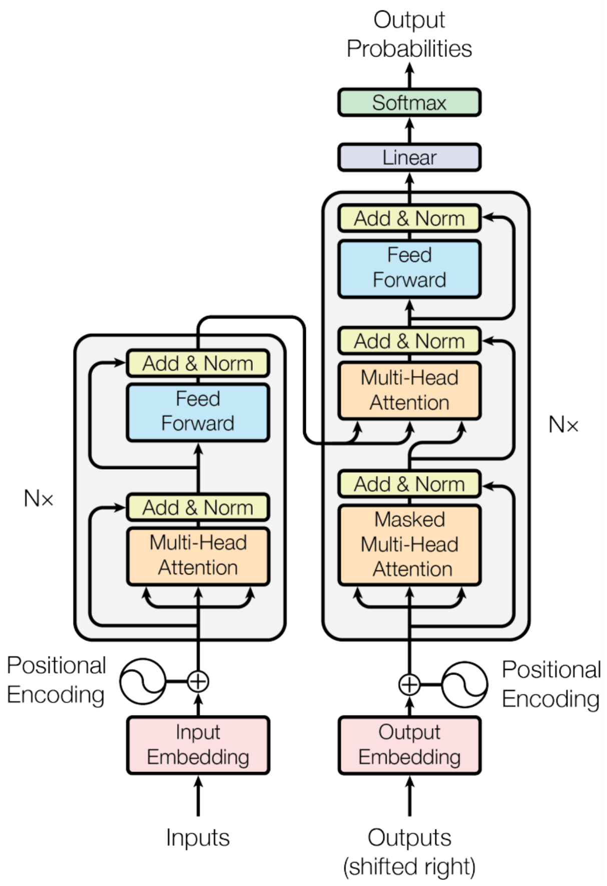

Fig. 5: The transformer architecture as it was implemented in the original paper, *Attention Is All You Need*. Image credit: https://arxiv.org/pdf/1706.03762.pdf.