Sequence to Sequence Modelling

Patrick Kahardipraja

September 2019

Mapping of a sequence to another sequence is an important paradigm because of the vast amount of problems that can be formulated in this manner. For instance, in automatic speech recognition (ASR), chunks of speech signals can be mapped to sequence of phonemes while in machine translation, a sequence of words in one language can be mapped to another language. Interestingly, many other tasks such as text summarization, question answering, and image caption generation can be phrased as a sequence to sequence problem. In this paper-style essay, I will attempt to distill how sequence to sequence learning works and the motivation behind it, with a particular focus on machine translation.

Introduction

Prior to neural machine translation (NMT), phrase-based stastistical machine translation (SMT) systems are widely used as it offers reliable performance. Despite its success, most of them are extremely complex and require a huge amount of effort, as it is often tailored to a specific language pair and do not generalize well to another languages. Furthermore, a lot of feature

engineering are required in order to capture a specific language phenomena, which prompt researchers to explore another approach.

The resurgence of deep neural networks (DNNs) in early 2010s, thanks to faster, parallel computation using GPUs and availability of large and high-quality datasets, bring a new wave of enthusiasm in deep models. With the capability to learn features automatically with multiple, hierarchical representation, DNNs achieve excellent performance on difficult tasks in computer vision [AlexNet] and speech recognition []. Albeit powerful, DNNs has its own limitation, as it requires input and output vectors with a fixed dimension and thus not suitable for sequence to sequence problem whose lengths are unknown beforehand. In addition, DNNs also do not generalize well across temporal patterns, because each neuron has its own specific connection and as a result, a single pattern may look totally different at different timesteps.

The natural remedy for this problem is to look onto recurrent neural networks (RNNs), as it allows operations over sequences of vectors. However, mapping using RNNs typically have one-to-one correspondence between the input vectors and the output vectors. It also has another problem, as the input and output sequences can have different lengths and non-monotonic alignments. Standard RNN architecture is also not reliable for learning long-range dependencies due to the vanishing gradient problem. This issue is addressed by Sutskever et al. [Seq2Seq], where they introduce a novel and straightforward method to solve general sequence to sequence mapping using Long-Short Term Memory (LSTM) architecture. With the success of sequence to sequence learning in machine translation tasks, research in neural machine translation continue to thrive, eventually resulting in many significant improvements such as attention mechanism [Bahdanau] and subword units to deal with rare words [WordPiece]. But, before delving in too deep, I will give some brief insight into the mechanism behind RNN and LSTM in the next section.

Recurrent Neural Networks

Recurrent neural networks [Rumelhart] are type of neural network that is able to process arbitrary sequential input via combination of its internal state and input vector. At every timestep t, the hidden state vector h_t is overwritten as a function of the hidden state at the previous timestep h_{t-1} and the current input vector x_t. The input vector x_t itself could be a representation of t-th word in a sentence, which is usually obtained using pre-trained word embeddings [GloVe, Word2Vec, ElMo]. The hidden state of RNNs can be perceived as a memory with a fixed dimensionality that can be tuned, containing distributed representation of the processed input sequence up to time t.

In a RNN, the forward step function consists of an affine transformation followed by a non-linear activation function. The hidden state then can be used to make predictions:

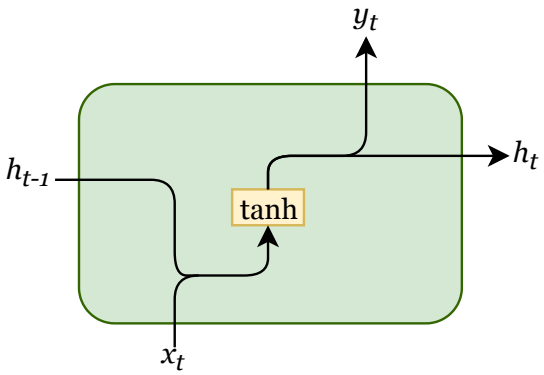
$$h_t = a(W^{(x)}x_t + W^{(h)}h_{t-1} + b_h) \tag{1}$$

$$y_t = g(W^{(y)}h_t + b_y) \tag{2}$$

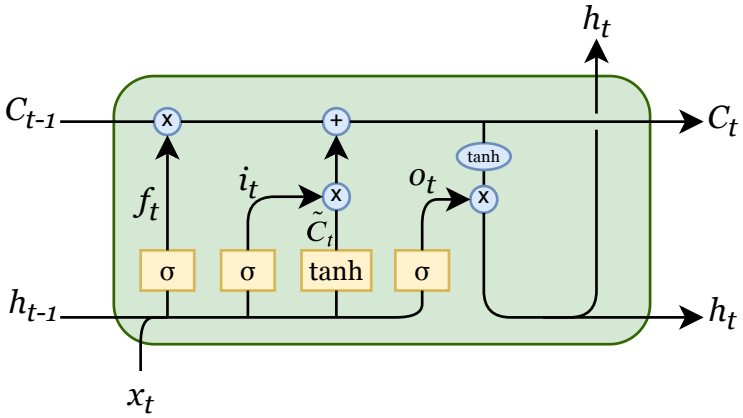
where in a typical application, a is the hyperbolic tangent function and g is the softmax function.

Although proven to be effective, RNN still has its own shortcoming. The main problem with RNN is that during training, magnitude of gradient can get weaker or stronger exponentially when backpropagating the error through time, especially with long sequences [Hochreiter, Bengio]. This phenomena is called vanishing or exploding gradient problem, which causes RNN model to experience difficulty when handling "long-term dependencies" that occur in a sequence.

Long Short Term Memory (LSTM) architecture [Hochreiter and Schmidhuber] addresses the problem of "long-term dependencies" by integrating a memory cell that is capable to memorize state that span over long sequences of time. The memory cell is controlled by gates, which have the ability to regulate how much information are added or removed in the memory cell. This means that while in a RNN a completely new hidden state is computed at every new timestep, in LSTM the hidden state is not competely overwritten, and updated according to the memory cell. The architecture of both RNNs and LSTMs are depicted in Figure 1.



(a) RNN unit



(b) LSTM unit, colah

Figure 1: Architecture of RNN (a) and LSTM (b)

A LSTM unit consists of 3 gates (input gate i_t, forget gate f_t, output gate o_t), memory cell C_t and hidden state h_t. In a high-level sense, the input gate decides how much and which values will be updated, the forget gate controls the amount of information to be forgotten in the previous memory cell, and the output gate decides the hidden state by filtering the internal memory cell for each timestep. Each gate produces vectors, where their values are between 0 (completely closed) and 1 (completely open) using the sigmoid activation function.

The formula of LSTM is described with the following equations:

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1} + b_i) \tag{3}$$
$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1} + b_f) \tag{4}$$
$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1} + b_o) \tag{5}$$
$$\tilde{C}_t = \tanh(W^{(c)}x_t + U^{(c)}h_{t-1} + b_c) \tag{6}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{7}$$
$$h_t = o_t \odot \tanh(C_t) \tag{8}$$

where x_t is the input vector for timestep t, σ is the sigmoid activation function and $\odot$ denotes the Hadamard product of matrices.

Variants of LSTM

Beside the standard LSTM architecture in literature that is introduced above, there also exist several popular variants which are commonly used. One of the variations which is introduced by [Gers, Schmidhueber] use "peephole connections". These connections allow the gates to look into the memory cell state in order to learn precise and stable timings.

Other notable LSTM variants is the Gated Recurrent Unit (GRU), in-

troduced by [Cho et al.]. Unlike LSTM, GRU is less complex and requires less computation. GRU only have 2 gates, the update gate that decides how much information will be transferred from previous and candidate hidden state to the current one and reset gate that controls to what extent the previous hidden state will affect the candidate hidden state. In this manner, the update gate can be thought as a combination of input and forget gates of LSTM unit. This architecture also merges the internal memory cell and hidden state of LSTM into a single hidden state.

Related Works

The first model that is able to map a sentence into a vector and then to its translation is introduced by [Kalchbrenner et al.]. The model, which they called Recurrent Continuous Translation Models (RCTM), is composed of 2 separate parts: convolutional neural network (CNN) for modelling the source sentence as the encoder and recurrent neural network for translation generation as a language modelling task, conditioned on the source sentence as the decoder. With this approach, the encoder can capture all the information contained in the source word representations and create a representation of the source sentences. The representation for the source sentences also restraint the generation of the target words in the language modelling phase.

The Recurrent Continuous Translation Models estimate the probability distribution over the sentence in the target language given sentences in the source language. Suppose that there exist a target sentence $f = f_1, f_2, ..., f_m$, which is a translation of source sentence $e = e_1, e_2, ..., e_n$. Then $P(f|e)$ can be obtained with the formula:

$$P(f|e) = \prod_{i=1}^{m} P(f_i|f_{1:i-1}, e) \tag{9}$$

As can be seen in the formulation above, the model estimates $P(f|e)$ by calculating the conditional probability $P(f_i|f_{1:i-1}, e)$ for every translated word occuring at position i, given the preceding generated words $f_{1:i-1}$ in the target sentence and the source sentence e. Conditioning the translation model to the preceding target words also ensure that it incorporates the target language model [Kalchbrenner et al.].

In RCTM, prediction of the target sentence use a language model based on a recurrent neural network [Mikolov et al.]. The recurrent language model (RLM) predict the i-th word of the target sentence depending on all the previous generated words $f_{1:i-1}$, making no Markov assumption about the words dependencies in the target sentence. However, using the standard RNN architecture makes the prediction to be strongly affected by words close to f_i and weakly influenced by long-range dependencies that occur in the target language due to the nature of RNN.

The RLM models probability of a sequence of words f that occur in a language, which is denoted by $P(f)$. The equation for $P(f)$ is almost identical to Eq. 9 :

$$P(f) = \prod_{i=1}^{m} P(f_i|f_{1:i-1}) \tag{10}$$

It also contains a vocabulary V for words f_i of the language and 3 transformation matrices for input vocabulary $\mathbf{I}$, recurrent transformation $\mathbf{R}$ and output vocabulary transformation $\mathbf{O}$. Each word $f_k \in V$ is distinguished by one-hot vector $v(f_k)$. The computation then proceed as follows:

$$h_1 = g(\mathbf{I} \cdot v(f_1) + b_h) \tag{11}$$

$$h_{i+1} = g(\mathbf{R} \cdot h_i + \mathbf{I} \cdot v(f_{i+1}) + b_h) \tag{12}$$

$$o_{i+1} = \mathbf{O} \cdot h_i + b_o \tag{13}$$

and the probability distribution is obtained using the softmax function,

$$P(f_i = v|f_{1:i-1}) = \frac{\exp(o_{i,v})}{\sum_{v=1}^{V} \exp(o_{i,v})} \tag{14}$$

where g is a nonlinearity and $\mathbf{I} \cdot v(f_i)$ is a continuous representation of word f_i.

There are 2 types of conditioning architecture in RCTM using CNN, using convolutional sentence model (CSM) and convolutional n-gram model (CGM). The CSM creates sentence representation in a bottom-up manner, using n-grams representations in the sentence itself. The hierarchical structure that is created by the model act quite similar like a parse tree in a implicit way. Using this type of structure, the model is able to capture the small, local representations in the lower layers of the model and more globally in the upper layers of the model as it spans more n-grams that comprise the sentence representations. This model also offers several advantages as it does not rely on a parse tree [Grefenstette 2011, Socher 2012]. As there exist many languages for which highly accurate and reliable parsers are not available, this model can still be robustly applied. Furthermore, the distribution of translation probability is learned by the model and does not depend on the chosen parse tree.

Using the continuous representations of words in the sentence, CSM models the representation of the sentence by applying sequence of one-dimensional convolution operations. The kernel of the convolutional layers is able to learn pattern within n-grams that convey syntactic, semantic or structural information relevant for constructing the sentence representation. After several convolution operations, the sentence vector representation $\mathbf{e}$ is created at the topmost layer of the network for the source sentence e. This vector representation is then used in the RLM, after applying learned sentence transformation $\mathbf{S}$. However, this model has a bias as the RLM tend to predict

target sentences with shorter length. The sentence vector representation $\mathbf{e}$ also constraint the target words, which is counterintuitive as it often occurs that the target translation has a strong dependencies on some parts of the source sentence and less on the other parts. In order to address these aspects, [Kalchbrenner et al.] also proposes the convolutional n-gram model as another conditioning architecture.

The CGM is a truncated version of the CSM, where the n-grams representation is extracted from a specific CSM layer for a chosen value of n. Using the n-grams representation of the source sentence e, the CGM can also be inverted to obtain representation for the target sentence f with deconvolutional operation, where the length of the target sentence m is estimated using Poisson distribution. This inverted CGM can also be thought as the truncated version of the inverted CSM for sentence length m. Before the inverted CGM unfolds the n-gram representation to a target sentence, a learned translation transformation $\mathbf{T}$ is applied. The reconstructed vector for the source sentence representation is then added in an incremental manner to the corresponding hidden state h_i in the RLM to predict the word f_i in the target language. The issue that is addressed with the CGM model, where generation of the target words can now incorporate different parts of the reconstructed source sentence representation, is also later improved by [Bahdanau et al.], where they propose attention mechanism to learn soft-alignment between the source and target sentences.

While the model proposed by [Kalchbrenner et al.] works quite well in for rescoring translation hypotheses from SMT system and computing perplexity of reference translations, using CNN as encoder means that the ordering of the words are not preserved. In an almost similar manner to this approach, [Cho et al.] attempt to map source sentence to a fixed vector representation then back to the target sentence, but with 2 RNNs as encoder and decoder. The encoder RNN reads each word in the source sentence sequentially until

it reach the end of the sequence, which is marked by an end-of-sequence (EOS) symbol. The hidden state of RNN after completely reading the source sentence is then encoded to a context vector $\mathbf{c}$, which contains the summary of the whole source sentences. Consider source sentence x with length N and target sentence y with length M. The encoder is formulated as follows:

$$h_t = RNN_{enc}(h_{t-1}, \text{emb}(x_t)) \tag{15}$$

$$\mathbf{c} = \tanh(\mathbf{V}_{enc} \cdot h_N) \tag{16}$$

where $\text{emb}(x_t)$ is a continuous representation of input word at timestep t and $\mathbf{V}_{enc}$ is a learned transformation for the encoder.

On another part, the decoder RNN is a recurrent language model, conditioned on all the previous generated target words and the context vector $\mathbf{c}$. It is computed as follows, where the decoder hidden state is initialized using the context vector:

$$h'_0 = \tanh(\mathbf{V}_{dec} \cdot \mathbf{c}) \tag{17}$$

$$h'_t = RNN_{dec}(h'_{t-1}, \text{emb}(y_{t-1}), \mathbf{c}) \tag{18}$$

where $\mathbf{V}_{dec}$ is a learned transformation for the decoder. The probability distribution of target words are obtained from a softmax function applied to the output of a feedforward neural network that consists of a single intermediate layer with maxout units [Goodfellow], using the decoder hidden state, context vector and target word generated from the previous timestep as inputs.

Both of the encoder and decoder use GRU instead of LSTM as it is easier to compute and implement. The encoder and decoder components of the model are then trained in an end-to-end fashion in order to maximize the conditional log-likelihood of the target sentence given the source sen-

tence $P(y|x) = \prod_{t=1}^{M} P(y^t|y^{t-1}, y^{t-2}, ..., x)$. For words representation, one-hot encoding is used to distinguish words in the vocabulary, which are then projected twice, yielding a 100-dimensional embedding for each word.

In their paper however, they focus on integrating the RNN encoder-decoder pair for conventional phrase-based SMT system. The trained RNN encoder-decoder pair is used to rescore phrase pairs between the source and target sentences. This new score is then added to the existing phrase table and used as additional features in the log-linear model for SMT system. Furthermore, the model is also able to produce well-formed phrases in the target language independently without any influence from the actual phrase table.

Seq2Seq Model

Improvement to Seq2Seq and Recent Advances

Conclusion