Phase-Based RNN Decoder

Haitang Hu, Yating Jing
hthu, yating@jhu.edu

Abstract

In this project, a RNN Encoder-Decoder is used to rescore the translation model scores based on existing phrase table. The RNN Encoder-Decoder consists of two recursive networks which can encodes the input sequences based on history and a bilingual word embedding, and decodes corresponding representations into output sequences respectively. The two networks are jointly trained using negative log-likelihood as the objective function, with sentence level mini-batch SGD. Due to the substantial amount of time required to train the model, we only conducted experiments on small data set.

1 Introduction

In the realm of machine translation, decoding is one of the most important tasks. A decoder takes a source sentence as input, then finds its best target translation under the decoding model. The decoder is trained to maximize the conditional probability of a target (English) sentence $\mathbf{e}$ given a source (Foreign) sentence $\mathbf{f}$.

$$\begin{aligned}\mathbf{e}^* &= \arg\max_{\mathbf{e}} p(\mathbf{e} \mid \mathbf{f}) \\ &= p_{TM}(\mathbf{e} \mid \mathbf{f}) \times p_{LM}(\mathbf{e})\end{aligned} \tag{1}$$

where *TM* represents a phrased-based translation model and *LM* denotes an n-gram language model, as described in (Philipp Koehn., 2009).

Deep neural networks are becoming increasingly popular in the area of natural language processing and have shown significant success. The RNN encoder-decoder model proposed in (Cho, K. et al., 2014) maps a variable-length source sentence to a fixed-length vector and then decodes and maps the vector back to a variable-length target sentence. By jointly training two recursive neural networks, the conditional probability of the target sentence given the source sentence is then maximized. The empirical evaluation conducted in (Cho, K. et al., 2014) shows significant performance improvement.

The primary goal is to integrate the Recursive Neural Network into the phrase-based translation model, use a RNN Encoder-Decoder to evaluate the translation probability $p_{TM}(\mathbf{e} \mid \mathbf{f})$.

2 Recurrent Neural Network

2.1 Introduction to RNN

A recurrent neural network(RNN) consists of an input layer $\mathbf{x}$, hidden layer $\mathbf{h}$ and an output layer $\mathbf{y}$, which operates on a variable-length sequence $\mathbf{x} = (x_1, \ldots, x_t)$, where x_t denotes the input variable at time step t. At each time t, the hidden layer $\mathbf{h}^{(t)}$ is updated by

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, x_t) \tag{2}$$

where f could be non-linear activation function or a long short-term memory(LSTM).

Considering the behavior of the RNN, at time t, the hidden state h_t encodes all the history from $1 \rightarrow t-1$.

Given an observed sequence X(a sequence of words), each word fully depends on all the preceding words, because the current hidden status h_t could capture all history preceding current time step. So the probability of observing sequence $\mathbf{x}$ can be written as

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \ldots, x_1) \tag{3}$$

For a specific x_t, the probability of emitting the jth word in Vocabulary K is then

$$p(x_t, e_t = j \mid x_{t-1}, \ldots, 1)$$

The distribution over $\mathbf{x}$ then becomes a multinomial distribution over K possible outputs, thereby the *softmax* would be an intuitive choice. Additionaly, since x_t could get all historical dependencies on hidden state h_t, each output word is represented as a word vector using 1-of-K encoding:

$$p(x_{t,j} = 1 \mid x_{t-1}, \ldots, x_1) = \frac{\exp\{w_j h_t\}}{\sum_{k=1}^{K} \exp\{w_k h_t\}} \quad (4)$$

where K is the size of the vocabulary.

2.2 Auto Encoder-Decoder

The basic structure of the RNN Encoder-Decoder is illustrated in Figure 1. Note that in length of the input vector T is not necessarily equal to the length of the output vector T'.

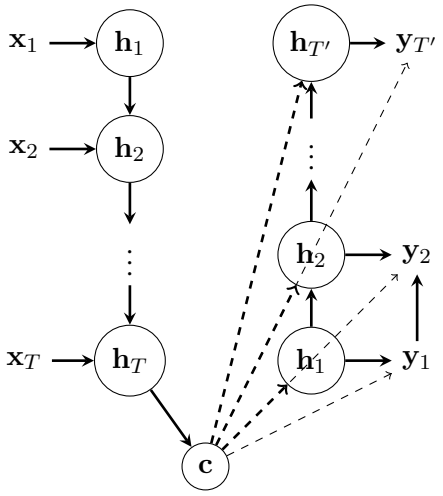

Figure 1: RNN Encoder-Decoder illustration

The **encoder** is a RNN takes in a variable length input sequence $\mathbf{x}$, and outputs a summary $\mathbf{c}$. More specifically, given an input sequence $\mathbf{x} = (x_1, \ldots, x_T)$, the update function for the hidden layer $\mathbf{h}^{(t)}$ at time step t is updated using equation (2), where f is the *sigmoid* activation function.

Afterwards, the hidden states of the RNN are summarized by a summary vector $\mathbf{c}$ of the whole input sequence, that is, the whole history of the sentence is memorized. The summary at the N^{th} step is calculated as below:

$$\mathbf{c} = \tanh(\mathbf{V}\mathbf{h}^{(N)}) \quad (5)$$

where $\mathbf{V}$ denotes the corresponding weight matrix in the encoder. Note that the summary vector $\mathbf{c}$ is produced when $t = |\mathbf{x}|$.

The **decoder** takes the trained history summary $\mathbf{c}$ produced by the encoder as the input, generates output sequence $\mathbf{y}$ from its own hidden states $\mathbf{h}^{(t-1)}$.

More specifically, the decoder RNN predicts the next word y_t given the hidden state $\mathbf{h}^{(t)}$, the preceding word y_{t-1} and the summary $\mathbf{c}$. The update function for the decoder hidden layer at time step t is:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, y_{t-1}, \mathbf{c}) \quad (6)$$

and the conditional probability of the next output word is:

$$p(y_t \mid y_{t-1}, y_{t-2}, \ldots, y_1, \mathbf{c}) = g(\mathbf{h}^{(t)}, y_{t-1}, \mathbf{c}) \quad (7)$$

where g is the *softmax* activation function.

The score used to choose the candidate translation is then the log-probability of $\mathbf{y}$:

$$\log p(\mathbf{y}) = \sum_{t=1}^{T} \log p(y_t \mid y_{t-1}, \ldots, y_1) \quad (8)$$

2.3 Training

2.3.1 Back Propagation

Neural networks can be trained by stochastic gradient descent using back-propagation(BP) algorithm as proposed in (Mikolov, T., 2012). Assume the weights for input, hidden and output layers are represented by matrices $\mathbf{U}$, $\mathbf{W}$ and $\mathbf{V}$. The weight matrices are adjusted after seeing every example. A cross entropy criterion is used to obtain gradient of an error vector in the output layer, which is then back-propagated to the hidden layer, then to the input layer. The illustration of one epoch of a simple RNN is shown in Figure 2.

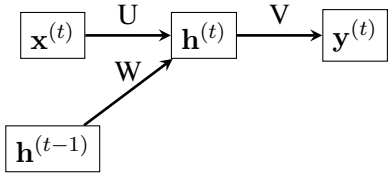

Figure 2: Illustration of one epoch of RNN

Since the optimization goal is to maximize the total log-likelihood of the target translation, which is

equivalent to minimize the negative log-likelihood. Here define the loss(error term) as the total negative log-likelihood of the current output sequence:

$$NLL^{(t)} = -\sum_{i=1}^{t} \log p(y_i \mid y_{i-1}, \ldots, y_1) \quad (9)$$

The gradient of error vector in the output layer $\mathbf{e}_o^{(t)}$ w.r.t a given weight matrix $\mathbf{m}$ at each time step is then:

$$\mathbf{e}_o^{(t)} = \frac{d}{d\mathbf{m}} NLL^{(t)} \quad (10)$$

where $\mathbf{m}$ denotes one of the weight matrices $\mathbf{U}$, $\mathbf{W}$, $\mathbf{V}$.

Therefore each element of the weight matrix $\mathbf{V}$ is adjusted as follows:

$$v_{jk}^{(t)} = v_{jk}^{(t-1)} + \eta\, h_j^{(t)} e_{ok}^{(t)} \quad (11)$$

where η is the learning rate and $e_{ok}^{(t)}$ is the error gradient of the k^{th} neuron in the output layer.

Then back propagate the error vector from the output layer to the hidden layer using the equation below:

$$\mathbf{e}_{hj}^{(t)} = \mathbf{e}_o^{(t)^T} \mathbf{V}\, h_j^{(t)} (1 - h_j^{(t)}) \quad (12)$$

Next update the weight matrix for the input layer $\mathbf{U}$:

$$u_{ij}^{(t+1)} = u_{ij}^{(t)} + \eta\, x_i^{(t)} e_{hj}^{(t)} \quad (13)$$

The update function for the recurrent weight matrix $\mathbf{W}$ is:

$$w_{lj}^{(t+1)} = w_{lj}^{(t)} + \eta\, h_l^{(t-1)} e_{hj}^{(t)} \quad (14)$$

2.3.2 Back Propagation Through time

For recurrent neural network training, the BP algorithm is not optimal, in that the network only tries to optimiaze the prediction of the next word given the previous word and previous state of the hidden layer. To extend such effect into the future, the short term information stored in the hidden layer can be replaced by some long context information.

According to (Mikolov, T., 2012), the back-propagation through time(BPTT) training algorithm can ensure that the network will learn what information to be stored in the hidden layer. A RNN with one hidden layer which is used for N time steps can be seen as a deep feedforward network with N hidden layers, which is illustrated in Figure 3, where the unfolding is applied for three time steps.

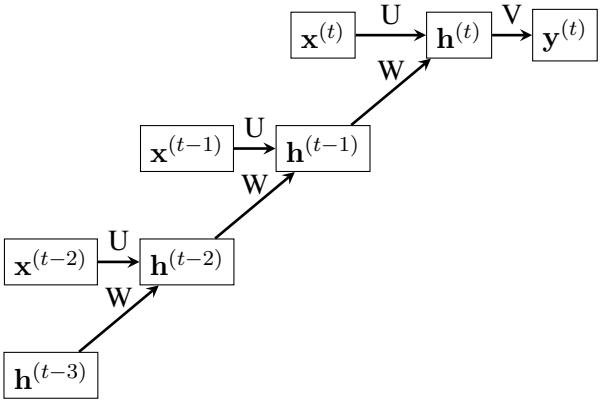

Figure 3: Illustration of one epoch of RNN as a deep feedforward network

In such RNN errors are propagated from the hidden layer $\mathbf{h}^{(t)}$ to the hidden layer from the previous time step recurrently as follows:

$$\mathbf{e}_h^{(t-\tau-1)} = d_h(\mathbf{e}_h^{(t-\tau)^T} \mathbf{W}, t - \tau - 1) \quad (15)$$

where the error vector is obtained using element-wise function $d_h()$:

$$d_{hj}(x, t) = x h_j^{(t)} (1 - h_j^{(t)}) \quad (16)$$

This unfolding process can be applied for as many time steps as many training samples were already seen, however a certain number of steps of unfolding would be sufficient since error gradients quickly vanish as back-propagated in time.

The weight matrix $\mathbf{U}$ between the input layer and hidden layer is then updated as:

$$u_{ij}^{(t+1)} = u_{ij}^{(t)} + \sum_{\tau=0}^{T} \eta\, x_i^{(t-\tau)} e_{hj}^{(t-\tau)} \quad (17)$$

where η is the learning rate. Note that $\mathbf{U}$ must be updated in one large change instead of during an incremental changing process, which can lead to instability.

The update function for the recurrent weight matrix $\mathbf{W}$ is then:

$$w_{lj}^{(t+1)} = w_{lj}^{(t)} + \sum_{\tau=0}^{T} \eta\, h_l^{(t-\tau-1)} e_{hj}^{(t-\tau)} \quad (18)$$

2.3.3 Long-Short Term Memory

In addition to the model architecture described above, a simplified version of long-short term memory(LSTM) units are adopted as the hidden units of the RNN, according to (Cho, K. et al., 2014), as depicted in Figure 4.

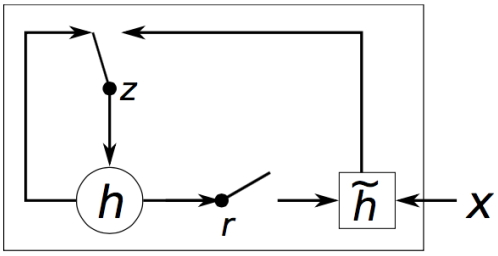

Figure 4: Illustration of hidden activation function

Namely, the reset gate allows the hidden units to drop any irrelevant information by reseting and the update gate decides how much information from previous hidden states should be passed on. When the reset gate is close to 0, the hidden state would ignore the hidden states from previous time steps, reset with current input, and learn to capture the short-term dependencies. On the other hand, those with frequently activated update gates tend to capture long-term dependencies.

More specifically, for the encoder, the activation of the j^{th} hidden unit is computed using a *reset* gate r_j and an *update* gate z_j. The two gates are computed as below:

$$\begin{aligned} r_j &= \sigma\left([\mathbf{W}_r\, e(x_t)]_j + [\mathbf{U}_r \mathbf{h}^{(t-1)}]_j\right) \\ z_j &= \sigma\left([\mathbf{W}_z\, e(x_t)]_j + [\mathbf{U}_z \mathbf{h}^{(t-1)}]_j\right) \end{aligned} \tag{19}$$

where σ is the logistic sigmoid function and $e(x_t)$ denotes the embedding of each input word x_t. Weight matrices $\mathbf{W}_r, \mathbf{U}_r, \mathbf{W}_z, \mathbf{U}_z$ are learned during training.

The actual activation of a unit h_j of the encoder is then computed as follows:

$$h_j^{(t)} = z_j h_j^{(t-1)} + (1 - z_j)\, \tilde{h}_j^{(t)} \tag{20}$$

where

$$\tilde{h}_j^{(t)} = \tanh\left([\mathbf{W} e(x_t)]_j + [\mathbf{U}(\mathbf{r} \odot \mathbf{h}^{(t-1)})]_j\right) \tag{21}$$

where $\odot$ denotes element-wise multiplication. In particular, the initial hidden state $h_j^{(t)}$ is fixed to 0.

For decoder, the *reset* gate r'_j and an *update* gate z'_j are computed as:

$$\begin{aligned} r'_j &= \sigma\left([\mathbf{W}'_r e(y_{t-1})]_j + [\mathbf{U}'_r \mathbf{h}'^{(t-1)}]_j + [\mathbf{C}_r \mathbf{c}]_j\right) \\ z'_j &= \sigma\left([\mathbf{W}'_z e(y_{t-1})]_j + [\mathbf{U}'_z \mathbf{h}'^{(t-1)}]_j + [\mathbf{C}_z \mathbf{c}]_j\right) \end{aligned} \tag{22}$$

where $e(y_t)$ denotes the embedding of each output word y_t. Weight matrices $\mathbf{W}'_r$, $\mathbf{U}'_r$, $\mathbf{W}'_z$, $\mathbf{U}'_z$, $\mathbf{C}_r$, $\mathbf{C}_z$ are learned during training.

The actual activation of a unit h_j of the decoder is then:

$$h'^{(t)}_j = z'_j h'^{(t-1)}_j + (1 - z'_j)\, \tilde{h}_j^{(t)} \tag{23}$$

where

$$\tilde{h}_j^{(t)} = \tanh\left([\mathbf{W}' e(y_{t-1})]_j + r'_j [\mathbf{U}' \mathbf{h}'^{(t-1)} \mathbf{C}\mathbf{c}]\right) \tag{24}$$

According to (Cho, K. et al., 2014), such hidden units are crucial in getting meaningful result.

3 Overall model structure

3.1 Word Embedding

We adopted word embeddings to project to and map back from a sequence of words into a continuous space vector: $W : words \rightarrow \mathcal{R}^n$. In other words, a sequence of words is mapped into some high-dimensional vector parameterized by a matrix θ, which each row representing a word:

$$W_\theta(w_n) = \theta_n \tag{25}$$

where W is initialized to have one-hot vectors for each word. It then learns to have the meaningful vectors during training.

In our model, we used a 500-dimension vector for each word. The input matrix fed into the hidden layer is thus the word embedding matrix learned by the RNN encoder-decoder.

3.2 Max-Pooling

We applied *Max-Pooling* between our *hidden layer* and *output layer*. Each two adjacent output from hidden layer will be combined to be a new output, where the maximum of those two will be chosen.

$$s_i = \max\{s_{2i-1}, s_{2i}\} \tag{26}$$

3.3 Layers Definition

The layers of the RNN encoder-decoder are illustrated in Figure 5. At each time step, the source word would be passed through the input layer of the encoder, then the hidden layer of the encoder. In particular, the encoder needs to memorize previous states of its hidden layer $\mathbf{h}^{(t-1)}$ when calculating the current hidden units.

At time step $t = |\mathbf{x}|$, the information of the entire input sentence is then passed as a summary vector from the encoder to the hidden layer $\mathbf{h}'^{(t)}$ of the decoder, then to the max-pooling layer and finally to the output layer. The functions of each layer are listed below:

a. Encoder input layer: word embedding

b. Encoder hidden layer: sigmoid function

c. Decoder hidden layer: sigmoid function

d. Decoder max-out unit: max-pooling

e. Decoder output layer: softmax function

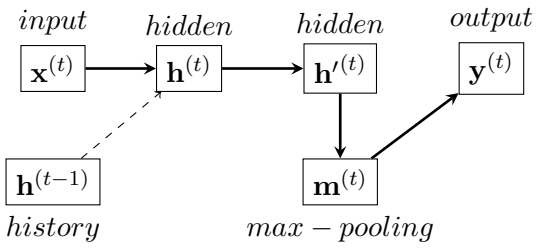

Figure 5: Layer definition

4 Implementation and Experiment

4.1 Implementation

We implemented above structure using *Theano* (Bastien et al., 2012), a scientific computation package. The source code is available at: `https://github.com/Nero-Hu/mt/tree/master/RNN`.

Note that the parameter initialization is conducted by sampling from normal distribution $\mathcal{N}(0, 0.01)$. Also, our vocabulary are selected to be the most common 1499 words in both source and target language pairs, all other words are treated to be *unknown*, which sit in the first index of vocabulary in both sides.

4.2 Experiment

The training was conducted on the Moses extracted phrase table in *homework 2*, which contains 12832 phrase pairs in total. The code runs on *GTX 690* GPU, and it takes around 14 hours to train with 50 epochs and almost 12 hours to rescore the phrase pairs.

We carried out mini-batch training on each phrase pair, and used the negative likelihood as the objective for back-propagation. The learning rate is fixed to be 0.01, and we observed a nice trend of the decrease of the negative log-likelihood, as plotted in Figure 6. The fixed learning rate could be problematic, in that we observed *unsmoothed* segments in the figure, which indicates that *adadelta* or *adaboost* should be taken into consideration.

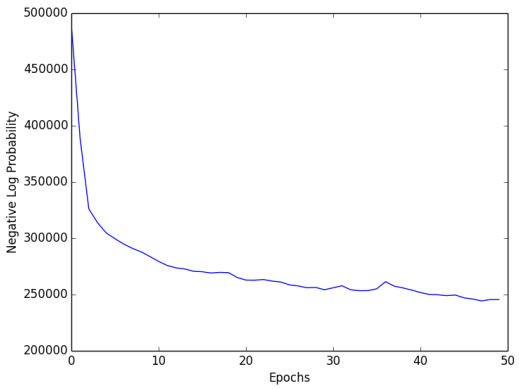

Figure 6: Negative data log-likelihood

Using the new trained phrase table and the default decoder, we obtained the worse score than given phrase table. The most reasonable explanation is that we are training on a too small data set with limited vocabulary(1500) size, also the rescoring could also need more thoughts, since the simple softmax log-probability add gives quite similar scores, and is dominated by the length of the words.

4.3 Conclusion

Due to the substantial time needed for the training process, we are only able to conduct the experiment on a limited number of phrase pairs. Nevertheless the empirial experiments indicate that this model has large potential for the decoding task.

Currently we are thinking of trying this model on

a Germen-English phrase table extracted from the Moses, which gives us larger vocabulary and more training materials. Hopefully we could find some better training resources in the future. Besides, incorpating a better neural language model into the current architecture should potentially yield a better performance.

References

Philipp Koehn. *Statistical machine translation*. 2009. Cambridge University Press.

Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078.*

Mikolov, T. (2012). Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April.*

Bastien, Frédéric and Lamblin, Pascal and Pascanu, Razvan and Bergstra, James and Goodfellow, Ian J. and Bergeron, Arnaud and Bouchard, Nicolas and Bengio, Yoshua (2012). Theano: new features and speed improvements *Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.*