MT Interim Report: Neural Word Alignemnt

Bailey Parker
Johns Hopkins University
bailey@jhu.edu

Vivian Tsai
Johns Hopkins University
viv@jhu.edu

William Watson
Johns Hopkins University
billwatson@jhu.edu

Abstract

We will discuss our intial progress made on our Neural Word Alignemnt. More specifically, we will discuss our data procurement methods and processing algorithms. In addition, we elaborate on our current experimental models, including the Dot Aligner and Bidirectional GRU Aligner. Our loss function, train/eval implmentation, and batching techiques are also discussed.

1 Introduction

2 Data Procurement and Processing

2.1 Symbol Tokenization

2.2 Number Tokenization

2.3 Proper Noun Tokenization

2.4 Lemmatization Techniques

2.5 POS Tagging

3 Model Components

3.1 Preliminary Notation

We begin by stating several operations frequently used in our discussion.

3.1.1 Hadamard Product

To perform element-wise multiplication of two matricies A and B, of equivalent dimensions, we use the hadamard product, defined as the $\circ$.

$$(A \circ B)_{ij} = A_{ij} \cdot B_{ij} \tag{1}$$

3.1.2 Sigmoid Function

The sigmoid function σ is applied element wise and defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

3.1.3 Hyperbolic Tangent Function

The hyperbolic tangent function tanh is defined as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

and is applied element-wise.

3.1.4 Variables

We define our source sequence as s, target sequence as t, and refer to the i-th source token as s_i. We refer to the j-th target token as t_j. We refer to a matrix ψ whose rows represent values related to source tokens s_i and whose columns refer to values realted to target tokens t_j. Hence, ψ is an $|s| \times |t|$ sized matrix.

3.2 Softmax and Log Softmax

The softmax function transforms a vector of values into a probability distirbution. Appliying the softmax function to an n-dimensional input tensor rescales it so that the elements of the n-dimensional output tensor lie in the range (0,1) and sum to 1.

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4}$$

For a matrix ψ, the rows correspond to source words s_i and the columns correspond to target words t_j. In addition, when we softmax with respect to the targets, i.e. $\sigma_t(\psi)$, the softmax is applied per column. If applied per row, we denote this as $\sigma_s(\psi)$.

$$\sigma_t(\psi)_{ij} = \frac{e^{\psi_{ij}}}{\sum_k e^{\psi_{kj}}} \tag{5}$$

$$\sigma_s(\psi)_{ij} = \frac{e^{\psi_{ij}}}{\sum_k e^{\psi_{ik}}} \tag{6}$$

However, it is sometimes better to work in log-space, and use the log softmax operator for numerical stability.

$$\log \sigma_t(\psi)_{ij} = \log \frac{e^{\psi_{ij}}}{\sum_k e^{\psi_{kj}}} \tag{7}$$

$$\log \sigma_s(\psi)_{ij} = \log \frac{e^{\psi_{ij}}}{\sum_k e^{\psi_{ik}}} \tag{8}$$

3.3 Word Embeddings

Word Embedding layers allow for a simple lookup table that stores embeddings of a fixed dictionary and size. More specifically, these layers are often used to store word embeddings and retrieve them using indices. The input to the embedding layer is a list of indices, and the output is the corresponding word embeddings. This allows words to be represented numerically to a set embedding dimension size, and thus can be passed onto later layers.

Word Embeddings can be formulated as a weight matrix W_e, where the vector representation of word w_i is the i-th row of the matrix, and can be represented as $W_e[w_i]$.

$$W_e = \begin{bmatrix} \longleftarrow w_1 \longrightarrow \\ \vdots \\ \longleftarrow w_i \longrightarrow \\ \vdots \\ \longleftarrow w_n \longrightarrow \end{bmatrix} \tag{9}$$

3.4 Gated Recurrent Units (GRU)

Gated Recurrent Units are a more complex formulation to recurrent layers to process sequences, and improve the vanishing gradient problem found in vanilla RNN layers while using less parameters than a Long Short-Term Memory (LSTM) layer[1]. A GRU makes use of 3 gates: r_t reset gate; z_t update gate; n_t new gate. These 3 gates allow for a blending of the hidden state with new input, and are computed as follows for each input x_t in a sequence $\mathbf{x}$:

$$\begin{aligned} r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}) \\ z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz}) \\ n_t &= \tanh(W_{in}x_t + b_{in} + r_t \circ (W_{hn}h_{t-1} + b_{hn})) \\ h_t &= (1 - z_t) \circ n_t + z_t \circ h_{t-1} \end{aligned} \tag{10}$$

where x_t is input at time t, h_{t-1} is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0, h_t is the new hidden state at time t, and σ is the sigmoid function.

3.5 Bidirectional GRU

GRUs only process sequences in a forward direction. However, for translation and in general NLP, we also care about the succeeding input. We thus introduce the Bidirectional GRU, which that runs two separate GRU layers in opposite directions and stacks the output s.t. each element has the context of elements before and after it.

Let us define GRU(*input*) as a GRU layer that outputs the hidden cells from *input* on a single forward pass. Let us also define the operator $\overrightarrow{h}$ as the tensor with elements ordered in order $[0, n]$, and $\overleftarrow{h}$ as the tensor elements ordered from $[n, 0]$, i.e. reversed.

When flipping the arrow, i.e., from $\overrightarrow{h}$ to $\overleftarrow{h}$, we define this operation as an invert/reverse function, and it flips the elements to the opposite orientation. We additionally declare $\|$ as the concatenation operator. Finally, we define $\overrightarrow{h_f}$ as the forward output, $\overleftarrow{h_b}$ as the backward output, and h_o as the final, stacked bidirectional output.

$$\begin{aligned} \overrightarrow{h_f} &= \text{GRU}(\overrightarrow{input}) \\ \overleftarrow{h_b} &= \text{GRU}(\overleftarrow{input}) \\ h_o &= \overrightarrow{h_f} \parallel \overrightarrow{h_b} \end{aligned} \tag{11}$$

From the above equations, we compute the forward output normally; compute the backwards output on the inverted input; and concatenate the forward outputs with the inverted backwards outputs. All final hidden states are provided as well. The output is of size (N, *batch*, *hidden size*$*2$) since we concatenate the outputs of the two GRU layers.

3.6 Alignment Prior

In order to learn diagonal alignemnts, we describe our formulation of the IBM Model 2 reparameterization as a custom PyTorch layer. We define the prior alignment matrix A as an $|s| \times |t|$. Given a source token s_i and target token t_j, and alignment hyperparameter λ, we deifne the alignment distortion function a_{ij}:

$$a_{ij} = -\lambda|i - j| \tag{12}$$

We set the default λ value to 4. This encodes our diagonal prior, and allows us to add in the diagonal weights to our network's weights. We allow for an optional, learnable scaling factor α to control the strength of this prior, defaulted to 0.25. Hence our IBM Model 2 module outputs the alignemnt prior matrix A:

$$A_{ij} = \alpha \cdot a_{ij} \tag{13}$$

3.7 Batch Matrix Multiplication (BMM)

To combine source and target tensors, we use Batch Matrix Multiplication. Given a matrix A of size (b, n, m) and matrix B of size (b, m, p), we perform matrix multiplication on the sub-matricies to create an output matrix O of size (b n, p).

$$O_i = A_i \times B_i \tag{14}$$

This allows us to combine the vectorized output of our network on each individual sequence and combine them into a matrix tensor of size (b, $|s|$, $|t|$). This requires the use of simple transpose functions to align the dimensions of each pairing sentence to correctly compute the BMM.

4 Model Descriptions

We use the aformentioned model components to construct complex alignment models, and provide a more through discussion on our techiques.

4.1 Dot Aligner

This is our baseline model. This model attempts to learn word embeddings to correctly give alignemnt outputs.

The intuition behind this baseline mdoel was to compute dot products between every source token s_i embedding and target token t_j embedding to generate an alignment matrix via BMM (Section 3.7), combined with the diagonal prior discussed in Section 3.6. The ouput is the alignment matrix ψ.

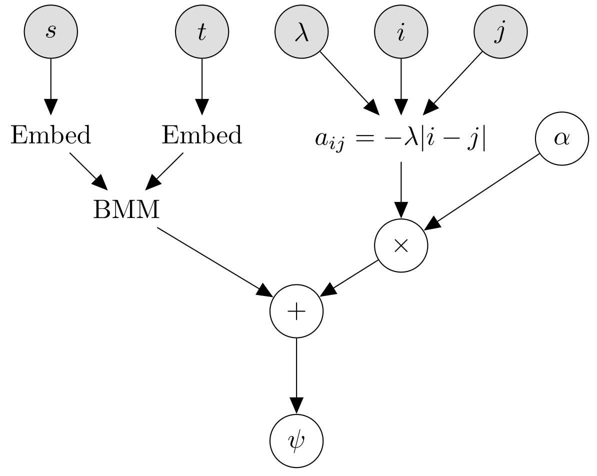

Figure 1: Model Architecture for Dot Aligner for source s, target t alignments. α is a global learnable scaling factor for the importance of alignment distribution a, and λ is the global alignment distortion parameter.

This is a very simple model that only learns the pure embeddings, and therefore we do not expect it to do as well as more complex models. The full computation graph can be seen in Figure 1.

4.2 Bidirectional GRU Aligner

An extension to the capacity of the Dot Aligner is to add a bidirectional GRU after the embedding layers. This will allow the model to consider not only forward propagation of the input sequence, but also the reverse. Therefore, our vectorized encodings of each sequence has the context of the words in its local area for context. The downstream algorithm for forward propagation is the same after the recurrent layers, as detailed in Figure 2.

4.3 Extensions

Here we discuss possible extensions to the two aformentioned models, to improve the capacity to learn alignments based upon our initial results.

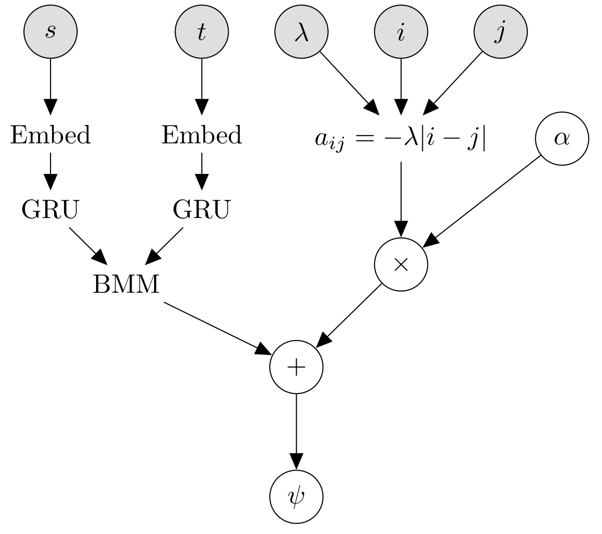

Figure 2: Model Architecture for Bidirectional GRU Aligner for source s, target t alignments. α is a global learnable scaling factor for the importance of alignment distribution a, and λ is the global alignment distortion parameter.

5 Loss Function

5.1 Supervised Loss

5.2 Unsupervised Alignment Loss

Our unsupervised loss function, to be minimized, is a 5-term equation. We define:

a_t as the target prior alignment matrix normalized per column with respect to t

a_s as the source prior alignment matrix normalized per row with respect to s

σ_s as the softmax operator applied on the rows of a matrix

σ_t as the softmax operator applied to each column of a matrix

Indexing is done under the assumption that source words s_i form rows and target words t_j form columns. The term ij is a shorthand for s_i and t_j indexing into our matrices.

$$\begin{aligned}
&Loss = \\
&\quad -\sum_{j}^{|t|} \log \left[\sum_{i}^{|s|} \exp\left(\log \sigma_s(\theta(t,s))_{ij} + \log \sigma_t(\psi)_{ij}\right) \right] \\
&\quad + \sum_{i}^{n} \sum_{j}^{m} \sigma_t(\psi)_{ij} \cdot \log \left[\frac{\sigma_t(\psi)_{ij}}{a_t(i,j)} \right] \\
&\quad -\sum_{i}^{|s|} \log \left[\sum_{j}^{|t|} \exp\left(\log \sigma_t(\theta(t,s))_{ij} + \log \sigma_s(\psi)_{ij}\right) \right] \\
&\quad + \sum_{i}^{n} \sum_{j}^{m} \sigma_s(\psi)_{ij} \cdot \log \left[\frac{\sigma_s(\psi)_{ij}}{a_s(i,j)} \right] \\
&\quad - \log \sum_{i}^{|s|} \sum_{j}^{|t|} \left[\sigma_s(\psi) \circ \sigma_t(\psi)\right]_{ij}
\end{aligned} \tag{15}$$

6 Train/Eval Implmentation

6.1 Vocabulary Building

6.2 Batching

6.3 Optimizer

6.4 Additonal Items

7 Ground Truth Alignment Results

8 Current Status

9 Future Work

References

[1] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv:1406.1078*, 2014.