MT & S2S Models

Attentional Encoder-Decoder MT

By

Varun Prashant Gangal

Language Technologies Institute
CMU

Submitted On
February 26, 2017

Model

Encoder

We implement a bidirectional encoder which encodes the sentence f through forward representations $\{\vec{h_t^f}\}_{t=1}^{t=|F|}$ and backward representations and $\{\vec{h_t^r}\}_{t=1}^{t=|F|}$. (The f and r here represent forward and reverse, while t represents the t-th word) There are different ways of combining these representations before passing them to the decoder, where they are used for state initialization and attention computation. We implement three different ways.

1. **Additive** : The combined state is generated by simply adding up the representation in either direction. $\vec{h_t} = \vec{h_t^f} + \vec{h_t^r}$
2. **Projective**: The combined state is generated by a non-linear *tanh* projection. $\vec{h_t} = tanh(H_f\vec{h_t^f} + H_r\vec{h_t^r} + b_H)$. H_f and H_r here are matrices of dimension $h \times h$, while b_H is a vector of dimension h. The dimension h here is the shared hidden dimension size of the encoders in either direction and the decoder.
3. **Concatenative**: Here, the combined state is generated by concatenating the representation in either direction. $\vec{h_t} = concat(\vec{h_t^f}; \vec{h_t^r})$. Note that the hidden state dimension of the decoder in this case needs to be twice that of the encoders.

We found that the **Additive** approach works best of the three approaches.

Initializing the Decoder State

The decoder state is initialized to the average of the final encoded states, i.e $\vec{h_0^e} = \frac{\vec{h_{|F|}^f} + \vec{h_1^r}}{2}$

Attention

All our approaches use dot product attention. We prefer this approach because it adds no new parameters to our model, thus not increasing training time and memory consumption.

Minibatching

Padding and masking for an encoder-decoder model is more complicated than that for a simple RNNLM. The canonical approach to padding-masking for a encoder-decoder model is to pad and mask both source and target sides in each minibatch example to the maximum source and maximum target lengths in the minibatch. However, this approach can three two problems.

1. Since we pass on the final state of the encoder after encoding the full source sentence to the decoder, the true final state of the encoded source will lie at different points for different training examples. Unless all source sentences in the minibatch are of one length, an additional indexing operation will be required to get the true final encoded state for each source sentence.

2. The attention mechanism attends over states corresponding to padded-masked which are not truly from the source sentence. In the case of dot-product attention, zero-masking prevents this problem from arising.

3. Combining the forward and backward encoded states becomes much simpler in the absence of source padding-masking

Instead, we adopt a slightly different style of minibatching, where each minibatch has sentences only of the same source length (but possibly different target lengths). In code, this is implemented by first hashing all training examples based on source length, and then splitting each bucket into smaller minibatches.

Hyperparameter Settings

We found that setting the size of the hidden state to 1.5-2 times the size of the embeddings works well. Our best results were obtained with embedding size 192 and hidden state size 384.

CPU vs GPU

All our models have been trained on a CPU.

Output

The simplest implementation just uses a softmax of the hidden state of the decoder at each time step t, i.e $p_t = softmax(Rh_t + b)$. A variant of this is where the context vector c_t is appended to the output vector h_t, i.e $p_t = softmax(Rconcat(h_t; c_t) + b)$. We refer to this as the $+Downstream$ method in our results

Trainer

We use the *ADAM* implementation of dynet, with its default settings. *ADAM* converges significantly faster than *SGD*, which takes roughly 2 times the number of epochs as *ADAM*.

We set a cutoff of 1 on both the English and German sentences. In other words, an English or a German word is not counted as an UNK only if it has a word frequency greater than 1. We choose a low frequency threshold considering the fact that German has a much larger vocabulary.

Search Methods

We implement the following search methods for decoding at test-time.

- **Greedy** : This method just picks the most likely next token at each step.

- **GreedyAvoidUNK**: Similar to the Greedy method, except that it picks the most likely token which is not UNK at each step. This method does not work well, since although it avoids UNKs, it leads the decoder down an erroneous context by avoiding the UNKs.
- **Beam**: Keeps the k-best sequences around. In each time-step, the existing k-best sequences are expanded, and the k-best of resulting candidates are retained.

Evaluation Measure

We use the multi-BLEU measure as computed by the *MOSES* decoder's script (from github) as our evaluation measure.

Results

Search Method	Model Type	BLEU
Greedy	Additive	17.19
GreedyAvoidUNK	Additive	16.67
Beam-4	Additive	18.46
Beam-5	Additive	18.49
Beam-6	Additive	18.41
Greedy	Additive+Downstream	22.11
Beam-4	Additive+Downstream	22.88
Beam-5	Additive+Downstream	22.84

Overall, we notice that beam search leads to an improvement of roughly 1 BLEU point over greedy search. The relationship between BLEU and beam size however, is not exactly monotonic (since we do not train our probability or loss function on BLEU itself). The highest BLEU score we attain, 22.88, is with decoding on a beam size of 4 for the *Additive+Downstream* case.

Note that we do not compute BLEU scores thoroughly for the **Projective** and **Concatenative** methods since they give poor validation perplexity (40 or above) after few epochs of training.

Code

The code for this assignment is accessible at https://github.com/vgtomahawk/MT_CMU