Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation

Authors: Yonghui Wu et. al.
Link: https://arxiv.org/abs/1609.08144

Introduction/Key points

- Three inherent weaknesses of Neural Machine Translation are responsible for this gap: its slower training and inference speed, ineffectiveness in dealing with rare words, and sometimes failure to translate all words in the source sentence.
- This model consists of a deep LSTM network with 8 encoder and 8 decoder layers using residual connections as well as attention connections from the decoder network to the encoder.
- To effectively deal with rare words, we use sub-word units (also known as “wordpieces”)
- Our LSTM RNNs have 8 layers, with residual connections between layers to encourage gradient flow.

Model Architecture

- For notation, we use bold lower case to denote vectors (e.g., $\mathbf{v}$,$\mathbf{o}_i$), bold upper case to represent matrices (e.g., $\mathbf{U}$,$\mathbf{W}$), cursive upper case to represent sets (e.g., $\mathscr{V}$,$\mathscr{T}$), capital letters to represent sequences (e.g. X, Y), and lower case to represent individual symbols in a sequence, (e.g., x_1, x_2).
- Let (X,Y) be a source and target sentence pair. Let X = $x_1,x_2,x_3, \ldots,x_M$ be the sequence of M symbols in the source sentence and let Y = $y_1, y_2, y_3, \ldots, y_N$ be the sequence of N symbols in the target sentence. The encoder is simply a function of the following form:

$$x_1,x_2, \ldots,x_M = \text{EncoderRNN}(x_1,x_2,x_3, \ldots,x_M)$$

- In this equation, $x_1,x_2, \ldots,x_M$ is a list of fixed size vectors. The number of members in the list is the same as the number of symbols in the source sentence (M in this example). Using the chain rule the conditional probability of the sequence P(Y|X) can be decomposed as:

$$\begin{aligned} P(Y|X) &= P(Y|\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \ldots, \mathbf{x_M}) \\ &= \prod_{i=1}^{N} P(y_i|y_0, y_1, y_2, \ldots, y_{i-1}; \mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \ldots, \mathbf{x_M}) \end{aligned}$$

where y_0 is a special “beginning of sentence” symbol that is prepended to every target sentence.

Model Architecture

- During inference we calculate the probability of the next symbol given the source sentence encoding and the decoded target sequence so far:

$$P(y_i | y_0, y_1, y_2, y_3, \ldots, y_{i-1}; x_1, x_2, x_3, \ldots, x_M)$$

- context a_i for the current time step is computed according to the following formulas:

$$s_t = \text{AttentionFunction}(y_{i-1}, x_t) \ \forall t,\ 1 \leq t \leq M$$

$$p_t = \exp(s_t) / \sum_{t=1}^{M} \exp(s_t) \qquad \forall t, \quad 1 \leq t \leq M$$

$$\mathbf{a}_i = \sum_{t=1}^{M} p_t . \mathbf{x}_t$$

where AttentionFunction in our implementation is a feed forward network with one hidden layer.

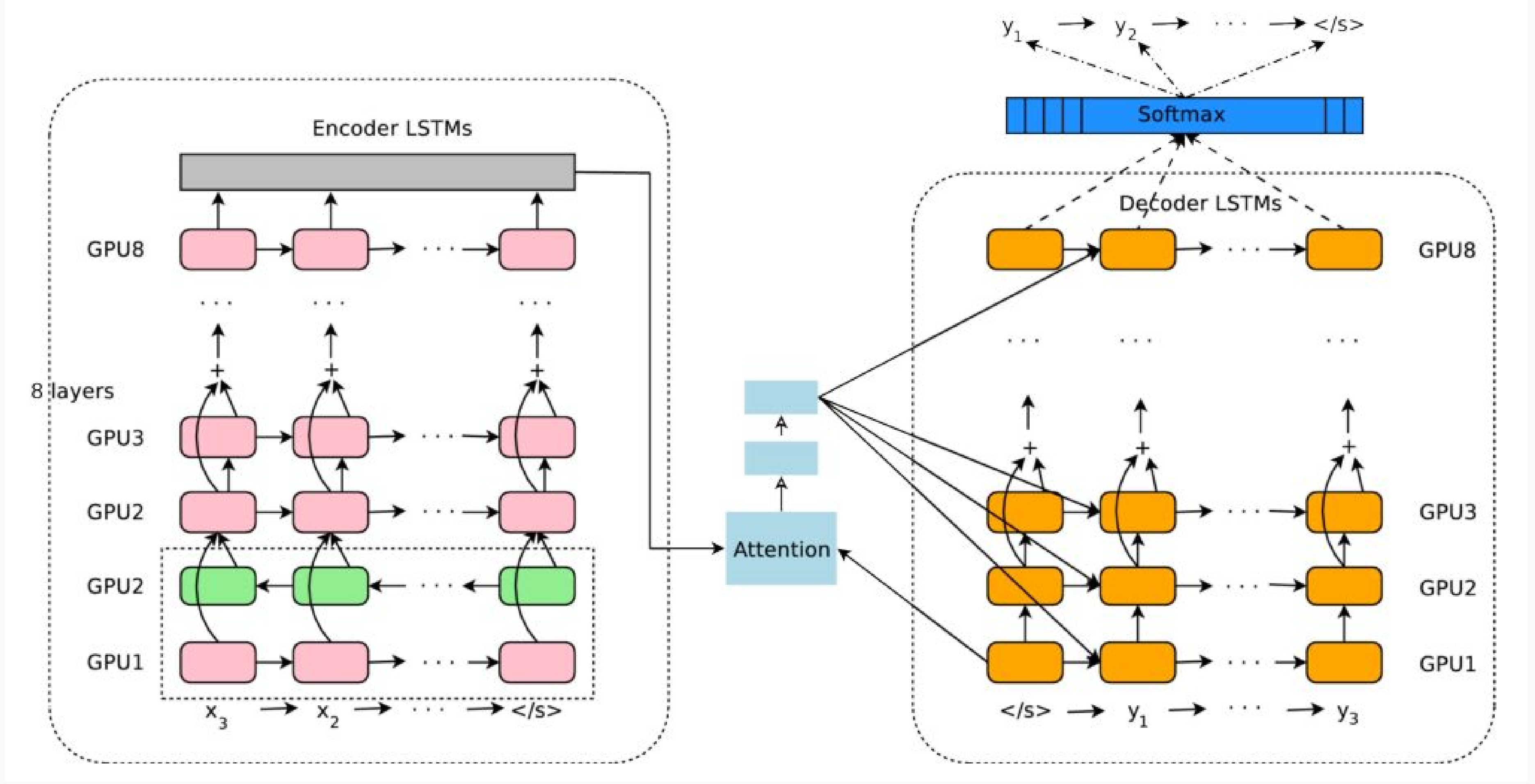

Figure 1: The model architecture of GNMT, Google's Neural Machine Translation system.

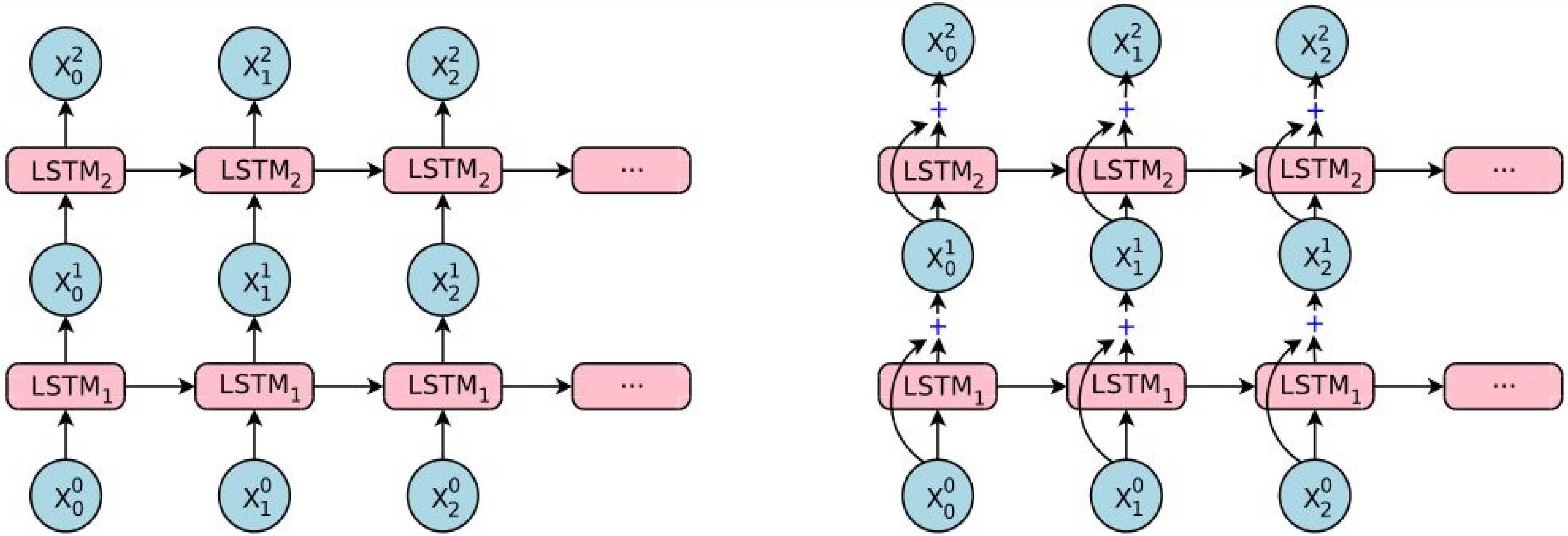

Figure 2: The difference between normal stacked LSTM and our stacked LSTM with residual connections. On the left: simple stacked LSTM layers [41]. On the right: our implementation of stacked LSTM layers with residual connections. With residual connections, input to the bottom LSTM layer ($\mathbf{x_i^0}$'s to LSTM_1) is element-wise added to the output from the bottom layer ($\mathbf{x_i^1}$'s). This sum is then fed to the top LSTM layer (LSTM_2) as the new input.

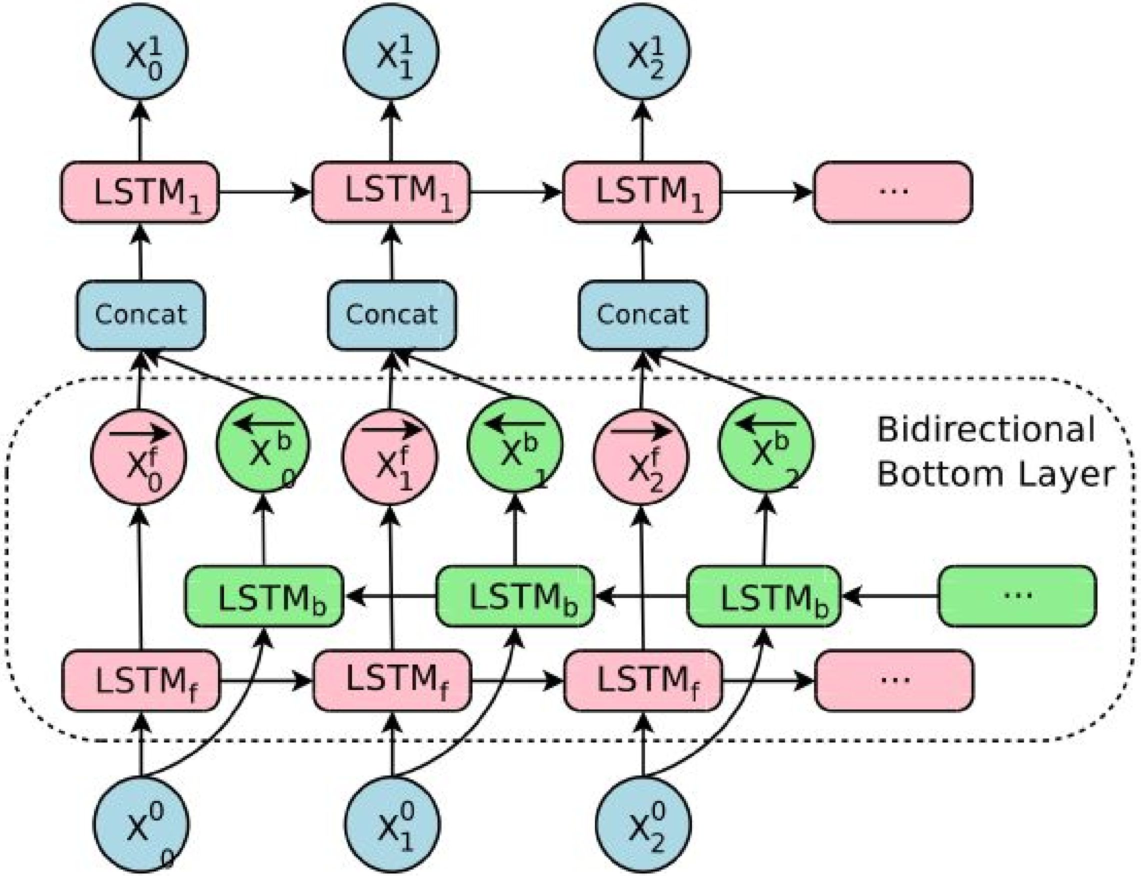

Figure 3: The structure of bi-directional connections in the first layer of the encoder. LSTM layer LSTM_f processes information from left to right, while LSTM layer LSTM_b processes information from right to left. Output from LSTM_f and LSTM_b are first concatenated and then fed to the next LSTM layer LSTM_1.

Model Parallelism

- Due to the complexity of our model, we make use of both **model parallelism** and **data parallelism** to speed up training.
- Data parallelism is straightforward: we train n model replicas concurrently using a Downpour Stochastic Gradient Descent algorithm. The n replicas all share one copy of model parameters, with each replica asynchronously updating the parameters using a combination of Adam and SGD algorithms. In our experiments, n is often around 10. Each replica works on a mini-batch of m sentence pairs at a time, which is often 128 in our experiments.
- Model parallelism is used to improve the speed of the gradient computation on each replica.
- The encoder and decoder networks are partitioned along the depth dimension and are placed on multiple GPUs, effectively running each layer on a different GPU. Since all but the first encoder layer are uni-directional, layer i+1 can start its computation before layer i is fully finished, which improves training speed.
- The softmax layer is also partitioned, with each partition responsible for a subset of symbols in the output vocabulary. Figure 1 shows more details of how partitioning is done.
- Model parallelism places certain constraints on the model architectures we can use. For example, we cannot afford to have bi-directional LSTM layers for all the encoder layers, since doing so would reduce parallelism among subsequent layers, as each layer would have to wait until both forward and backward directions of the previous layer have finished.

Training Criteria

Given a dataset of parallel text containing N input-output sequence pairs, denoted $D \equiv \{(X^{(i)}, Y^{*(i)})\}^{N}_{i=1}$, standard maximum-likelihood training aims at maximizing the sum of log probabilities of the ground-truth outputs given the corresponding inputs,

$$\mathcal{O}_{\mathrm{ML}}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log P_{\theta}(Y^{*(i)} \mid X^{(i)}) \tag{7}$$

The main problem with this objective is that it does not reflect the task reward function as measured by the BLEU score in translation. Further, this objective does not explicitly encourage a ranking among incorrect output sequences – where outputs with higher BLEU scores should still obtain higher probabilities under the model – since incorrect outputs are never observed during training. In other words, using maximum-likelihood training only, the model will not learn to be robust to errors made during decoding since they are never observed, which is quite a mismatch between the training and testing procedure.