Implementing a Neural Machine Translation(NMT) system using PyTorch

Thibatsane Ivan Mahlelebe

February 20, 2024

1 Introduction

This paper discusses implementation details of Recurrent Neural Networks(RNNs) in machine translation. Before RNNs, window-based models were widely adopted, but struggled to capture long-range dependencies, resulting in suboptimal translations due to incomplete context understanding. This paper thus presents how a sequence-to-sequence model with attention mechanism is used for this task. The adopted approach involves training the system using the following procedure:

- Input a source sequence.
- Compute its embeddings.
- Pass them through a convolutional and bidirection LSTM layers for encoding.
- Initialize the decoder with the final hidden state of the encoder
- Compute Attention Scores and Attention Distribution
- Then finally use a softmax layer to generate the Attention Output

The usual cross-entropy loss function is used to train the network, with parameters being optimized with Adams algorithm. The overall aim is to develop a comprehensive understanding of the components and training procedure of NMT systems.

Disclaimer

The bulk of code used for this paper was acquired as part of the assignment package and was not written by the author. However, it is in view of the author that, while they felt hand-held for this assignment, the overall experience was positive and informative. The need to modify and debug the code required a holistic understanding of the entire codebase, leading to acquisition of valuable insights into software engineering principles like refactoring. In addition, it was also a great experience to help them learn how to navigate a relatively complex codebase that was written by other people, which is a valuable skill for working in the industry.
Code implemented by the author is discussed below.

Source Code

The Source Code can be found on the author's Github Repo: 03-natural-machine-translator

2 Model Description - During Training

This section only discusses the author's contributions to the codebase and does not explain other elements of the code, unless necessary.

2.1 Model Architecture

Describe the architecture of your NLP model(s), including the choice of neural network architecture, optimization algorithms, and hyperparameters.

2.2 pad_sents in utils.py

This first task was a rather easy implementation of a funtion that simply ensure that word sequences(sentences) in a given batch are of the same length, so we simply find the longest sequence's length, then ensure all other sequences are of this length by padding.

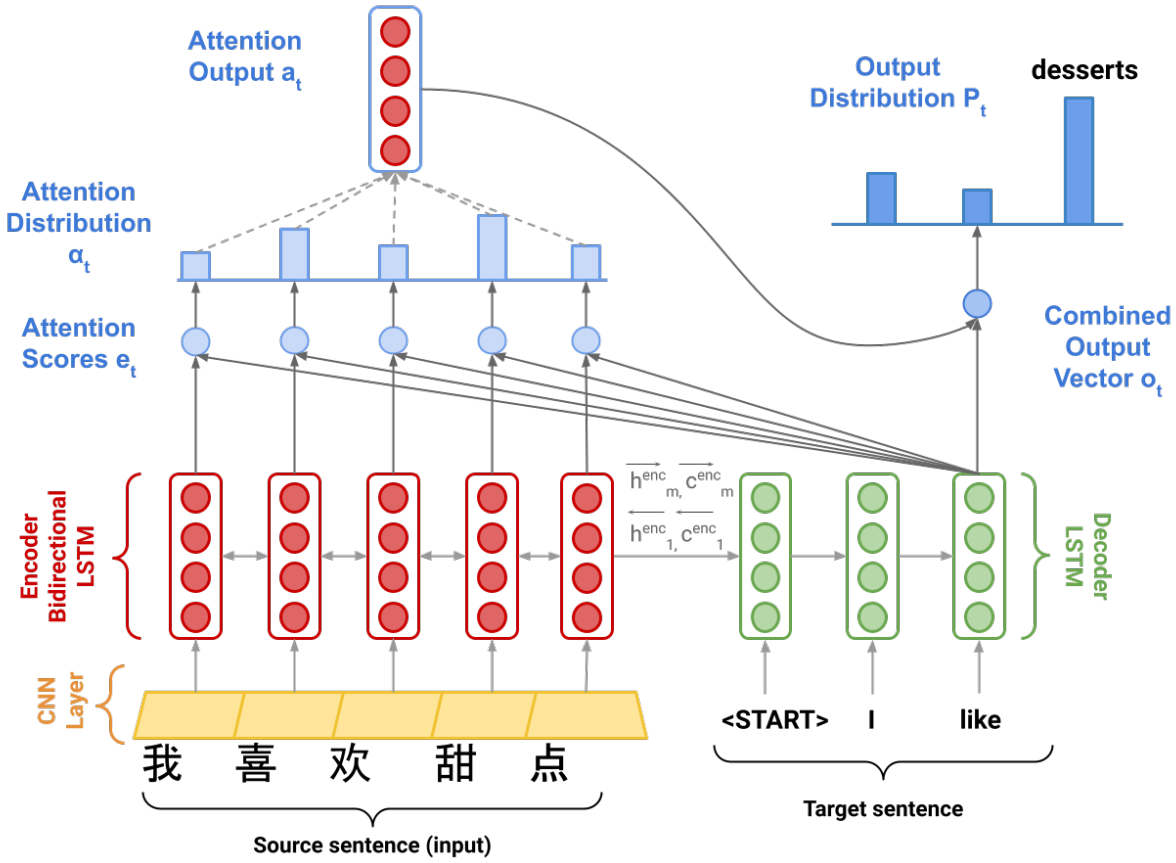

Figure 1: Model Architecture: Seq2Seq Model with Multiplicative Attention, shown on the third step of the decoder.

2.3 __init__ in model_embeddings.py

In order to train our model for translation, we need to input pairwise parallel sentences of corresponding source language and target language sentences. However, since our model cannot understand natural language, let alone deduce its meaning, we need to convert words into *word vectors*. These are real-valued multi-dimensional vectors(even though we'll use tensors in the implementation) whose values are meant to provide semantic context of words they represent.
PyTorch already has a pre-built module for this operation, therefore it was used. It simply takes indices of these words and creates weights for each. At this stage, the model has not yet training, so PyTorch simply initializes these weights random values from a Normal Distribution.

2.4 __init__ in nmt_model.py

Once the training data has been converted into *word embeddings*, we can now use these embeddings together with other building-blocks to initialize our model according to its architecture demonstrated by Figure 1 above. The model was therefore initialized with with the following variables:

2.4.1 Convolutional Neural Network Layer

At this point, CNNs have not yet been studied in depth, thus, the paper cannot fully explain the design decision to input word embeddings into a Convolutional layer.

2.4.2 Encoder

A neural network architecture that we have actually studied is a recurrent. This is an improvement over window-based models as it can take a sequence of any arbitrary length. The core idea here is to input each token of a sequence into a hidden layer, predict the next word, then multiply the next hidden layer with the same learnable weights $\mathbf{W}$. We can summarise this operation with the equation:

$$\mathbf{h_i} = \text{ReLU}(\mathbf{W_h} \cdot \mathbf{h^{t-1}} + \mathbf{W_e} \cdot \mathbf{e}^t + b_1)$$

The *encoder* takes the following inputs:

- **embed_size** (int) - Because the word embeddings input will be of dimesions (1, *embed_size*).
- **hidden_size** (int) - Number of neurons(outputs) in each hidden state.
- **bias** (bool) - True since we'd like to have bias terms.
- **bidirectional** (bool) - True because we would like our encoder to be bidirectional, so that it can capture the full context of a sequence by multiplying the weights in both directions i.e. add a backward layer and concatenate it with a forward layer.

2.4.3 Decoder

Using the context defined by the encoder above, the *Decoder* is then responsible for converting it into a an actual target sequence. This is the "cell version" of the LSTM which enables the decoder to store long-term information(for long sequences) in order to work around the vanishing gradients problem. This is a rather complex idea which I will summarize by just saying, if the gradients(during training) are too small, then far away weights become less useful to the prediction of the next word due to chained derivatives computed during parameter optimization. So the idea of a cell resolves this issue by using *input*, *forget* and *output* gates. These gates will then be used to determine which information should be forgotten, input into and output from the cell that will be passed between decoder hidden states.

2.4.4 Dropout

This is a regularization method used to help neural network generalize predictions better even in the face of overfitting. Just like in bagging, in neural networks, dropout involves randomly deactivating non-output units within a neural network by multiplying them by zero during training. Amongst other reasons, this idea mitigates the risk of overfitting by introducing variation and preventing units from relying too heavily on specific features or patterns in the data.
So ultimately, the drop-out rate is the rate at which these units are randomly removed.

2.5 step in nmt_model.py

This function is the actual implementation of how Multiplicative Attention is incorporated in the architecture. The idea of Attention solves the *bottle-neck problem* introduced by simple LSTMs where the context from the encoder is aggregated(for lack of a better word) on the last hidden state of the encoder. This is solved by calculating Attention Scores from each encoder hidden state such that decoder hidden states have direct access of information from encoder hidden states.
All Attention mechanism have the same idea and differ by how much they compute Attention Scores. We used a Reduced-rank Multiplicative Attention(RMA), which is an improvement over the Multiplicative Attention(MA). In in MA, we use a learnable weights matrix $\mathbf{W}$ which captures the parts of encoder hidden states $\mathbf{h}$ and decoder hidden states $\mathbf{s}$ to pay attention to when calculating similarities(hence, attention scores), so Attention is calculated by:

$$\mathbf{e}_i = \mathbf{s}^T \cdot \mathbf{W} \cdot \mathbf{h_i}, \text{ where } \mathbf{h} \in \mathbb{R}^{d_1} \text{ and } \mathbf{s} \in \mathbb{R}^{d_2}$$

which is costly due to a huge number of parameters(they are $d_1 \times d_2$). RMA resolves the issue by decomposing $\mathbf{W}$ as follows:

$$\begin{aligned}\mathbf{e}_i &= \mathbf{s}^T \cdot \mathbf{W} \cdot \mathbf{h_i}, \text{ where } \mathbf{h} \in \mathbb{R}^{d_1} \text{ and } \mathbf{s} \in \mathbb{R}^{d_2} \\ &= \mathbf{s}^T \cdot (\mathbf{U}^T \cdot \mathbf{V}) \cdot \mathbf{h_i}, \text{ where } \mathbf{U} \in \mathbb{U}^{k \times d_1} \text{ and } \mathbf{V} \in \mathbb{R}^{k \times d_2} \\ &= (\mathbf{U} \cdot \mathbf{s})^T \cdot (\mathbf{V} \cdot h_i)\end{aligned}$$

where the dimensions will now be $d_1 \times k + d_2 \times k = k(d_1 + d_2)$ where $k < \min(d_1, d_2)$ meaning we'll have far less parameters for our model to learn. The Attention distribution α_t will then be calculated as:

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^N, \text{ and the Attention Output } \mathbf{a} \text{ will be:}$$
$$\mathbf{a} = \sum_{i=1}^{N} \alpha_i \cdot \mathbf{h}_i \in \mathbb{R}^{d_1}$$

3 Conclusion

In closing, this is a completed the assignment from a course [2] whose lecture videos are available on YouTube [3]. I've also used the book called *Deep Learning* [1] as an additional course material. Through it, I have grasped comprehensive understanding of NMT systems and the practical aspects of RNNs, attention mechanims and dropout regularization in PyTorch and the insights gained from this project would serve as a solid foundation for further exploration and experimentation in the field of deep learning and natural language processing.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[2] Stanford University. Cs224n: Natural language processing with deep learning. https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1234/.

[3] Stanford University. Cs224n: Natural language processing with deep learning (stanford). https://www.youtube.com/playlist?list=PLoROMvodv4rMFqRtEuo6SGjY4XbRIVRd4.