Semantic Parsing

Jake Imyak

Nov 5th, 2021

The core problem for this project we are attempting the solve is how can we use a Seq2Seq model to answer natural language questions about geography.

1 Part 1: Basic Encoder-Decoder

To run the basic model: python main.py and you can specify the number of epoch with –epoch=NUM

For the model of the Decoder, a single cell of an LSTM was created. This was initialized with an input size, hidden size, and the number of classes. Then, an LSTM from Pytorch is then initialized along with the linear layer and initial weights are created as described in the TA session. Then within the forward method, the embedded input is packed together using the pack_padded_sequence method. Next using this packed embedding representation and the hidden input, we pass this through the LSTM. From that, we can get the last hidden state and pass part of that tuple through the linear layer to create a linear transformation. Thus, we are essentially taking an output token and predicting the next token similar in the Seq2Seq slide deck. Also I referenced The Model section of the Sequence to Sequence Learning with Neural Networks by Sutskever.

To train the model, first we initialize an embedding layer for both the input output and the encoder decoder. The dimensions of these layers were varied through the experiments. The hyper-parameters were as followed: the Adam optimizer and CrossEntropyLoss provided by Pytorch. The training loop worked as follows: loop over epochs, randomize examples, initialize a loss tensor, zero grad our models, create tensors for input tokens and length of sentence in batch, get encoder outputs and state, get embedded representations of the words. Then using the encoded hidden part and embedded representation of the first layer, we were able to move forward by looping through all of the labels for the i-th training example until we find the gold stopping point through the decoder network computing the loss for each cell. Then back propagate to update the hyper-parameters. Each training epoch takes about 15 seconds.

Inference worked very similar to the last part of the besides back propagating through the network. Essential it is looping through all the tokens, gathering the different representations as specified above. Then we loop through until we either reach the EOS token or we reach the max prediction length (65)

and store all of the output tokens in a list by getting the output from the decoder and moving forward through the network. This is used to make predictions.

The results can be seen from the table. Note the Dimension is (Input/Output Embedding Layer Dim, Encoder/Decoder Embedding Layer):

LR	Epochs	Dimension	Token acc	Denote matches
.001	10	(100,200)	0.675	0.100
.001	20	(100,200)	0.722	0.233
.001	30	(100,200)	0.801	0.375
.005	10	(100,200)	0.083	0.108
.001	10	(200,400)	0.729	0.175
.001	20	(200,400)	0.778	0.392
.001	30	(200,400)	0.784	0.442

2 Part 2: Attention

To run the attention model: python main_atttention_mech.py and you can specify the number of epoch with –epoch=NUM

The attention mechanism for the decoder is implemented similar to as the Jia and Liang 2016 paper (referencing the findings from the Luong paper) that was given on the course website. Essentially, the initialization of the decoder and weights are but rather we are using two linear transformations rather than one. Further, we are using a bidirectional model in the case which difference from the original decoder. The forward method works as follows: we pack the embedded representation as before and get the output and hidden state using the torch RNN as before. Then, we utilize the encoder and decoder hidden states to get the not normalized attention scores. Then we pass these not normalized scores through a linear layer that changes the dimensions linear layer to the hidden_size and hidden_size * 2 and perform a matrix multiplication with outputs from the encoder. Next we softmax over these attention scores to get the attention weights. This is similar to the slide where the outputs from the encoder are passed to the LSTM. Then to get the context vector: we multiply the attention weights and hidden representation. Then we leverage attention weights and the decoder hidden representation concatenated together passed through a tanh function. Finally we pass this through another linear transformation such that it is the dimension of output vocabulary. This is subbed out for the decoder without attention in both the training and inference phase. The training time took approximately 21 seconds. The results are as follows:

LR	Epochs	Dimension	Token acc	Denote matches
.001	10	(100,200)	0.716	0.433
.001	20	(100,200)	0.811	0.542
.001	30	(100,200)	0.782	0.550
.005	10	(100,200)	0.649	0.317
.001	10	(200,400)	0.726	0.400
.001	20	(200,400)	0.817	0.575
.001	30	(200,400)	0.731	0.483

Next steps and things I wish I would have done are the following: I would have loved to implement batching, random teacher forcing, and using something as Word2Vec as the representation for the words. I think all three of those would have greatly increased my results. A lot of my time was spent trying to debug different dimension errors or random Pytorch issues so I did not have time for these. I found my results to be particularly interesting. I tried to maintain a simple model to keep it faster and because it was what I comprehended from the slides and I was content with how well it did on the dataset. Increasing the learning rate in both cases had a negative effect so I think .001 is the sweet spot. For the dimensions, increasing it also made the program take long and the results weren't extremely better so I think the slightly smaller network was worth it in a semi time constraint project like this. Also, increasing the number of epochs helped improve scores a lot, however, it is important to note the variation with the randomization of the dataset. The scores were sometimes significantly different so it is hard to note the best amount of epochs to run. The variation is due to the randomness in terms of pulling data in.

References:

Jia, Robin , Liang, Percy (2016) Data Recombination for Neural Semantic Parsing https://arxiv.org/pdf/1606.03622.pdf

Luong, Minh-Thang, Pham, Hieu, Manning, Christopher D. (2015) Effective Approaches to Attention-based Neural Machine Translation https://arxiv.org/pdf/1508.04025.pdf

Ilya Sutskever Oriol Vinyals Quoc V. Le. Sequence to Sequence Learning with Neural Networks https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf