CAP 6610 – MACHINE TRANSLATION PROJECT FINAL PROJECT REPORT

JAMES DIFFENDERFER

Preface

- The goal of our project was to perform machine translation from English to French. We studied various methods for performing the embedding of our dictionaries in a vector space and different models that have provided successful results in statistical machine translation.
- For the machine translation portion of the project, we decided to use an encoder-decoder model for machine translation where each node in the network used a long short-term memory (LSTM) architecture.
- This report will outline a sample of derivations required for the model, how we implemented the encoder-decoder architecture, how the model was trained, a discussion of our results, and ideas that may lead to improvements.

Model for the Project

- Based on several papers we read this semester presenting good results, we chose to implement an encoder-decoder model for statistical machine translation.
- Our model was largely based on the one presented in detail in the lecture notes by Kyunghyun Cho (assistant professor at the Courant Institute of Mathematical Sciences).

Derivations

- The forward propagation for the model was very straightforward as each node in the encoder and decoder only needed to compute the following values (which need to be presented here for in order for the partial derivatives formulas we derived to make sense):

$$
\begin{aligned}
h_t &= o_t \odot \tanh(c_t) && \text{(Hidden output at time t)}\\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{(Cell state at time t)}\\
o_t &= \sigma\left(W_o u_t + b_o\right) && \text{(Output gate at time t)}\\
\tilde{c}_t &= \tanh\left(W_c u_t + b_c\right) && \text{(Internal Cell state at time t)}\\
i_t &= \sigma\left(W_i u_t + b_i\right) && \text{(Input gate at time t)}\\
f_t &= \sigma\left(W_f u_t + b_f\right), && \text{(Forget gate at time t)}
\end{aligned}
$$

where $\odot$ represents component-wise multiplication, $u_t = [z_{t-1}; x_t]^\intercal$ in the encoder model, and $u_t = [z_{t-1}; [x_t, y_{t-1}]]^\intercal$ in the decoder model.

- Additionally, in the decoder model, we needed to compute the values

$$
\begin{aligned}
E_t &= -\frac{1}{N}\log \mu_{k,t} && \text{(Log likelihood at a single timestep)}\\
E &= \sum_{n=1}^{N}\sum_{t=0}^{K} E_t && \text{(Log likelihood Function)}\\
\mu_{k,t} &= \frac{e^{y_k^\intercal h_t + b_k}}{\sum_{i=1}^{K} e^{y_i^\intercal h_t + b_i}}, && \text{(Softmax function)}
\end{aligned}
$$

  where N represents the total number of samples in the data set and K is the size of the target dictionary.
- The backward propagation of the decoder required the partial derivatives of the log likelihood function with respect to the weights and bias terms. A result for the derivation of the partial derivatives of the log likelihood at timestep t with respect to the weights and bias terms for the forget gate of the decoder are provided below:

$$
\begin{aligned}
\frac{\partial E_t}{\partial W_{zf}} &= (1-\mu_{k,t})\Bigg[h_t(1-o_t)W_{ho}\frac{\partial h_{t-1}}{\partial W_{zf}} + o_t\left(1-\tanh(c_t)^2\right)\Bigg(c_{t-1}f_t(1-f_t)h_{t-1}\\
&\quad + \left[c_{t-1}f_t(1-f_t)W_{zf} + \tilde{c}_t i_t(1-i_t)W_{zi} + i_t(1-\tilde{c}_t^2)W_{zc}\right]\frac{\partial h_{t-1}}{\partial W_{zf}}\\
&\quad + \sum_{k=1}^{t-1} f_{k+1}\left[c_{k-1}f_k(1-f_k)W_{zf} + \tilde{c}_k i_k(1-i_k)W_{zi} + i_k(1-\tilde{c}_k^2)W_{zc}\right]\frac{\partial h_{k-1}}{\partial W_{zf}}\Bigg)\Bigg]\\
\frac{\partial E_t}{\partial W_{xf}} &= (1-\mu_{k,t})x_t c_{t-1}f_t(1-f_t)o_t\left(1-\tanh(c_t)^2\right)\\
\frac{\partial E_t}{\partial b_f} &= (1-\mu_{k,t})c_{t-1}f_t(1-f_t)o_t\left(1-\tanh(c_t)^2\right)
\end{aligned}
$$

Implementation and Training

- The encoder-decoder model was coded using python. The code ended up being approximately 2000 lines consisting mainly of a few classes:
  - **Parm**: This class contained parameters and update routines for the parameters used in the LSTM nodes and the encoder or decoder. An example of some parameters and update functions implemented in this class are:
    - Weights and bias terms for each gate in the LSTM node and their partial derivatives at each timestep
    - Arrays used to store the states of the gates at each timestep
    - Routine to update the partial derivatives using the formulas derived in the previous section
  - **Encoder** and **Decoder**:
    - These classes each contained a *Parm* class which was used to store and update the weights for the encoder and decoder
    - Since the encoder and decoder have a similar structure but some differences between the forward and backward propagation updates, using the *Parm* class I wrote in the encoder and decoder saved a lot of time and only required a few extra functions to implement the different updates in the forward and backward propagations.

- The model was trained using a subset of the French-English parallel corpus from European Parliament Proceeding Parallel Corpus 1996-2011 which consisted of 2,190,579 sentences. We used approximately 90% of these sentences for training the model.
- In order to train the model, the weights and bias terms were initialized to be values uniformly distributed between -0.08 and 0.08.
- In order to update the weights, we used a batch stochastic gradient descent with a batch size of 10.
- After experimenting with various dictionary sizes, we chose to use a dictionary of size 100000 for the source (English) and target (French) languages. These sizes were chosen in order to prevent too many unknown words from arising during the training process.
- The model was trained three separate times for 48 hours using various initial weights and bias terms generated using the above method. Afterwards it was tested on sentences that were not included in the original training set.

Results and Conclusions

- The weights that were trained for 48 hours did not perform very well when presented with new source sentences. We think this is in part to the following issues:
  - The model was not trained long enough to produce suitable weights.
    - This was in part due to the fact that it took a considerable amount of time to perform unit tests on all of the functions in the code to ensure that each part of the program was working appropriately. This did not leave much time for training to obtain good weights.
    - For reference, in one of the papers we read which presented good results, they trained their weights for five days and their cluster of computers was performing much faster than the single desktop computer we had at our disposal.
  - The objective function was getting stuck at a local minimimum during the training procedure.
    - We believe that this may be another cause of our poor performance since we observed better performance when the weights were updated using very small training sets and then tested on sentences with similar words. However, these weights were not used as they performed very poorly on the majority of the testing set.
- I believe that the two main improvements that could be made to our project are:
  - Implement or develop a routine that helps prevent the objective function from getting stuck at a local minimum.
    - We read about and discussed several methods that could be used for the update of the weights at each iteration. I believe that implementing one of these methods could help us overcome the problem we seemed to encounter of getting stuck at a local minimum that was not producing desirable results. We originally chose a batch stochastic gradient descent since it was very easy to code.
  - Increase the amount of training time.
    - Based on many tests it seems that the code is performing the minimization procedure correctly so we believe that given more time to train the weights would eventually begin to perform relatively well.