CS 295: Statistical NLP: Winter 2018

Homework 4: Neural Machine Translation

Sameer Singh (and Robert L. Logan)
http://sameersingh.org/courses/statnlp/wi18/

One of the most widespread and public-facing applications of natural language processing is machine translation. It has gained a lot of attention in recent years, both infamously for its lack of ability to understand the nuance in human communications, and for near human-level performance achieved using neural models. In this homework, we will be looking at neural machine translation from modern English to Shakespearean English. The submissions are due by midnight on **March 18, 2018**.

1 Task: Neural Machine Translation

Machine translation is the task of designing a model which automatically translates text or speech from one language (which we refer to as the *source* language S) to another language (which we refer to as the *target* language T). In order to train such a model, we assume we have access to a *parallel corpus* $C = \{< \mathbf{s}, \mathbf{t} > | \mathbf{s} \in S, \mathbf{t} \in T\}$ of pairs of sentences in each language which have equivalent meanings. Our goal is then to find a model which maximizes the probability $P(\mathbf{t}|\mathbf{s})$ for the sentences in this corpus.

The sub-field of neural machine translation parameterizes this probability distribution as a neural network. I will briefly describe the neural machine translation model in this section. For more details you should refer to Sutskever et al.'s paper on the topic available here: https://arxiv.org/pdf/1409.3215.pdf.

1.1 Sequence-to-sequence Models

At its core, this problem entails mapping a sequence of inputs (words in the source language) to a sequence of outputs (words in the target language). As we've discussed in class, recurrent neural networks (RNNs) are effective at working with this kind of sequential data. One difficulty that arises in machine translation is that there is not a one-to-one correspondence between the input and output sequence. That is, the sequences are typically of different lengths and the word alignment may be non-trivial (e.g. words that are direct translations of each other may not occur in the same order).

To address these issues, we will use a more flexible architecture known as a sequence-to-sequence model. This model is composed of two parts, an *encoder* and a *decoder*, both of which are RNNs. The *encoder* takes as its input the sequence of words in the source language, and outputs the final hidden states of the RNN layers. The *decoder* is similar, except it also has an additional fully connected layer (w/ softmax activation) used to define a probability distribution over the next words in the translation. In this way, the *decoder* essentially functions as a neural language model for the target language. The key difference is that the *decoder* uses the output of the *encoder* as its initial hidden state, as opposed to a vector of zeros.

1.2 Data and Source Code

I have released the initial source code, available at https://github.com/sameersingh/uci-statnlp/tree/master/hw4, and the data archive available on Canvas. You will need to uncompress the archive, and put it in the `data/` folder for the code to work. The source code contains the following:

- **`config.yaml`**: This file lists all of the hyperparameters used by the model, as well as the location of the data used by the model. It will be copied to the folder where your model checkpoints are saved during training. This is done to prevent you from inadvertently making changes to your hyper-parameters in the middle of training, as well as to remind you of the exact configurations used for the experiments you perform (which is useful if you train many different models). If you introduce additional hyper-parameters to the model (e.g. dropout-rate, number of layers, etc.) I recommend keeping track of them in this file.
- **`model.py`**: This code provides implementations of the `Encoder` and `Decoder` modules. The basic structure is similar to the examples in the recurrent neural network tutorial notebook[1], however there are a few key differences:

[1] https://github.com/sameersingh/uci-statnlp/tree/master/tutorials/rnn_examples.ipynb

1. The encoder and decoder are separate models. Both consist of an embedding layer which is fed into a RNN. For the encoder, we are only concerned with using the hidden states from the RNN so the model only contains these two components. For the decoder, we want to define a probability distribution over the words in the target language so the model also includes a fully connected layer with a softmax activation function.
2. Both modules are designed to take inputs one-by-one instead of in batches. This prevents having to sort/pack/mask the source and target sentences, which can be complicated as the source and target sentence are not guaranteed to have the same length.

Any architectural changes you make should be done in this file.

- **`train.py`**: This script is used to train your model. Example usage:

```
    python train.py --config config.yaml
```

You will probably not need to modify this file unless you decide to use a training method other than teacher-forcing, or an optimizer other than Adam.

- **`evaluation.py`**: This script will measure the BLEU score of your model on the test set, as well as provide a couple example translations for qualitative evaluation. Example usage:

```
    python evaluate.py --config config.yaml
```

- **`utils/data.py`**: Defines the `ShakespeareDataset` class. Used to load the parallel corpora into tensors.
- **`utils/vocab.py`**: Defines the `Vocab` class. Used to associate an integer id to the words encountered in the corpora.

Details about what you need to implement is in the sections below.

2 What to Submit?

This assignment is designed to be extremely open-ended. The only task is to improve upon the provided baseline model. Accordingly, there are a variety of things you can try. Here are a couple suggestions:

Trivial modifications:
- Use an LSTM/GRU cell in place of the RNN.
- Make the encoder bidirectional.

Easy modifications:
- Vary number of hidden layers/ units in the model. Perform experiments to determine the optimal amount.
- Add a regularization penalty to the loss, and determine the optimal regularization strength.
- Apply dropout between layers. Perform experiments to determine the optimal dropout rate.

Medium modifications:
- Use attention in the `Decoder`. There are a lot of different variants you can try, this survey gives a good overview: `https://nlp.stanford.edu/pubs/emnlp15_attn.pdf`.
- In the `Encoder`, use frozen word embeddings pretrained on a large corpus. **Hint:** the source language is modern English, we've already provided numerous corpora in previous homeworks which could be used to learn the embeddings. Alternatively, you could use pretrained word2vec[2] or GloVe[3] embeddings.

Hard modifications:
- Derive word embeddings by performing character-level convolutions. For more detail, read: `https://arxiv.org/abs/1508.06615`.
- Improve decoding by using beam search, and determine the optimal beam-width.

Prepare and submit a single write-up (**PDF, maximum 5 pages**) and your `model.py` and any other modified files (compressed in a single `zip` or `tar.gz` file; we will not be compiling or executing it, nor will we be evaluating the quality of the code) to Canvas. Your final score will be based on the following criteria.

2.1 Difficulty of Approach (50 points)

As the suggestions above indicate, some modifications may be much more difficult to implement than others. In order to encourage students to pursue more difficult approaches, points will be assigned based on the amount of effort put into the work. We assume as a baseline that all students will make the trivial modifications to the model. After that basic rubric is: 1 hard modification = 2 medium modifications = 3 easy modifications. To make this more clear, full credit would be earned in each of the following scenarios:

[2] `https://code.google.com/archive/p/word2vec/`
[3] `https://nlp.stanford.edu/projects/glove/`

- Student A: Uses LSTM cells in all experiments. Performs grid search to determine the optimal number of hidden layers / hidden units. After that the student then determines the optimal regularization strength.
- Student B: Implements an attention mechanism in the decoder, then performs validation to determine the optimal number of hidden layers to use and whether LSTM or GRU units perform better.
- Student C: Uses GRU cells. Implements character-level convolutions to obtain word embeddings.

2.2 Quality of the Write-Up (50 points)

You should give a detailed description of the approach you took in the above section, using tables and figures to describe what you did. Separately, you need to perform an analysis of your approach, in terms of the both quantitiative (BLEU score) and qualitative (examples of translations) results, again, using tables and figures as necessary. In the above examples the students could receive full credit by doing the following:

- Student A: Gives a basic description of what an LSTM is. Then describes the experiments they carried out, and the reasoning behind them. For example, why did they change the number of hidden units? What happens when there are too few hidden units? What happens when there are too many? They then describe the results of their experiments, both verbally and through figures (e.g. a 2D heatmap of dev loss where the x and y axes are the number of hidden layers and hidden units respectively, and line plot of the dev loss/BLEU score as a function of the regularization strength). Lastly, they provide a table comparing the BLEU score of their best model to the baseline model on the test set.
- Student B: Gives a basic description of the attention mechanism they implemented. Then provide a plot of the attention weights for an example translation, and comment on the plot. They then describe their experiments/results similar to how Student A did above. Lastly, they provide a table comparing the BLEU score of their best model to the baseline model on the test set.
- Student C: Gives a thorough description of how character-level convolutions work. They then compare the word embeddings they obtained to the word embeddings learned by the baseline model (see Table 6 in the reference paper for a good example of how to do this). They then provide a qualitative analysis of the model outputs (e.g. find an example translation where their model performs better than the baseline model, find examples where it does not work as well, etc.). Lastly, they provide a table comparing the BLEU score of their best model to the baseline model on the test set.

3 Suggestions/Tips

Being the last homework, I really do not want you to be struggling with it. Please post on Piazza if you have any concerns, and come to my/Rob's office hours. That said, here are some suggestions to consider.

- **If you want your model to finish training before the deadline, you will almost certainly need to train it using a GPU**. As noted on Piazza, Google has provided free credits to use its cloud computing platform for this purpose (and you can request for more credits if needed). Instructions for getting started are provided in the tutorials section of the course GitHub repository[4]. We have provided an Ubuntu image with all of the necessary software/drivers pre-installed. Accordingly, lack of access to computational resources will not be considered a valid excuse for not finishing this homework.
- **Start early**. Especially, if you are unfamiliar with neural networks, UNIX-based operating systems, and/or the PyTorch API. As noted above, neural networks take a considerable amount of time to train. Furthermore, it can be difficult to debug errors when making changes to the provided architecture. This, in order to ensure that you meet the deadline we recommend you get started on this assignment right away.
- **Consult the PyTorch documentation and tutorials**. The documentation should be your first point-of-reference if you run into issues using any of the neural modules, and looking at the tutorials may help clarify how to implement certain architectures (**cough** **cough** attention **cough** **cough**).

[4] https://github.com/sameersingh/uci-statnlp/blob/master/tutorials/setting_up_google_cloud.md

4 Statement of Collaboration

It is **mandatory** to include a *Statement of Collaboration* in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using Piazza) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content *before* they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to Piazza, etc.). Especially *after* you have started working on the assignment, try to restrict the discussion to Piazza as much as possible, so that there is no doubt as to the extent of your collaboration.

Since we do not have a leaderboard for this assignment, you are free to discuss the numbers you are getting with others, and again, I encourage you to use Piazza to post your translations and compare them with others.

Acknowledgements

This homework was made possible by the course reader Robert Logan, who wrote both the source code and this assignment description. In addition, we would like to thank the PyTorch team whose machine translation tutorial was drawn heavily from to create this assignment.