CSEP 517: Homework #4
Machine Translation

Due Sun. March 15, 2020 at 11:59pm.

Submission instructions Submit 3 files on Canvas:

1 **Completed NMT Model:** (`nmt_model.py`): The completed `nmt_model.py` for problem 1.

2 **Code:** (`HW4.tar.gz`): The code needed to reproduce your results from problem 2. As usual, this should include a README file with instructions for how to run it. This code does not have to contain a very large change from the model you constructed in problem 1, but we want to give you a chance to modify the provided scripts however you want in order to improve performance. This does not need to include the datasets files we provide.

3 **Report:** (`HW4.pdf`): You will also submit a typed report in the pdf format. No restriction on font sizes, page margins, or number of pages. As before, consider creating tables and figures to organize the experimental results and try to make concrete arguments through examples when doing error analysis.

[Code] Part 1: Neural Machine Translation (40 points)

In this problem, we require you to implement a sequence-to-sequence (seq2seq) network with attention to build a neural machine translation (NMT) system (translating from French to English). For this problem we provide a partially completed implementation for you and ask you fill the missing pieces. This will give you a chance to work with a completed codebase for a complex NLP task. To make this task feasible without a GPU, we will be restricting ourselves to short sentences with limited vocabulary, although the codebase could potentially scale to the much larger datasets you would need to train a full-fledged MNT system. We encourage you to read through all the code to get an idea of how the whole project is put together.

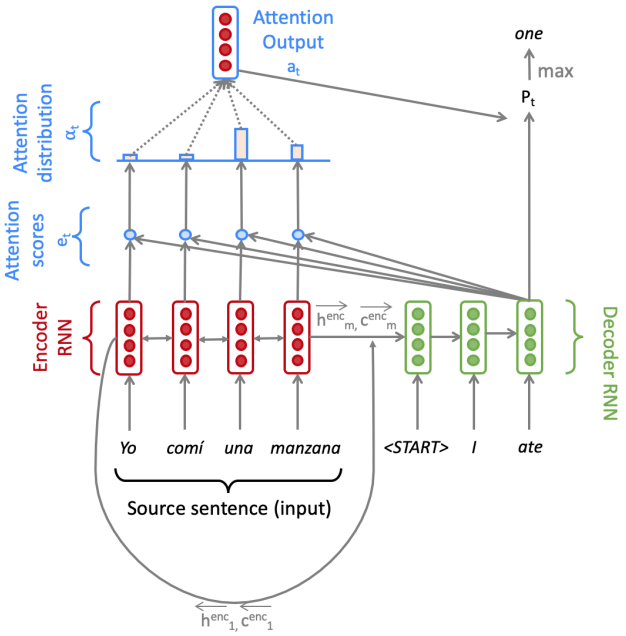
Setup To get started, make sure you have python 3.5+ installed, and then download the data and starter code from canvas (CSE517-HW4-Data.zip) and unzip the file. To install the libraries (including `PyTorch 1.3.1`), run:

```
pip install -r requirements.txt
```

Data We provide 6 data files in the `data` directory: `train.fr`, `train.en`, `dev.fr`, `dev.en`, `test.en` and `test.fr`. Our goal is to build a machine translation system which translates a French sentence into an English sentence. `train.fr` and `train.en` (same for `dev.fr` and `dev.en`) have the same number of lines, and each sentence in `train.fr` corresponds to a sentence in `train.en`. All the sentences are already tokenized and converted to BPE (byte pair encoding) tokens. For example,

```
i'm not sleeping well. ⟹ i m not sleep@@ ing well .
    we're committed. ⟹ we re comm@@ it@@ ted .
```

The idea of using BPE tokens is to address the sparsity of words and share the subword units between different words. If you are interested in learning more, see the paper Neural Machine Translation of Rare Words with Subword Units. In this problem, we will train a model on the training set and evaluate the model on `dev.fr`.



Model The model we will be implementing is a seq2seq (or called encoder-decoder) with attention model that we learned in the class. The model is formally specified as follows:

Given a sentence in the source language, we look up the word embeddings from an embeddings matrix, yielding $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^{e \times 1}$, where m is the length of the source sentence and e is the embedding size. We feed these embeddings to the bidirectional LSTM encoder, yielding hidden states and cell states for both the forward ($\rightarrow$) and backward ($\leftarrow$) LSTMs. The forward and backward versions are concatenated to give hidden states $\mathbf{h}_i^{\text{enc}}$ and cell states $\mathbf{c}_i^{\text{enc}}$:

$$\mathbf{h}_i^{\text{enc}} = [\overleftarrow{\mathbf{h}_i^{\text{enc}}}; \overrightarrow{\mathbf{h}_i^{\text{enc}}}] \text{ where } \mathbf{h}_i^{\text{enc}} \in \mathbb{R}^{2h \times 1}, \overleftarrow{\mathbf{h}_i^{\text{enc}}}, \overrightarrow{\mathbf{h}_i^{\text{enc}}} \in \mathbb{R}^{h \times 1} \qquad 1 \leq i \leq m$$

$$\mathbf{c}_i^{\text{enc}} = [\overleftarrow{\mathbf{c}_i^{\text{enc}}}; \overrightarrow{\mathbf{c}_i^{\text{enc}}}] \text{ where } \mathbf{c}_i^{\text{enc}} \in \mathbb{R}^{2h \times 1}, \overleftarrow{\mathbf{c}_i^{\text{enc}}}, \overrightarrow{\mathbf{c}_i^{\text{enc}}} \in \mathbb{R}^{h \times 1} \qquad 1 \leq i \leq m$$

We then initialize the Decoder's first hidden state $\mathbf{h}_0^{\text{dec}}$ and cell state $\mathbf{c}_0^{\text{dec}}$ with a linear projection of the Encoder's final hidden state and final cell state.

$$\mathbf{h}_0^{\text{dec}} = \mathbf{W}_h[\overleftarrow{\mathbf{h}_1^{\text{enc}}}; \overrightarrow{\mathbf{h}_m^{\text{enc}}}] \text{ where } \mathbf{h}_0^{\text{dec}} \in \mathbb{R}^{h \times 1}, \mathbf{W}_h \in \mathbb{R}^{h \times 2h}$$

$$\mathbf{c}_0^{\text{dec}} = \mathbf{W}_c[\overleftarrow{\mathbf{c}_1^{\text{enc}}}; \overrightarrow{\mathbf{c}_m^{\text{enc}}}] \text{ where } \mathbf{c}_0^{\text{dec}} \in \mathbb{R}^{h \times 1}, \mathbf{W}_c \in \mathbb{R}^{h \times 2h}$$

With the Decoder initialized, we must now feed it a matching sentence in the target language. On the t^{th} step, we look up the embedding for the t^{th} word, $\mathbf{y}_t \in \mathbb{R}^{e \times 1}$. We then concatenate $\mathbf{y}_t$ with the *combined-output vector* $\mathbf{o}_{t-1} \in \mathbb{R}^{h \times 1}$ from the previous timestep (we will explain what this is later down this page!) to produce $\overline{\mathbf{y}_t} \in \mathbb{R}^{(e+h) \times 1}$. Note that for the first target word (i.e. the start token) $\mathbf{o}_0$ is a zero-vector. We then feed $\overline{\mathbf{y}_t}$ as input to the Decoder LSTM.

$$\mathbf{h}_t^{\text{dec}}, \mathbf{c}_t^{\text{dec}} = \text{Decoder}(\overline{\mathbf{y}_t}, \mathbf{h}_{t-1}^{\text{dec}}, \mathbf{c}_{t-1}^{\text{dec}}) \text{ where } \mathbf{h}_t^{\text{dec}} \in \mathbb{R}^{h \times 1}, \mathbf{c}_t^{\text{dec}} \in \mathbb{R}^{h \times 1}$$

We then use $\mathbf{h}_t^{\text{dec}}$ to compute multiplicative attention over $\mathbf{h}_1^{\text{enc}}, \ldots, \mathbf{h}_m^{\text{enc}}$:

$$\mathbf{e}_{t,i} = (\mathbf{h}_t^{\text{dec}})^T \mathbf{W}_{\text{attProj}} \mathbf{h}_i^{\text{enc}} \text{ where } \mathbf{e}_t \in \mathbb{R}^{m \times 1}, \mathbf{W}_{\text{attProj}} \in \mathbb{R}^{h \times 2h} \qquad 1 \leq i \leq m$$

$$\alpha_t = \text{Softmax}(\mathbf{e}_t) \text{ where } \alpha_t \in \mathbb{R}^{m \times 1}$$

$$\mathbf{a}_t = \sum_{i=1}^{m} \alpha_{t,i} \mathbf{h}_i^{\text{enc}} \text{ where } \mathbf{a}_t \in \mathbb{R}^{2h \times 1}$$

We now concatenate the attention output $\mathbf{a}_t$ with the decoder hidden state $\mathbf{h}_t^{\text{dec}}$ and pass this through a linear layer, tanh, and Dropout to attain the *combined-output* vector $\mathbf{o}_t$.

$$\mathbf{u}_t = [\mathbf{a}_t; \mathbf{h}_t^{\text{dec}}] \text{ where } \mathbf{u}_t \in \mathbb{R}^{3h \times 1}$$

$$\mathbf{v}_t = \mathbf{W}_u \mathbf{u}_t \text{ where } \mathbf{v}_t \in \mathbb{R}^{h \times 1}, \mathbf{W}_u \in \mathbb{R}^{h \times 3h}$$

$$\mathbf{o}_t = \text{Dropout}(\tanh(\mathbf{v}_t)) \text{ where } \mathbf{o}_t \in \mathbb{R}^{h \times 1}$$

Then, we produce a probability distribution $\mathbf{P}_t$ over target words at the t^{th} timestep:

$$\mathbf{P}_t = \text{Softmax}(\mathbf{W}_{\text{vocab}}\mathbf{o}_t) \text{ where } \mathbf{P}_t \in \mathbb{R}^{V_t \times 1}, \mathbf{W}_{\text{vocab}} \in \mathbb{R}^{V_t \times h}$$

Here, V_t is the size of the target vocabulary. Finally, we train the model using the sum of cross-entroy loss for each target word at timestep t using $\mathbf{P}_t$ (and the gold target word).

Tips

- Before you get started, make sure you understand all the equations in the model specification, and then take a look at the `__init__` function in `NMT` in `nmt_model.py` that we have already implemented and understand what each of these variables is used for.
- Although the problem might look intimidating at first, we provide detailed documentation for each part you need to implement. The total number of lines that you'd need to implement to get a working model won't exceed 30 lines. Read them carefully and look up PyTorch's documentation when needed https://pytorch.org/docs/stable/.
- Make sure you have passed all the sanity checks before you start running the full model.

1.1 (8 points) Complete the `encode` function in `nmt_model.py` which applies the encoder to source sentences to obtain encoder hidden states. To check if your implementation is correct, run

```
python sanity_check.py encode
```

If you see `All Sanity Checks Passed!`, you are ready to move to the next step.

1.2 (8 points) Complete the `decode` function in `nmt_model.py` which computes combined output vectors for a batch. To check if your implementation is correct, run

```
python sanity_check.py decode
```

If you see `All Sanity Checks Passed!`, you are ready to move to the next step.

1.3 (8 points) Complete the `step` function in `nmt_model.py`: one forward step of the LSTM decoder, including the attention computation. To check if your implementation is correct, run

```
python sanity_check.py step
```

If you see `All Sanity Checks Passed!`, you are ready to move to the next step.

1.4 (16 points) Now you should be ready to train an NMT system on the real data. First run:

```
python vocab.py
```

It builds a vocabulary file `vocab.json` which contains vocabulary for both source and target languages. *Report how many word types are used in French and English respectively.*

Next, start training the model by running

```
python main.py --mode train --model_path model.bin
```

This script will load the training and development sets and the pre-computed vocabulary file and start the training process using the model that you just completed. If you have access to a GPU, you can use it by setting the `--cuda` flag. Take a look at the hyperameters defined in `main.py` (don't change them though!), and observe your training log and *report the following*:

- How many training examples are in total? What batch size is used and therefore how many batches per epoch?
- How often does the model report the (accumulated) perplexity on the training set? How often does the model report the perplexity on the development set? Do you observe that the perplexity on both training and development sets are always decreasing?
- How many epochs does it run in total?

- Include a plot of the the perplexity on training and development sets in one figure and explain what you observed.

On our CPU machine, it took around 10 seconds per an epoch.
Next, run (check that you have a `model.bin` file in your directory):

```
python main.py --mode eval_dev --model_path model.bin --output_file dev.out
```

report the BLEU score you've obtained on the development set. Open the `dev.out` file that your model generated and compare it with `./data/dev.en`. What do you think of the quality of the translations? Are these grammatical English sentences? Can you identify any common mistakes?

Submission Submit a version of `mnt_model.py` that passes the sanity checks and can be trained using the unmodified `main.py` script to canvas.

[Code] Part 2: Neural Machine Translation Experiments (40 points)

Now you have a baseline implementation, this problem asks you to experiment with the design of the MNT model.

2.1 (10 points) Currently the implementation fixes the hidden size of the decoder and the encoder to be the same. Modify the code so that these values can be set independently. Note you should still project the output of the decoder so that $v_t \in \mathbb{R}^h$ where h is the encoder hidden size, however now you should have $h_t^{dec} \in \mathbb{R}^{h'}$ and $c_t^{dec} \in \mathbb{R}^{h'}$ where h' is the decoder hidden size. Trying running the model with the hidden size of the decoder set to 128 while the hidden size of the encoder is set to 64. What BLEU score does this get on the dev set?

2.2 (15 points) We have chosen to use multiplicative attention above: $\mathbf{e}_i = \mathbf{s}^\intercal \mathbf{W} \mathbf{h}_i \in \mathbb{R}$. Implement two other common types of attention[1]:

- Addictive attention: $\mathbf{e}_i = \mathbf{v}^\intercal \tanh(\mathbf{W}_1 \mathbf{s} + \mathbf{W}_2 \mathbf{h}_i)$. This will require adding additional parameters to the `__init__` method.
- Dot-product attention: $\mathbf{e}_i = \mathbf{s}^\intercal \mathbf{h}_i$. This will require that $h' = 2h$ and that the `att_projection` attribute is set to `None`.

Repeat the experiments above and report the results from using these three different types of attention.

2.2 (15 points) Can you improve the performance of the model? Or, alternatively, can you reduce the runtime of the model without harming performance? Run some experiments on the dev set and report your findings (even if they are not successful). The script comes with some built in hyperparameters you can tune, but you should also come up an idea for a more substantial change in the network design. You could also consider changing the optimization procedure in `main.py`. Feel free to modify the code however your want, or add additional code to help you run the experiments.

Finally, run your best model on the test set and report the results. For the default script, this means running:

```
python main.py --mode eval_test --model_path your_best_model.bin --output_file test.out
```

Submission Submit your code to canvas, you may exclude the data files we provided but include everything else needed to run the code and a README.

[Written] Part 3: Word2Vec (20 points)

It's only words, and words are all I have. To take your heart away.– Bee Gees

Word2vec is one of the most widely used methods for creating dense word representations. Here we will review the Skip Gram with Negative Sampling (SGNS) formulation of Word2vec. In this formulation, assume we have a vocabulary of words, V, and additionally have a dataset, D, of word pairs (w, c) where $w \in V$ and

[1] More details can be found at Effective Approaches to Attention-based Neural Machine Translation, Section 3.1.

$c \in V$. Typically D is constructed by scanning a large corpus of text and then pairing each word with context words that occur within a small window, win, around that word. Each word, w is associated with a learned word-vector, u_w, and a learned context-vector v_w. To train these vectors, we maximize the following objective:

$$\sum_{(w,c)\in D} (\log e^{u_w \cdot v_c} - \log \sum_{c\prime \in N} e^{u_w \cdot v_{c'}}) \tag{0.1}$$

A key component of this loss is the set of *negative samples*, N. This set is constructed by randomly sampling words from a smoothed unigram distribution:

$$p_\alpha(w) = \frac{\#(w)^\alpha}{\sum_{w'} \#(w')^\alpha} \tag{0.2}$$

where $\#(w)$ refers to the frequency of w in corpus and α is a smoothing hyperparameter.

3.1 (5 points) Another way to formulate the training of word vectors is to predict the context around each word in the corpus. Given a set of word-context pairs as input, we seek of maximize $\log p(c|w)$. We could achieve this by directly maximizing the following objective.

$$\sum_{(w,c)\in D} \log \frac{e^{v_c \cdot u_w}}{\sum_{c\prime \in V} e^{v_{c\prime} \cdot u_w}} \tag{0.3}$$

where V denotes the vocabulary. Compare this approach to SGNS:
a. Describe the disadvantage(s) to this approach.
b. Does SGNS depend on how the negatives are sampled? If so, what would happen if we sample negatives uniformly at random from the vocabulary?

3.2 (5 points) Imagine that we train the model on a large corpus (e.g. English Wikipedia). Describe the effects of context window size win to the resulting word vectors u_w, i.e. what if we use $win = 1, 5, or 100$?

3.3 (5 points) The smoothing hyperparameter, α in Equation 0.3 is typically set to 0.75. Compared to a the unsmoothed variant ($\alpha = 1.0$), will the cosine similarities of most word vectors with context vectors of rare words generally increase or decrease when $\alpha = 0.75$? Justify your answer.

3.4 (5 points) One of the most satisfying results from the word2vec paper was the model's ability to do well in analogies. For example, given "man is to woman, as king is to ?" (sometimes written as *man:woman::king:?*), word2vec successfully predicts *queen*. More generally, given *a:b::x:?*, we predict the missing word from the vocabulary V as follows:

$$\underset{y\in V}{\operatorname{argmax}}\, cos(y, b - a + x) \tag{0.4}$$

where we overload a, b, x, and y to indicate normalized (unit) word vectors. However, we also observed some interesting failure cases. For example, given *london:england::baghdad:?*, the model predicted *mosul* instead of *iraq*. Can you hypothesize why this could have happened? *Hint: See if you can decompose the cos term in Equation 0.4 into a series of cos computations.*