Computer Science 401
St. George Campus

15 February 2021
University of Toronto

Homework Assignment #2
Due: Wednesday, 10 March 2021 at 19h00 (7 PM),

Neural Machine Translation

TAs: Zhewei Sun (zheweisun@cs.toronto.edu); Frank Niu (niu@cs.toronto.edu)

1 Overview

1.1 Canadian Hansards

The main corpus for this assignment comes from the official records (*Hansards*) of the 36^{th} Canadian Parliament, including debates from both the House of Representatives and the Senate. This corpus is available at `/u/cs401/A2/data/Hansard/` and has been split into `Training/` and `Testing/` directories.

This data set consists of pairs of corresponding files (`*.e` is the English equivalent of the French `*.f`) in which every line is a sentence. Here, sentence alignment has already been performed for you. That is, the n^{th} sentence in one file corresponds to the n^{th} sentence in its corresponding file (e.g., line n in `fubar.e` is aligned with line n in `fubar.f`). Note that this data only consists of sentence pairs; many-to-one, many-to-many, and one-to-many alignments are not included.

1.2 Seq2seq

We will be implementing a simple *seq2seq* model, without attention, with single-headed attention, and with multi-headed attention based largely on the course material. You will train the models with teacher-forcing and decode using beam search. We will write it in *PyTorch* version 1.2.0 (`https://pytorch.org/docs/1.2.0/`), which is the version installed on the *teach.cs* servers. For those unfamiliar with PyTorch, we suggest you first read the PyTorch tutorial (`https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`).

1.3 Tensors and batches

PyTorch, like many deep learning frameworks, operate with tensors, which are multi-dimensional arrays. When you work in PyTorch, you will rarely if ever work with just one bitext pair at a time. You'll instead be working with multiple sequences in one tensor, organized along one dimension of the batch. This means that a pair of source and target tensors `F` and `E` actually correspond to multiple sequences $F = (F^{(m)}_{1:S^{(m)}})_{m\in[1,M]}$, $E = (E^{(m)}_{1:T^{(m)}})_{m\in[1,M]}$. We work with batches instead of individual sequences because: a) backpropagating the average gradient over a batch tends to converge faster than single samples, and b) sample computations can be performed in parallel. For example, if we want to multiply source sequences $F^{(m)}$ and $F^{(m+1)}$ with an embedding matrix W, we can tell one CPU core to compute the result for $F^{(m)}$ and another for $F^{(m+1)}$, halving the overall time it would take to multiply them independently. Learning to work with tensors can be difficult at first, but is integral to efficient computation. We suggest you read more about it in the *NumPy* docs (`https://docs.scipy.org/doc/numpy/user/theory.broadcasting.html#array-broadcasting-in-numpy`), which PyTorch borrows for tensors.

1.4 Differences from the lectures

There are three changes to the seq2seq architectures that we make for this assignment. First, instead of scaled dot-product energy scores for attention $score(a, b) = |a|^{-1/2}\langle a, b\rangle$, we'll use the cosine similarity between vectors a and b:

$$score(a, b) = \frac{\langle a, b\rangle}{\max\left(\|a\|_2 \|b\|_2, \varepsilon\right)}$$

Where $0 < \varepsilon \ll 1$ ensures $score(a, b) = 0$ when $a = 0$ or $b = 0$.

The second relates to how we calculate the first hidden state for the decoder when we don't use attention. Recall that a bidirectional recurrent architecture processes its input in both directions separately: the forward direction processes $(x_1, x_2, \ldots, x_S)$ whereas the backward direction processes $(x_S, x_{S-1}, \ldots, x_1)$. The bidirectional hidden state concatenates the forward and backward hidden states for the same time $h_t = [h_t^{forward}, h_t^{backward}]$. This implies h_S has processed all the input in the forward direction, but only one input in the backward direction (and vice versa for h_1). To ensure the decoder gets access to all input from both directions, you should initialize the first decoder state

$$\tilde{h}_1 = [h_S^{forward}, h_1^{backward}]$$

When you use attention, set $\tilde{h}_1 = 0$.

Our final "change" isn't so much a change as a clarification. In multi-headed attention, recall we have N heads such that $\tilde{h}_t^{(n)} = \tilde{W}^{(n)}\tilde{h}_t$ and $h_s^{(n)} = W^{(n)}h_s$. $W^{(n)}$ and $\tilde{W}^{(n)}$ need not be square matrices; the size of $\tilde{h}_t^{(n)}$ need not be the size of $\tilde{h}_t$ nor $h_s^{(n)}$ the size of h_s. For this assignment, we will be setting the size of $|\tilde{h}_t^{(n)}| = |\tilde{h}_t|/N$ and $|h_s^{(n)}| = |h_s|/N$ (you may assume N evenly divides the hidden state size).

2 Your tasks

2.1 Setup

You are expected to run your solutions on *teach.cs*. **Download the starter code from MarkUs.** You can download the starter files by clicking the "Download" button in the "Starter Files" section of the Assignment page. The Hansards parallel text data are located in the `/h/u1/cs401/A2/data/` directory on `teach.cs`. You don't need to copy the text data to your own `teach.cs` directory.

Download the starter code MarkUs into your working directory to get started. You should have 8 files: `a2_abc.py`, `a2_bleu_score.py`, `a2_encoder_decoder.py`, `a2_dataloader.py`, `a2_run.py`, `a2_training_and_testing.py`, `test_a2_bleu_score.py`, and `test_a2_encoder_decoder.py`.

2.2 Calculating BLEU scores

Modify `a2_bleu_score.py` to be able to calculate BLEU scores on single reference and candidate strings. We will be using the definition of BLEU scores from the lecture slides:

$$BLEU = BP_C \times (p_1 p_2 \ldots p_n)^{(1/n)}$$

To do this, you will need to implement the functions `grouper(...)`, `n_gram_precision(...)`, `brevity_penalty(...)`, and `BLEU_score(...)`. Make sure to **carefully** follow the doc strings of each function. Do not re-implement functionality that is clearly performed by some other function.

Your functions will operate on sequences (e.g., lists) of tokens. These tokens could be the words themselves (strings) or an integer ID corresponding to the words. Your code should be agnostic to the type of token used, though you can assume that both the reference and candidate sequences will use tokens of the same type.

2.3 Building the encoder/decoder

You are expected to fill out a number of methods in `a2_encoder_decoder.py`. These methods belong to sub-classes of the abstract base classes in `a2_abcs.py`. The latter defines the abstract classes `EncoderBase`, `DecoderBase`, and `EncoderDecoderBase`, which implement much of the boilerplate code necessary to get a seq2seq model up and running. Though you are welcome to read and understand this code, it is not necessary to do so for this assignment. You will, however, need to read the doc strings in `a2_abcs.py` to understand what you're supposed to fill out in `a2_encoder_decoder.py`. **Do not modify any of the code in** `a2_abcs.py`.

A high-level description of the contents of the requirements for `a2_encoder_decoder.py` follows here. More details can be found in the doc strings in `a2_{abcs,encoder_decoder}.py`.

2.3.1 Encoder

`a2_encoder_decoder.Encoder` will be the concrete implementation of all encoders you will use. The encoder is always a multi-layer neural network with a bidirectional recurrent architecture. The encoder gets a batch of source sequences as input and outputs the corresponding sequence of hidden states from the last recurrent layer.

`Encoder.forward_pass` defines the structure of the encoder. For every model in PyTorch, the `forward` function defines how the model will run, and the `forward` function of every encoder or decoder will first clean up your input data and call `forward_pass` to actually define the model structure. Now you need to implement the `forward_pass` function that defines how your encoder will run.

`Encoder.init_submodules(...)` should be filled out to initialize a word embedding layer and a recurrent network architecture. `Encoder.get_all_rnn_inputs(...)` accepts a batch of source sequences $F^{(m)}_{1:S^{(m)}}$ and lengths $S^{(m)}$ and outputs word embeddings for the sequences $x^{(m)}_{1:S^{(m)}}$.

`Encoder.get_all_hidden_states(...)` converts the word embeddings $x^{(m)}_{1:S^{(m)}}$ into hidden states for the last layer of the RNN $h^{(m)}_{1:S^{(m)}}$ (note we're using (m) here for the batch index, not the layer index).

2.3.2 Decoder without attention

`a2_encoder_decoder.DecoderWithoutAttention` will be the concrete implementation of the decoders that do not use attention (so-called "transducer" models). Method implementations should thus be tailored to not use attention.

In order to feed the previous output into the decoder as input, the decoder can only process one step of input at a time and produce one output. Thus `DecoderWithoutAttention` is designed to process one slice of input at a time (though it will still be a batch of input for that given slice of time). The goal, then, is to take some target slice from the previous time step $E^{(m)}_{t-1}$ and produce an un-normalized log-probability distribution over target words at time step t, called $logits^{(m)}_t$. Logits can be converted to a categorical distribution using a softmax:

$$P(y^{(m)}_t = i | \ldots) = \frac{\exp(logits^{(m)}_{t,i})}{\sum_j \exp(logits^{(m)}_{t,j})}$$

`DecoderWithoutAttention.forward_pass` defines the structure of network. Similar to what you did for your encoder, you need to assemble the model here.

`DecoderWithoutAttention.init_submodules(...)` should be filled out to initialize a word embedding layer, a recurrent *cell*, and a feed-forward layer to convert the hidden state to logits.

`DecoderWithoutAttention.get_first_hidden_state(...)` produces $\tilde{h}^{(m)}_1$ given the encoder hidden states $h^{(m)}_{1:S^{(m)}}$.

`DecoderWithoutAttention.get_current_rnn_input(...)` takes the previous target $E_{t-1}^{(m)}$ (or the previous output $y_{t-1}^{(m)}$ in testing) and outputs word embedding $\tilde{x}_t^{(m)}$ for the current step.

`DecoderWithoutAttention.get_current_hidden_state(...)` takes $\tilde{x}_t^{(m)}$ and $\tilde{h}_{t-1}^{(m)}$ and produces the current decoder hidden state $\tilde{h}_{t-1}^{(m)}$.

`DecoderWithoutAttention.get_current_logits(...)` takes $\tilde{h}_t^{(m)}$ and produces $logits_t^{(m)}$.

2.3.3 Decoder with (single-headed) attention

`a2_encoder_decoder.DecoderWithAttention` will be the concrete implementation of the decoders that use single-headed attention. It inherits from `DecoderWithoutAttention` to avoid re-implementing `get_current_hidden_states(...)` and `get_current_logits(...)`. The remaining methods must be re-implemented, but slightly modified for the attention context.

Two new methods must be implemented for `DecoderWithAttention`.

`DecoderWithAttention.get_energy_scores(...)` takes in a decoder state $\tilde{h}_t^{(m)}$ and all encoder hidden states $h_{1:S^{(m)}}^{(m)}$ and produces energy scores for *that* decoder state but *all* encoder hidden states: $e_{t,1:S^{(m)}}^{(m)}$.

`DecoderWithAttention.attend(...)` takes in a decoder state $\tilde{h}_t^{(m)}$ and all encoder hidden states $h_{1:S^{(m)}}^{(m)}$ and produces the attention context vector $c_t^{(m)}$. Between `get_energy_scores` and `attend`, use `get_attention_weights(...)` to convert $e_{t,1:S^{(m)}}^{(m)}$ to $\alpha_{t,1:S^{(m)}}^{(m)}$, which has been implemented for you.

2.3.4 Decoder with multi-head attention

`a2_encoder_decoder.DecoderWithMultiHeadAttention` implements a multi-headed variant of attention. It inherits from `DecoderWithAttention`.

Two methods must be re-implemented for this variant.

`DecoderWithMultiHeadAttention.init_submodules(...)` should initialize new submodules for the matrices W, $\tilde{W}$, and Q.

`DecoderWithMultiHeadAttention.attend(...)` should "split" hidden states $\tilde{h}_t^{(m)}$ into $\tilde{h}_t^{(m,n)}$ and $h_s^{(m)}$ into $h_s^{(m,n)}$, where m still indexes the batch number and n indexes the head. Then it should call `super().attend(...)` to do the attention, and combine $c_t^{(m,n)}$ of the N heads. We want you to do this *without ever actually "splitting" any tensors*! The key is to *reshape* the a full hidden output into N chunks. If it's a little bit too tricky, try starting from writing the case where $N = 1$, i.e. when there's no need for splitting.

2.3.5 Putting it together: the Encoder/Decoder

`a2_encoder_decoder.EncoderDecoder` coordinates the encoder and decoder. Its behaviour depends on whether it's being used for training or testing. In training, it receives both $F_{1:S^{(m)}}^{(m)}$ and $E_{1:T^{(m)}}^{(m)}$ and outputs $logits_{1:T^{(m)}}^{(m)}$ un-normalized log-probabilities over $y_{1:T^{(m)}}^{(m)}$. In testing, it receives only $F_{1:S^{(m)}}^{(m)}$ and outputs K paths from beam search per batch element n: $y_{1:T^{(n,k)}}^{(n,k)}$.

`EncoderDecoder.init_submodules(...)` initializes the encoder and decoder.

`EncoderDecoder.get_logits_for_teacher_forcing(...)` provides you the encoder output $h_{1:S^{(m)}}^{(m)}$ and the targets $E_{1:T^{(m)}}^{(m)}$ and asks you to derive $logits_{1:T^{(m)}}^{(m)}$ according to the MLE (teacher-forcing) objective.

`EncoderDecoder.update_beam(...)` asks you to handle one iteration of a simplified version of the beam search from the slides. While a proper beam search requires you to handle the set of finished paths

f, `update_beam` doesn't need to. Letting (n, k) indicate the n^{th} batch elements' k^{th} path:

$$\begin{aligned}\forall n, k, v. b_{t,0}^{(n,k\to v)} &\leftarrow \tilde{h}_{t+1}^{(n,k)} \\ b_{t,1}^{(n,k\to v)} &\leftarrow [b_{t,1}^{(n,k)}, v] \\ \log P(b_t^{(n,k\to v)}) &\leftarrow \log P(b_t^{(n,k)}) + \log P(y_{t+1} = v|\tilde{h}_{t+1}^{(n,k)})\end{aligned}$$

$$\forall n, k. b_{t+1}^{(n,k)} \leftarrow \text{argmax}^k_{b_t^{(n,k'\to v)}} \log P(b_t^{(n,k'\to v)})$$

In short, extend the existing paths, then prune back to the beam width. A greedy update function `update_greedy` is provided for you in `a2_abcs.py`. You can use the option `--greedy` to switch to greedy update. This option might be handy when you want to test the correctness of the rest of your assignment.

2.3.6 Padding

An important detail when dealing with sequences of input and output is how to deal with sequence lengths. Individual sequences within a batch $F^{(m)}$ and $E^{(m)}$ can have unequal lengths $S^{(m)} \neq S^{(n+1)}$, $T^{(m)} \neq T^{(n+1)}$, but we pad the shorter sequences to the right to match the longest sequence. This allows us to parallelize across multiple sequences, but it's important that whatever the network learns (i.e., the error signal) is not impacted by padding. We've mostly handled this for you in the functions we've implemented, with three exceptions: first, no word embedding should be learned for padding (which you'll have to guarantee); second, you'll have to ensure the bidirectional encoder doesn't process the padding; and third, the first hidden state of the decoder (without attention) should not be based on padded hidden states. You are given plenty of warning in the starter code when these three cases ought to be considered. The decoder uses the end-of-sequence symbol as padding, which is entirely handled in `a2_training_and_testing.py`.

2.4 The training and testing loops

After following the PyTorch tutorial, you should be familiar with how models are trained and tested in PyTorch. You are expected to implement training and testing loops in `a2_training_and_testing.py`.

In `a2_training_and_testing.compute_batch_total_bleu(...)`, you are given reference (from the dataset) and candidate batches (from the model) in the target language and asked to compute the total BLEU score over the batch. You will have to convert the PyTorch tensors in order to use `a2_bleu_score.BLEU_score(...)`.

In `a2_training_and_testing.compute_average_bleu_over_dataset(...)`, you are to follow instructions in the doc string and use `compute_batch_total_bleu(...)` to determine the average BLEU score over a data set.

In `a2_training_and_testing.train_for_epoch(...)`, once again follow the doc strings to iterate through a training data set and update model parameters using gradient descent.

2.5 Running the models

Once you have completed the coding portion of the assignment, it is time you run your models. In order to do so in a reasonable amount of time, you'll have to train your models using a machine with a GPU. A number of teaching labs in the Bahen building have GPUs (listed in `https://www.teach.cs.toronto.edu/faq.html#ABOUT4`), but you must log in at the physical machines to use them (as opposed to remote access). **Even on a GPU, the code can take upwards of 2 hours to complete in full.** Be sure to plan accordingly!

If you have access to your own GPU, you may run this code locally and report the results. However, **any modifications you make to run the code locally must be reverted to work on teach before you submit!**

You are going to interface with your models using the script `a2_run.py`. This script glues together the components you implemented previously. The only meaningful remaining code is in `a2_dataloader.py`, which converts the Hansard sentences into sequences of IDs. Suffice to say that you do not need to know how either `a2_run.py` nor `a2_dataloader.py` works, only use them (unless you are interested).

Run the following code block line-by-line from your working directory. In order, it:

1. Builds maps between words and unique numerical identifiers for each language.
2. Splits the training data into a portion to train on and a hold-out portion.
3. Trains the encoder/decoder without attention and stores the model parameters.
4. Trains the encoder/decoder with single-headed attention and stores the model parameters.
5. Trains the encoder/decoder with multi-headed-attention and stores the model parameters.
6. Returns the average BLEU score of the encoder/decoder without attention on the test set.
7. Returns the average BLEU score of the encoder/decoder with single-headed attention on the test set.
8. Returns the average BLEU score of the encoder/decoder with multi-headed attention on the test set.

```
export TRAIN=/h/u1/cs401/A2/data/Hansard/Training/
export TEST=/h/u1/cs401/A2/data/Hansard/Testing/
# 1. Generate vocabularies
python3.7 a2_run.py vocab $TRAIN e vocab.e.gz
python3.7 a2_run.py vocab $TRAIN f vocab.f.gz
# 2. Split train and dev sets
python3.7 a2_run.py split $TRAIN train.txt.gz dev.txt.gz
# 3. Train a model without attention
srun -p csc401 \
    python3.7 a2_run.py train $TRAIN \
    vocab.e.gz vocab.f.gz \
    train.txt.gz dev.txt.gz \
    model_wo_att.pt.gz \
    --device cuda
# 4. Train a model with attention
srun -p csc401 \
    python3.7 a2_run.py train $TRAIN \
    vocab.e.gz vocab.f.gz \
    train.txt.gz dev.txt.gz \
    model_w_att.pt.gz \
    --with-attention \
    --device cuda
# 5. Train a model with multi-head attention
srun -p csc401 \
    python3.7 a2_run.py train $TRAIN \
    vocab.e.gz vocab.f.gz \
    train.txt.gz dev.txt.gz \
```

```
    model_w_mhatt.pt.gz \
    --with-multihead-attention \
    --device cuda
# 6. Test the model without attention
srun -p csc401 \
    python3.7 a2_run.py test $TEST \
    vocab.e.gz vocab.f.gz model_wo_att.pt.gz \
    --device cuda
# 7. Test the model with attention
srun -p csc401 \
    python3.7 a2_run.py test $TEST \
    vocab.e.gz vocab.f.gz model_w_att.pt.gz \
    --with-attention --device cuda
# 8. Test the model with multi-head attention
srun -p csc401 \
    python3.7 a2_run.py test $TEST \
    vocab.e.gz vocab.f.gz model_w_mhatt.pt.gz \
    --with-multihead-attention --device cuda
```

Steps 1 and 2 should not fail and need only be run once. Step 3 onward depend on the correctness of your code.

The `srun -p csc401` is necessary to run on a GPU on `teach`. You do not need a GPU for the first two steps. If you are enrolled in CSC 2511, please use the switch `-p csc2511` instead. If you are running the training/testing locally, you will not need `srun -p csc401` when running steps 3-6. `srun` uses SLURM (`https://en.wikipedia.org/wiki/Slurm_Workload_Manager`) to schedule processes on the department's cluster. Because all students in this class or any class requiring GPUs will be running their jobs on the cluster **please only run steps 3-6 after you have debugged your code**. We discuss how you can train a smaller network below that will only take a fraction of the time anyway.

In a file called `analysis.txt`, provide the following:

- The printout after every epoch of the training loop of both the model for the model trained without, with single-headed, and with multi-headed attention. Clearly indicate which is which.
- The average BLEU score reported on the test set for each model. Again, clearly indicate which is which.
- A brief discussion on your findings. Was there a discrepancy in between training and testing results? Why do you think that is? If one model did better than the others, why do you think that is?

2.6 Bonus [up to 15 marks]

We will give bonus marks for innovative work going substantially beyond the minimal requirements. However, your overall mark for this assignment cannot exceed 100%. Submit your write-up in `bonus.txt`.

You may decide to pursue any number of tasks of your own design related to this assignment, although you should consult with the instructor or the TA before embarking on such exploration. Certainly, the rest of the assignment takes higher priority. Some ideas:

- Perform substantial data analysis of the error trends observed in each method you implement. This must go well beyond the basic discussion already included in the assignment.

- There are many possible ways to assemble attentions for the encoder / decoder. For example, dot-product attention similar to (Vaswani et al., 2017)[1], and structured self attention (Lin et al., 2017)[2]. Analyze several attention mechanisms, compare their performances, and include discussions towards their reasons of the performance differences.

[1] https://arxiv.org/abs/1706.03762
[2] https://arxiv.org/abs/1703.03130

3 Submission requirements

This assignment is submitted electronically. Submit your assignment on MarkUs. Do **not** `tar` or `compress` your files, and do not place your files in subdirectories. Do not format your discussion as a PDF or Word document — use plain text only. You should submit:

1. The files `a2_bleu_score.py`, `a2_encoder_decoder.py`, and `a2_training_and_testing.py` that you filled out according to assignment specifications. We will not accept `a2_abc.py`, `a2_dataloader.py`, nor `a2_run.py` - your assignment must be compatible with the versions we provided you.
2. Your write-up on the experiment in `analysis.txt`.
3. If you are submitting a bonus, tell us what you've done by submitting a write-up in `bonus.txt`. Please distribute bonus code amongst the above `*.py` files, being careful not to break functions, methods, and classes related to the assignment requirements.

You should not submit any additional files that you generated to train and test your models. For example, do not submit your model parameter files `*.pt.gz` or vocab files `*.{e,f}.gz`. Only submit the above files. **Additional source files containing helper functions are not permitted.**

A Suggestions

A.1 Check Piazza regularly

Updates to this assignment as well as additional assistance outside tutorials will be primarily distributed via Piazza (`http://piazza.com/utoronto.ca/winter2021/csc4012511hslec9101/home`). **It is your responsibility to check Piazza regularly for updates.**

A.2 Run cluster code early and at irregular times

Because GPU resources are shared with your peers, your `srun` job may end up on a weaker machine (or even postponed until the resources are available) if too many students are training at once. To help balance resource usage over time, we recommend you finish this assignment as early as possible. You might find that your peers are more likely to run code at certain times in the day. To check how many jobs are currently queued or running on our partition, please run `squeue -p csc401`.

If you decide to run your models right before the assignment deadline, please be aware that we will be unable to request more resources or run your code sooner. **We will not grant extensions for this reason.**

By the way, if you end up scheduling your job on one of the slower machines, **you might end up with slightly different results** than on the fastest ones. This is an unfortunate but normal by-product of how the GPUs calculate results in slightly different ways. Your BLEU score should not differ by more than 3% absolute.

A.3 Keep training after disconnecting

Training a model can take hours. If your internet connection is weak, you run the risk of losing your progress. You can use the Linux `screen` (`https://linux.die.net/man/1/screen`) command to create a persistent shell that is only destroyed when you exit from it. This will allow training to continue even if disconnected.

The most basic usage of screen is as follows. To start a new shell, run the command `screen`. You should see the same shell interface as before (i.e. `wolf:<something>$`) but you are now in a new shell instance. You can type `Ctrl+D` or the command `exit` to kill the screen, or type `Ctrl+A` followed by `Ctrl+D` to *detach* from the screen. A detached screen persists between SSH sessions. If you want to reconnect to your shell, try the command `screen -r`.

A.4 Using your own computer

If you want to do some or all of this assignment on your laptop or other computer, you will have to do the extra work of downloading and installing the requisite software and data. You take on the risk that your computer might not be adequate for the task. You are strongly advised to upload regular backups of your work to teach.cs, so that if your machine fails or proves to be inadequate, you can immediately continue working on the assignment at teach.cs. When you have completed the assignment, you should try your programs out on teach.cs to make sure that they run correctly there. **A submission that does not work on teach.cs will get zero marks.**

That said, due to concerns of limited resources, we will allow you to report the results of training/testing your model on your local machine. The code must still conform to the teach.cs environment, but the contents of `analysis.txt` can be based on your local environment.

A.5 Unit testing

We strongly recommend you test the methods and functions you've implemented prior to running the training loop. You can test actual output against expected input for complex methods like `update_beam(...)`. The Python 3.7 teach environment has installed pytest (`https://docs.pytest.org/en/5.1.2/`). We have included some preliminary tests for `BLEU_score(...)` and `update_beam(...)`. You can run your test suite on teach by calling

```
python3.7 -m pytest
```

While the test suite will execute initially, the provided tests will fail until you implement the necessary methods and functions. While passing these initial tests is a necessity for full marks, they are not sufficient on their own. Please be sure to add your own tests.

Unit testing is not a requirement, nor will you receive bonus marks for it.

A.6 Debugging task

Instead of re-running the entire task on GPU when debugging your code, we recommend that you run a much smaller version of the task until you are confident that you are error free. Ideally, **you should only need to run the full task once.** The following commands may be used to set up such a task.

```
# Variables TRAIN and TEST as before.
export OMP_NUM_THREADS=4  # avoids a libgomp error on teach
# create an input and output vocabulary of only 100 words
python3.7 a2_run.py vocab $TRAIN e vocab_tiny.e.gz --max-vocab 100
python3.7 a2_run.py vocab $TRAIN f vocab_tiny.f.gz --max-vocab 100
# only use the proceedings of 4 meetings, 3 for training and 1 for dev
python3.7 a2_run.py split $TRAIN train_tiny.txt.gz dev_tiny.txt.gz --limit 4
# use far fewer parameters in your model
python3.7 a2_run.py train $TRAIN \
    vocab_tiny.e.gz vocab_tiny.f.gz \
    train_tiny.txt.gz dev_tiny.txt.gz \
    model.pt.gz \
    --epochs 2 \
    --word-embedding-size 51 \
    --encoder-hidden-size 101 \
    --batch-size 5 \
    --cell-type gru \
    --beam-width 2
# use with the flags --with-attention and --with-multihead-attention to test single-
# and multi-headed attention, respectively.
```

We request that you first see if running directly on the teach.cs server (`wolf`) runs quickly enough before attempting to use the cluster. Tested at low occupancy, each epoch took about 30 seconds with well-optimized code directly on `wolf`. At high occupancy, each epoch took 4 minutes.

Note that your BLEU will likely be high in the reduced vocabulary condition - even using very few parameters - since your model will end up learning to output the out-of-vocabulary symbol. **Do not report your findings on the toy task in** `analysis.txt`.

A.7 Beam search not finished warning

You might come across a warning like this during training:

```
a2_abcs.py:882: UserWarning: Beam search not finished by t=100. Halted
```

This just means your model failed to output an end-of-sequence token after $t = 100$.

This may not mean you've made a mistake. On certain machines in the cluster and certain network configurations, even our solutions give this warning. However, **i**t should not occur over all epochs. Your BLEU score shouldn't change much whether or not you get this warning because it should only occur on a few sentences. If the warning keeps popping up or your BLEU scores are close to zero, you probably have an error.

A.8 Recurrent cell type

You'll notice that the code asks you to set up a different recurrent cell type depending on the setting of the attribute `self.cell_type`. This could be an LSTM, GRU, or RNN (the last refers to the simple linear weighting $h_t = \sigma(W[x, h_{t-1}] + b)$ that you saw in class). The three cell types act very similarly - GRUs and RNNs are largely interchangeable from a programming perspective - except the LSTM cell often requires you to carry around both a cell state and a hidden state. Pay careful attention to the documentation for when `h_t`, `htilde_t`, etc. might actually be a pair of the hidden state and cell state as opposed to just a hidden state. Sometimes no change is necessary to handle the LSTM cell. Other times you might have to repeat an operation on the elements individually. Take advantage of the following pattern:

```
if self.cell_type == 'lstm':
    # do something
else:
    # do something else
```

Be sure to rerun training with different cell types (i.e. use the flag `--cell-type`) to ensure your code can handle the difference.

B Variable names and slides

We try to match the variable names in `a2_abc.py` to those in the lecture slides on machine translation. The table below serves as reference for converting between the two.

In general, we denote a specific indexed value of a variable from the slides, such as E_t, with an underscored variable name, e.g. `E_t`. When an index is omitted, it means the variable contains all the indexed values at once, e.g. `F` corresponds to $F_{1:S}$.

Note that the slides are 1-indexed, whereas code is 0-indexed. Also, all variables in the PyTorch code are batched, but the slides only look at one sequence at a time.

Variable	Slides	Notes
`F`	$F_{1:S}$	Source sequence.
`F_lens`	S	In the code, the maximum-length source sequence in the batch is said to have length `S`. The actual length of each sequence in the batch (before padding) is stored in `F_lens`.
`x`	x	Encoder RNN inputs.
`h`	$h_{1:S}$	Encoder hidden states. Always refers to the last encoder layer's hidden states, with both directions concatenated.
`htilde_0`	$\tilde{h}_1$	The first decoder hidden state.
`E_tm1`	E_{t-1}	The target token at $t-1$ (previous).
`xtilde_t`	$\tilde{x}_t$	Decoder RNN input at time t (current).
`htilde_t`	$\tilde{h}_t$	Decoder hidden state at time t (current). For the LSTM architecture, this can be a pair with the cell state. Note in the beam search update `htilde_t` also includes paths, i.e. $\tilde{h}_t^{(1:K)}$.
`logits_t`	$\log P(y_t \vert \ldots) + C$	Un-normalized log-probabilities over the target vocabulary at time t (current). Pre-softmax.
`E`	$E_{1:T}$	Target sequence.
`logits`	$\log P(y_t \vert \ldots)_{1:T} + C$	Un-normalized log-probabilities over the the target vocabulary across each time step. Pre-softmax.
`b_tm1_1`	$b_{t-1,1}^{(1:K)}$	All prefixes in the beam search at time $t-1$ (previous).
`logpb_tm1`	$\log P(b_{t-1}^{(1:K)})$	The log-probabilities of the beam search prefixes up to time $t-1$ (previous).
`logpy_t`	$\log P(y_t \vert \ldots)$	Valid (normalized) log-probabilities over the target vocabulary at time t (current). Post-softmax.
`b_t_0`	$b_{t,0}^{(1:K)}$	Decoder hidden states at time t (current). The difference between $b_{t,0}^{(1:K)}$ and $\tilde{h}_t^{(1:K)}$ is contextual: the latter points to the paths in the beam before the update, the former after the update.
`b_t_1`	$b_{t,1}^{(1:K)}$	All prefixes in the beam seach at time t (current).
`logpb_t`	$\log P(b_t^{(1:K)})$	The log-probabilities of the beam search prefixes up to time t (current).
`c_t`	c_t	Context vector (for attention) at time t (current).
`alpha_t`	$\alpha_{t,1:S}$	Attention weights over all source times at target time t (current).
`e_t`	$e_{t,1:S}$	Energy scores (for attention) over all source times at target time t (current).