Assignment2: Neural Machine Translation

COMP542-442: Natural Language Processing

April 27, 2023

1 Overview

This assessment aims to make you familiar with seq2seq models and experiment with them in a machine translation task. You are going to submit your work via Blackboard.

1.1 Turkish-to-English Dataset

In this assignment, you are going to work on a machine translation task by using wmt16 Turkish-to-English translation dataset adapted from Mukayese. This corpus is available at the link 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 Turkish *.f) in which every line is a sentence. Here, sentence alignment has already been performed for you. That is, the nth sentence in one file corresponds to the nth sentence in its corresponding file. 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.

1.3 Some Notes

Instead of scaled dot-product attention scores $\text{score}(a, b) = |a|^{-1/2}\langle a, b\rangle$, we'll use the cosine similarity between vectors a and b:

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

Where $0 < \varepsilon \ll 1$ ensures $\text{score}(a, b) = 0$ 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 $h_t = \left[h_t^{\text{forward}}, h_t^{\text{backward}}\right]$. 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

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

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

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 $\left|\tilde{h}_t^{(n)}\right| = \left|\tilde{h}_t\right| / N$ and $\left|h_s^{(n)}\right| = |h_s| / N$ (you may assume N evenly divides the hidden state size).

2 Your tasks

2.1 Setup

You should have 8 files: **a2_abcs.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 Building the encoder/decoder

You are expected to fill out a number of methods in **a2_encoder_decoder.py**. These methods belong to subclasses 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 highlevel 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.py** and **a2_encoder_decoder.py**.

2.2.1 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 $F_{1:S^{(m)}}^{(m)}$ and lengths $S^{(m)}$ and outputs word embeddings for the sequences $x_{1:S^{(m)}}^{(m)}$.

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

2.2.2 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 $E_{t-1}^{(m)}$ and produce an un-normalized log-probability distribution over target words at time step t, called $logits_t^{(m)}$. Logits can be converted to a categorical distribution using a softmax:

$$P(y_t^{(m)} = i | \ldots) = \frac{\exp(logits_{t,i}^{(m)})}{\sum_j \exp(logits_{t,j}^{(m)})}$$

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 $\tilde{h}_1^{(m)}$ given the encoder hidden states $h_{1:S^{(m)}}^{(m)}$.

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 $s_t^{(m)}$.

2.2.3 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 reimplemented, but slightly modified for the attention context.

Two new methods must be implemented for ***DecoderWithAttention***.

DecoderWithAttention.get_attention_scores(...) takes in a decoder state $\tilde{h}_t^{(m)}$ and all encoder hidden states $h_{1:S^{(m)}}^{(m)}$ and produces attention 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_attention_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.

2.2.4 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 full hidden output into N chunks. If it's a little bit too tricky, try starting by writing the case where $N = 1$, i.e. when there's no need for splitting.

2.2.5 Putting it together: the Encoder/Decoder

a2_encoder_decoder.EncoderDecoder coordinates the encoder and decoder. Its behavior 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 $\text{logits}_{1:T^{(m)}}^{(m)}$ 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:

$$\forall n, k, v \cdot b_{t,0}^{(n,k \to v)} \leftarrow \tilde{h}_{t+1}^{(n,k)}$$
$$b_{t,1}^{(n,k \to v)} \leftarrow \left[b_{t,1}^{(n,k)}, v \right]$$
$$\log P\left(b_t^{(n,k \to v)}\right) \leftarrow \log P\left(b_t^{(n,k)}\right) + \log P\left(y_{t+1} = v \mid \tilde{h}_{t+1}^{(n,k)}\right)$$
$$\forall n, k \cdot b_{t+1}^{(n,k)} \leftarrow \text{argmax}^k_{b_t^{(n,k' \to v)}} \log P\left(b_t^{(n,k' \to v)}\right)$$

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.

2.2.6 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**.

2.3 The training and testing loops

You are expected to implement training and testing loops in **a2_training_and_testing.py**. The BLEU metric codes are already implemented and provided for you. In ***a2_training_and_testing.compute_batch_total_bleu(...)***, you are given a 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 dataset.

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.

2.4 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. There are a few ways you can do this:

1. If you have access to your own GPU, you may run this code locally and report the results.
2. You can work on Google Colab to access free GPUs.

Even on a GPU, the code can take upwards of few hours to complete in full. Be sure to plan accordingly!

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 sentences into sequences of IDs. Suffice to say that you 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.

- Generate vocabularies
  python a2_run.py vocab TRAIN_PATH e vocab.e.gz
  python a2_run.py vocab TEST_PATH f vocab.f.gz

- Split train and dev sets
  python a2_run.py split TRAIN_PATH train.txt.gz dev.txt.gz

- Train a model without attention
  python a2_run.py train TRAIN_PATH \
  vocab.e.gz vocab.f.gz \
  train.txt.gz dev.txt.gz \
  model_wo_att.pt.gz \
  --device cuda

- Train a model with attention
  python a2_run.py train TRAIN_PATH \
  vocab.e.gz vocab.f.gz \
  train.txt.gz dev.txt.gz \
  model_w_att.pt.gz \
  --with-attention \
  --device cuda

- Train a model with multi-head attention
  python a2_run.py train TRAIN_PATH \
  vocab.e.gz vocab.f.gz \
  train.txt.gz dev.txt.gz \
  model_w_mhatt.pt.gz \
  --with-multihead-attention \
  --device cuda

- Test the model without attention
  python a2_run.py test TEST_PATH \
  vocab.e.gz vocab.f.gz model_wo_att.pt.gz \
  --device cuda

- Test the model with attention
  python a2_run.py test TEST_PATH \
  vocab.e.gz vocab.f.gz model_w_att.pt.gz \
  --with-attention --device cuda

- Test the model with multi-head attention
  python a2_run.py test TEST_PATH \
  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.

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?

2.5 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.pdf.

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, dotproduct attention similar to (Vaswani et al., 2017)[1], additive attention (Bahdanau et al., 2014)[2] and structured self attention (Lin et al., 2017)[3]. Analyze several attention mechanisms, compare their performances, and include discussions towards their reasons of the performance differences.
- Explore the effects of using different 'attention score function' for computing attention score as discussed above and include attention visualization of the different attention functions.

3 Submission requirements

You should submit:

1. The files **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**.
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.pdf. 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 $*.\{e, f\}\cdot$ gz. Only submit the above files. Additional source files containing helper functions are not permitted.

A Suggestions

A.1 Check Discussion Board

Updates to this assignment as well as additional assistance outside tutorials will be primarily distributed via BlackBoard. It is your responsibility to check BlackBoard regularly for updates.

A.2 Start early.

Training on GPUs can be time-consuming. It is crucial to begin your implementation early in order to allocate sufficient time for training. Typically, achieving optimal results requires a few attempts, as you may need to train your models multiple times before identifying and resolving any bugs in your code.

[1] https://arxiv.org/abs/1706.03762
[2] https://arxiv.org/abs/1409.0473
[3] https://arxiv.org/abs/1703.03130

A.3 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 (...). You can run your test suite on teach by calling: python3.9 -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.4 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.

- Create an input and output vocabulary of only 100 words
  python3.9 a2_run.py vocab TRAIN_PATH e vocab_tiny.e.gz --max-vocab 100
  python3.9 a2_run.py vocab TRAIN_PATH f vocab_tiny.f.gz --max-vocab 100
- Only use the proceedings of 4 meetings, 3 for training and 1 for dev
  python3.9 a2_run.py split TRAIN_PATH train_tiny.txt.gz dev_tiny.txt.gz --limit 4
- Use far fewer parameters in your model
  python3.9 a2_run.py train TRAIN_PATH \
  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.

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.5 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, it 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.6 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 $h_t = \sigma\left(W\left[x, h_{t-1}\right] + b\right)$).

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.

4 Disclaimer

This homework is adapted from the CSC401 course, which is offered by the University of Toronto.