Assignment 5: Semantic Parsing with Encoder-Decoder Models

Academic Honesty: Please see the course syllabus for information about collaboration in this course. While you may discuss the assignment with other students, **all work you submit must be your own!**

Goal: In this project you'll implement an encoder-decoder model for semantic parsing. This is conceptually similar to Assignment 4, but involves attention as an extra piece You will have to figure out how to implement the decoder module, combine it with the encoder, do training, and do inference. Additionally, you'll be exploring attention mechanisms as a way of improving the decoder's performance.

Background

Semantic parsing involves translating sentences into various kinds of formal representations such as lambda calculus or lambda-DCS. These representations' main feature is that they fully disambiguate the natural language and can effectively be treated like source code: executed to compute a result in the context of an environment such as a knowledge base. In this case, you will be dealing with the Geoquery dataset (Zelle and Mooney, 1996). Two examples from this dataset formatted as you'll be using are shown below:

```
what is the population of atlanta ga ?
_answer ( A , ( _population ( B , A ) , _const ( B , _cityid ( atlanta , _ ) ) ) )

what states border texas ?
_answer ( A , ( _state ( A ) , _next_to ( A , B ) , _const ( B , _stateid ( texas ) ) ) ) )
```

These are Prolog formulas similar to the lambda calculus expressions we have seen in class. In each case, an answer is computed by executing this expression against the knowledge base and finding the entity A for which the expression evaluates to true.

You will be following in the vein of Jia and Liang (2016), who tackle this problem with sequence-to-sequence models. These models are not guaranteed to produce valid logical forms, but circumvent the need to come up with an explicit grammar, lexicon, and parsing model. In practice, encoder-decoder models can learn simple structural constraints such as parenthesis balancing (when appropriately trained), and typically make errors that reflect a misunderstanding of the underlying sentence, i.e., producing a valid but incorrect logical form, or "hallucinating" things that weren't there.

We can evaluate these models in a few ways: based on the denotation (the answer that the logical form gives when executed against the knowledge base), based on simple token-level comparison against the reference logical form, and by exact match against the reference logical form (slightly more stringent than denotation match).

For background on Pytorch implementations of seq2seq models, check out the helpful tutorial at this URL: `https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html`. Note that the attention description there is Bahdanau attention, so a bit different than the Luong attention (Luong et al., 2015) we've focused on.

Getting Started

Please use Python 3.5+ and PyTorch 1.0+ for this project, following past assignments. You will also need Java for executing the evaluator for logical forms on Geoquery. However, if you don't have Java on your machine or have a hard time getting it to work (sometimes true in Windows environments), **you can pass**

in `--no_java_eval` or just comment out this piece of the code.[1] Denotation match is typically a few percent higher than exact logical form match, and exact logical form match is a strict lower bound.

Data The data consists of a sequence of (example, logical form) sentence pairs. `geo_train.tsv` contains a training set of 480 pairs, `geo_dev.tsv` contains a dev set of 120 pairs, and `geo_test.tsv` contains a blind test set of 280 pairs (the standard test set). This file has been filled with junk logical forms (a single one replicated over each line) so it can be read and handled in the same format as the others.

Code We provide several pieces of starter code:

> `main.py`: Main framework for argument parsing, setting up the data, training, and evaluating models. Contains a `Seq2SeqSemanticParser` class: this is the product of training and you will implements its `decode` function, as well as the `train_model_encdec` function to train it. Also contains an `evaluate` function that evaluates your model's output.
>
> `models.py`: Contains an implementation of an encoder. This is a Pytorch module that consumes a sentence (or batch of sentences) and produces (h, c) vector pairs. This is a vetted implementation of an LSTM that is provided for your convenience; however, if you want to use a different encoder or build something yourself, you should feel free! Note that this implementation does not use GloVe embeddings, but you're free to use them if you want; you just need to modify the embedding layer. **Note that updating embeddings during training is very important for good performance.**
>
> `data.py`: Contains an `Example` object which wraps an pair of sentence (x) and logical form (y), as well as tokenized and indexed copies of each. `load_datasets` loads in the datasets as strings and does some necessary preprocessing. `index_datasets` then indexes input and output tokens appropriately, adding an EOS token to the end of each output string.
>
> `lf_evaluator.py`: Contains code for evaluating logical forms and comparing them to the expected denotations. This calls a backend written in Java by Jia and Liang (2016). You should not need to look at this file, unless for some reason you are getting crashes during evaluation and need to figure out why the Java command is breaking.[2]
>
> `utils.py`: Same as before.

Next, try running

```
python main.py --do_nearest_neighbor
```

This runs a simple semantic parser based on nearest neighbors: return the logical form for the most similar example in the training set. This should report a denotation accuracy of 24/120 (it's actually getting some examples right!), and it should have good token-level accuracy as well. You can check that the system is able to access the backend without error.

Part 1: Basic Encoder-Decoder (50 points)

Your first task is to implement the basic encoder-decoder model. There are three things you need to implement.

[1] Change line 160 of `lf_evaluator.py` to `msg = ""` as described in the comments there.

[2] Common issues: permissions on the `geoquery` file (`chmod a+x geoquery` will fix this) or spaces in the path (may break things, especially on Windows).

Model You should implement a decoder module in Pytorch. Following the discussion in lecture, one good choice for this is a single cell of an LSTM whose output is passed to a feedforward layer and a softmax over the vocabulary. You can piggyback off of the encoder to see how to set up and initialize this, though not all pieces of that code will be necessary. This cell should take a single token and a hidden state as input and produce an output and a new hidden state. At both training and inference time, the input to the first decoder cell should be the output of the encoder.

Training You'll need to write the training loop in `train_model_encdec`. Parts of this have been given to you already. You should iterate through examples, call the encoder, scroll through outputs with the decoder, accumulate log loss terms from the prediction at each point, then take your optimizer step. You probably want to use "teacher forcing" where you feed in the correct token from the previous timestep regardless of what the model does. The outer loop of the training procedure should look very similar to what you implemented in Assignments 2 and 4. Training should return a `Seq2SeqSemanticParser`. You will need to expand the constructor of this method to take whatever arguments you need for decoding: this probably includes one or more Pytorch modules for the model as well as any hyperparameters.

Inference You should implement the `decode` method of `Seq2SeqSemanticParser`. You're given all examples at once in case you want to do batch processing. This looks somewhat similar to the inner loop of training: you should encode each example, then repeatedly call the decoder. However, in this case, you want the most likely token out of the decoder at each step until the stop token is generated. Then, de-index these and form the Derivation object as required.

After 10 epochs taking 50 seconds per epoch, the reference implementation can get roughly 70% token accuracy and 10% denotation accuracy. You can definitely do better than this with larger models and training for longer, but attention is necessary to get much higher performance. **Your LSTM should be in roughly this ballpark; you should empirically demonstrate that it "works", but there is no hard minimum performance.**

Part 2: Attention (50 points)

Your model likely does not perform well yet; even learning to overfit the training set is challenging. One particularly frustrating error it may make is predicting the right logical form but using the wrong constant, e.g., always using `texas` as the state instead of whatever was said in the input. Attention mechanisms are a major modification to sequence-to-sequence models that are very useful for most translation-like tasks, making models more powerful and faster to train.

Attention requires modifying your model as described in lecture: you should take the output of your decoder RNN, use it to compute a distribution over the input's RNN states, take a weighted sum of those, and feed that into the final softmax layer in addition to the hidden state. This requires passing in each word's representation from the encoder, but this is available to you as `output` (returned by the encoder).

You'll find that there are a few choice points as you implement attention. First is the type of attention: linear, dot product, or general, as described in Luong et al. (2015). Second is how to incorporate it: you can compute attention before the RNN cell (using h_{t-1} and x) and feed the result in as (part of) the cell's input, or you can compute it after the RNN cell (using h_t) and use it as the input to the final linear and softmax layers. Feel free to play around with these decisions and others!

After only 10 epochs taking 20 seconds per epoch, our model using Luong style "general" attention gets roughly 77% token accuracy and 30-45% denotation accuracy (it's highly variable), achieving 80% token

/ 53% denotation after 30 epochs. **To get full credit on this part, your LSTM should get at least 50% denotation accuracy.**

Implementation and Debugging Tips

- One common test for a sequence-to-sequence model with attention is the copy task: try to produce an output that's exactly the same as the input. Your model should be able to learn this task *perfectly* after just a few iterations of training. If your model struggles to learn this or tops out at an accuracy below 100%, there's probably something wrong.
- Optimization in sequence-to-sequence models is tricky! Many optimizers can work. For SGD, one rule of thumb is to set the step size as high as possible without getting NaNs in your network, then decrease it once validation performance stops increasing. For Adam, step sizes of 0.01 to 0.0001 are typical when you use the default momentum parameters, and higher learning rates can often result in faster training.
- If using dropout, be sure to toggle `module.train()` on each module before training and `module.eval()` before evaluation.
- Make sure that you do everything in terms of Pytorch tensors! If you do something like take a Pytorch tensor and convert to numbers and back, Pytorch won't be able to figure out how to do backpropagation.

Submission and Grading

You will upload your code to Canvas (the Gradescope autograders are unable to handle this assignment).

We will have **multiple intakes** of the assignment as with Assignment 3. Precise due dates will be posted on Piazza; the number of intakes will depend on how long it takes us to run each batch. You will then be able to resubmit if there are any issues with your code.

Your code will be graded on the following criteria:

1. Execution: your code should train and evaluate within the time limits without crashing
2. Accuracy of your models on the development set
3. Accuracy on the blind test set: this is not explicitly reported by the autograder but we may consider it

Note that partial credit is awarded even for non-functional solutions. If you have even partially implemented attention, please submit this with your code and we will score it appropriately.

Make sure that the following command works:

```
python main.py
```

This command should run without error and train in the allotted time limits. This should train your best model, whether or not it uses attention.

Grading and Extra Credit As mentioned on Piazza, this assignment has 100 possible points to obtain but will be graded out of 50. That is, it is possible to get up to 200% on this assignment if you get all 100 points.

Your model should get at least **70% token accuracy** to get 50 points; this is roughly what a basic LSTM should be able to get. Credit on the second part (the next 50 ppints of the grade) will then be awarded based on how close you get to an exact match accuracy of **45%**. This is challenging to get to! Credit will be awarded according to the following formula:

$$\min\left(\frac{\text{exact match acc} - 5}{40} \times 50, 50\right)$$

We will run your system twice and take the better result to account for variance.

References

Robin Jia and Percy Liang. 2016. Data Recombination for Neural Semantic Parsing. In *ACL*.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP*.

John M. Zelle and Raymond J. Mooney. 1996. Learning to Parse Database Queries Using Inductive Logic Programming. In *AAAI*.