

HW 4: Semantic Parsing with Encoder-Decoder Models

Goal: In this project you'll implement an encoder-decoder model for semantic parsing. We have provided you already the encoder part of the model that you can refer to and use directly. Your objective here is to figure out how to implement the decoder module, combine it with the encoder, and then do training and 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. 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. 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 (Seq2Seq). 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, instead of Luong attention (Luong et al., 2015) or dot-product attention. **You can feel free to implement whichever types of attention you'd prefer for part 2.**

Getting Started

We've prepared you a starter colab notebook that sets up the environment and installs java (needed for denotation evaluation) for you. The notebook also contains some helper functions ¹ that you may need. Please implement all the sections as indicated.

¹Some of the helper functions are inside the notebook; others are imported from Github repo <https://github.com/realliyifei/cis5300-semantic-parsing>.

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 Overall, the starter code already has an implementation of an encoder. The main parts that you should complete and implement are `Seq2SeqSemanticParser(nn.Module)`, `RNNDecoder(nn.Module)` and `train_model_encdec()`.

To further help you understand the structure of the code:

`main(args)`: We've provided this runner code for you to do argument parsing, set up the data, train, and evaluate your model.

`NearestNeighborSemanticParser(object)`: This is provided as a baseline model. You should run and go through the code first to get some insight of how you can run a model using the code provided, and also see how well a nearest neighbor model can achieve.

`EmbeddingLayer(nn.Module)`: This is an embedding layer similar to the embedding layer you implemented in HW3. It maps a word to its embedding. The difference here is that we are implementing using `nn.module` and with dropout.

`RNNEncoder(nn.Module)`: 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.**

`RNNDecoder(nn.Module)`: This is the decoder part of your encoder-decoder model that you need to finish implementing. Recall that at each step of decoding, a basic decoder is given an embedded input and last hidden states from the encoder.

`Seq2SeqSemanticParser(nn.Module)`: This is the Seq2Seq module that integrates your encoder and decoder. You need to complete the `forward()` function so that it feeds the input through the encoder and decoder and return the loss. You also need to implement the `decode()` function that return the predicted outputs. **Note that the `evaluate()` function imported will call this function to generate outputs to evaluate.**

To get started, you can first run the baseline using the code we provided and observe the results. The baseline is 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 (85 points)

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

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. 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.

Part 2: Attention (35 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.

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 should upload three files to GradeScope:

1. Your output result file, namely "test_output.tsv"
2. Your source code, by converting notebook to a python file, namely "homework4.py"²
3. Your report, namely "homework4.pdf"

Your output result file will be graded on the Gradescope autograder following criteria:

1. **Part I [85 points]** Token level accuracy, range from 41% to 70% linearly. That is, your model should get at least **70% token accuracy** to get 85 points; and lower than 41% token accuracy will lead to zero.
2. **Part II [35 points]** Exact match accuracy, computed by the formula

$$\min \left(\frac{\text{exact match acc} - 5}{40} \times 35, 35 \right)$$

That is, your model should get at least **45% exact match accuracy** to get 35 points.

For the coding part, the full points is 100 points and you are able to get up to 120 points on it. The implementation of basic encoder-decoder LSTM is enough to get at least **70%** token level accuracy to take the points in part I. But to get the full points and even 20 bonus, you must do well on attention as it's awarded based on how close you get to an exact match accuracy of **45%**. This is challenging to get to! Good luck!

For the report part, your report submission is worth 75 points. The rubric of report is attached inside the latex template. Please refer to there.

²You don't need to submit any other starter code

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*.