# CS388 Project 2: Semantic Parsing with Encoder-Decoder Models

**Due date: Tuesday, November 5, 2019 at 11:59pm CT**

**Collaboration:** As stated in the syllabus, you are free to discuss the homework assignments with other students and work towards solutions together. However, all of the code you write must be your own! Your extension should also be distinct from those of your classmates and your writeup should be your own as well. **Please list your collaborators at the top of your written submission.**

**Goal:** In this project you'll implement an encoder-decoder model for semantic parsing. A sample encoder implementation is provided to you and conceptually resembles the encoder you built in Mini 2. 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 as well as some other features of encoder-decoder models as part of your extension.

Note the point breakdown on this assignment (described below). In particular, there is less weight placed on the extension than for Project 1 and you are not expected to be as ambitious, as attention is challenging to implement in and of itself.

## 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 (slide 21 of the attention lecture), so a bit different than what we've been discussing in class.

## Getting Started

Download the data and code from the course website. You will need Python 3, numpy, and Pytorch. See the instructions in Mini 2 for information about installing these. If you want to, you're free to use Tensorflow for this project, though you will need to implement the encoder yourself. You will also need Java for executing the evaluator for logical forms on Geoquery.

**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 the encoder you implemented in Mini 2, 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. See the documentation for `encode_input_for_decoder` in `main.py`, which describes its usage.

`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.

`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%)

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. 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 Mini 2. 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 (20%)

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 on some training run.**

**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.

## Part 3: Extension (10%)

Because attention is a challenging component to implement, you're not expected to be as ambitious with your extension in this project as you were in Project 1. Here are some ideas you might explore.

**Beam Search** Beam search can have two advantages for models like this: incorporating lookahead so the model can avoid making certain bad decisions, and giving you a set of finished derivations to choose from in case some don't compile.[1]

**Batching** Batch processing of inputs provides a great speed boost when training on GPUs, and usually a modest boost on CPUs. For parts of your network, you can simply increase the order of all tensors by one and have the first dimension reflect parallelism across examples. However, for the decoder, you will typically need tensors indexed like [num time steps, num examples in batch, ...] since the decoder runs one timestep at a time but handles all examples in that timestep for that batch. Another factor to consider is that you'll need to run the decoder as long as the longest item in the batch; this will require you to think about what to do with the loss when padding. (Hint: don't incorporate terms into the loss computation after the EOS token. You don't want to encourage production of PAD tokens; they're only for bookkeeping.)

[1] It is perfectly "legal" from an evaluation standpoint to pick the highest-scoring derivation that successfully executes; this doesn't require looking at the gold answer or cheating in any way, merely recognizing that the model's output is obviously wrong.

**Training improvements** Scheduled sampling (Bengio et al., 2015) is a way to make training more reliable in light of noisy outputs. However, it may not make a significant difference. You might also want to incorporate denotations at train time and give the model "credit" for producing any derivation leading to the correct denotation. Doing so is more ambitious, but there are a few different ways to do this, as described in Guu et al. (2017): broadly, you can view this problem as either reinforcement learning or as maximizing the marginal likelihood of the correct denotation. Note that using these methods for training requires implementing beam search as well. You may want to "pretrain" the model using the standard cross-entropy objective and only train for a few iterations with these more time-consuming techniques.

**Copying** You can follow the model of Jia and Liang (2016) and implement a copy mechanism. This is challenging! There is significant bookkeeping surrounding dealing with how to juggle the two distributions (copy vs. generate) and how to train marginalizing out the choice of copy or generate at training time.

## Submission and Grading

The grading for this project uses the breakdowns above, plus 20% for the writeup.

You should submit on Canvas **as three file uploads**:

1. Your code as a .zip or .tgz file
2. Your model's output on the blind test set.
3. A report of around 2-3 pages, not including references, though you aren't expected to reference many papers. Your report should list your collaborators, **briefly** restate the core problem and what you are doing, describe relevant details of your implementation, present results from both your basic CRF model as well as with different system variants as part of your extension, and optionally discuss error cases addressed by your extension or describe how the system could be further improved. Your report should be written in the tone and style of an ACL/NeurIPS conference paper. Any LaTeX format with reasonably small ($\approx$1" margins) is fine, including the ACL style files[2] or any other one- or two-column format with equivalent density.

**Slip Days** You may apply slip days to this project as described in the syllabus.

## References


Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *arXiv*.

Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. 2017. From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood. In *ACL*.

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*.


[2] Available at `http://acl2017.org/calls/papers/`