# Assignment 5: Transformers

**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 a Transformer encoder from scratch to learn how they work. You'll then use an encoder-decoder model for semantic parsing, which combines the ideas from Part 1 with ideas from Assignment 4. Rather than implement this yourself, you'll instead use Huggingface Transformers, a popular open-source library for NLP. You'll then explore different inference methods for this model.

## Part 1: Building a "Transformer" Encoder (50 points)

In this first part, you will implement a simplified Transformer (missing components like layer normalization and multi-head attention) from scratch for a simple task. **Given a string of characters, your task is to predict, for each position in the string, how many times the character at that position occurred before, maxing out at 2.** This is a 3-class classification task (with labels 0, 1, or $> 2$ which we'll just denote as 2), structurally similar to the language modeling task in Assignment 4. This task is easy with a rule-based system, but it is not so easy for a model to learn. However, Transformers are ideally set up to be able to "look back" with self-attention to count occurrences in the context. Below is an example string (which ends in a trailing space) and its corresponding labels:

```
i like movies a lot 
00010010002102021102
```

We also present a modified version of this task that counts both occurrences of letters before *and after* in the sequence:

```
i like movies a lot 
22120120102102021102
```

Note that every letter of the same type always receives the same label no matter where it is in the sentence in this version. Adding the `--task BEFOREAFTER` flag will run this second version; default is the first version.

`lettercounting-train.txt` and `lettercounting-dev.txt` both contain character strings of length 20. **You can assume that your model will always see 20 characters as input.**

**Getting started** Run:

```
python letter_counting.py --task BEFOREAFTER
```

This loads the data for this part, but will fail out because the Transformer hasn't been implemented yet. (We didn't bother to include a rule-based implementation because it will always just get 100%.)

**Starting point (not graded)** Implement Transformer and TransformerLayer for the BEFOREAFTER version of the task. You should identify the number of other letters of the same type in the sequence. This will require implementing both Transformer and TransformerLayer, as well as training in `train_classifier`.

Your Part 1 solutions **should not** use `nn.TransformerEncoder`, `nn.TransformerDecoder`, or any other off-the-shelf self-attention layers. You should only use Linear, softmax, and standard nonlinearities to implement Transformers from scratch.

**TransformerLayer** This layer should follow the format discussed in class: (1) self-attention (single-headed is fine; you can use either backward-only or bidirectional attention); (2) residual connection; (3) Linear layer, nonlinearity, and Linear layer; (4) final residual connection. With a shallow network like this, you likely don't need layer normalization, which is a bit more complicated to implement. Because this task is relatively simple, you don't need a very well-tuned architecture to make this work. You will implement all of these components from scratch.

You will want to form queries, keys, and values matrices with linear layers, then use the queries and keys to compute attention over the sentence, then combine with the values. You'll want to use `matmul` for this purpose, and you may need to transpose matrices as well. Double-check your dimensions and make sure everything is happening over the correct dimension.

**Transformer** Building the Transformer will involve: (1) adding positional encodings to the input (see the `PositionalEncoding` class; but we recommend leaving these out for now) (2) using one or more of your TransformerLayers; (3) using Linear and softmax layers to make the prediction. As in Assignment 4, you are simultaneously making predictions over each position in the sequence. Your network should return the log probabilities at the output layer (a 20x3 matrix) as well as the attentions you compute, which are then plotted for you for visualization purposes in `plots/`.

**Training** follows previous assignments. A skeleton is provided in `train_classifier`. We have already formed input/output tensors inside `LetterCountingExample`, so you can use these as your inputs and outputs. Whatever training code you used for Assignment 4 should likely work here too. NLLLoss can help with computing a "bulk" loss over the entire sequence.

Without positional encodings, your model may struggle a bit, but you should be able to get at least 85% accuracy with a single-layer Transformer in a few epochs of training. The attention masks should also show some evidence of the model attending to the characters in context.

**Your task** Now extend your Transformer classifier with positional encodings and address the main task: identifying the number of letters of the same type **preceding** that letter. Run this with `python letter_counting.py`, no other arguments. Without positional encodings, the model simply sees a bag of characters and cannot distinguish letters occurring later or earlier in the sentence (although loss will still decrease and something can still be learned).

We provide a `PositionalEncoding` module that you can use: this initializes a `nn.Embedding` layer, embeds the *index* of each character, then adds these to the actual character embeddings.[1] If the input sequence is `the`, then the embedding of the first token would be $\text{embed}_{\text{char}}(\text{t}) + \text{embed}_{\text{pos}}(0)$, and the embedding of the second token would be $\text{embed}_{\text{char}}(\text{h}) + \text{embed}_{\text{pos}}(1)$.

Your final implementation should get **over 95% accuracy** on this task. **Our reference implementation achieves over 98% accuracy in 5-10 epochs of training taking 20 seconds each using 1-2 single-head Transformer layers (there is some variance and it can depend on initialization).** Also note that **the autograder trains your model on an additional task as well.** You will fail this hidden test if your model uses anything hardcoded about these labels (or if you try to cheat and just return the correct answer that you computed by directly counting letters yourself), but any implementation that works for this problem will work for the hidden test.

[1]The drawback of this in general is that your Transformer cannot generalizes to longer sequences at test time, but this is not a problem here where all of the train and test examples are the same length. If you want, you can explore the sinusoidal embedding scheme from Attention Is All You Need (Vaswani et al., 2017), but this is a bit more finicky to get working.

**Debugging Tips** As always, make sure you can overfit a very small training set as an initial test, inspecting the loss of the training set at each epoch. You will need your learning rate set carefully to let your model train. Even with a good learning rate, it will take longer to overfit data with this model than with others we've explored! Then scale up to train on more data and check the development performance of your model. Calling `decode` inside the training loop and looking at the attention visualizations can help you reason about what your model is learning and see whether its predictions are becoming more accurate or not.

Consider using small values for hyperparameters so things train quickly. In particular, with only 27 characters, you can get away with small embedding sizes for these, and small hidden sizes for the Transformer (100 or less) may work better than you think!

## Part 2: Semantic Parsing with Seq2seq Models (50 points)

### 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).

### Getting Started

Please see the `requirements.txt` distributed with the assignment for package requirements. In particular, Huggingface Transformers requires up-to-date packages.

**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.

There are a few more steps taken to convert to a usable format for HuggingFace training. First, `convert_to_hf_datase` pads the inputs and labels to be square tensors, one input example per row in the input tensor and one target decoder output per row in the labels. The longest input is 23, the longest output is 65. With a BART model, -100 is the output pad index.[2] Second, we prepare an attention mask. This is a matrix the same size as the inputs with 1s in positions corresponding to "real" inputs and 0s in positions corresponding to pad inputs (e.g., those past the length of the corresponding input example). Note that because of this matrix, the input pad index shouldn't actually matter because attention to these tokens is zeroed out and they will not impact the decoder.

We then returns a dict consisting of three things:

```
encodings = {'input_ids': inputs, 'attention_mask': attention_mask, 'labels': labels}
```

**BART** We base our code on the `BartForConditionalGeneration` model from HuggingFace. However, we are just using this configuration for simplicity; we are not actually using the BART pre-trained weights (which we haven't discussed in class yet). Setting aside the pre-training, `BartForConditionalGeneration` is an encoder-decoder Transformer model in the style of (Vaswani et al., 2017). Our hyperparameters give a modest-sized model

**`decode_basic`** This function calls `generate` from the trained model to produce hypotheses for each development example, then truncates at the first occurrence of EOS and de-indexes the predictions to form the actual decoded hypothesis.

**Code** We provide several pieces of starter code:

`main.py`: Main framework for argument parsing, setting up the data, training, and evaluating models.

`models.py`: Contains code to initialize, train, and do decoding with a `BartForConditionalGeneration` model.

`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.

`utils.py`: Same as before.

Next, try running

```
python main.py
```

Unlike past projects, this will actually train and evaluate a fully working model! It will take a bit of time to run. It saves the outputs to a `models/` directory, which you can then load by using `--eval_from_checkpoint` later.

The system reports two metrics: (1) Token-level recall: what fraction of the gold tokens exactly match the prediction at the same position in the sequence. Although this can fail catastrophically if a model's output is misaligned by the gold by just one token, it can still give you an idea of whether a model is getting "partial credit" or not. (2) Exact match: how often the entire predicted sequence exactly matches the gold.

[2] `https://huggingface.co/transformers/v4.0.1/model_doc/bart.html\#transformers.models.bart.modeling_bart._prepare_bart_decoder_inputs`

Theoretically what we most care most about is the denotation (the answer the logical form gives to the question); however, on this dataset, it tracks very closely with exact match and requires an extensive set of Java libraries to evaluate.

**Your Task: `decode_oracle` and `decode_fancy`**

Your job is to explore possible extensions to basic decoding of seq2seq models with two extensions.

Both of these approaches will center around **reranking**. Reranking involves getting a set of $n$ options (usually from beam search) and returning one that may not have scored the highest under the original model using some sort of auxiliary objective.

You can start with `decode_basic` as a template for how to use `generate` from HuggingFace. You can configure `num_beams` and `num_return_sequences` to run beam search and see the outputs.

**Oracle** Your first task in reranking is to compute an *oracle* over the beam. An oracle is essentially a "cheating" reranker: you should look at the options returned from beam search, score them, and return the one with the highest score. This is an effective debugging tool: it tells you whether your general reranking code is working correctly and can improve accuracy. Furthermore, it tells you what the theoretical limit of a reranker is: if you know that the oracle performance is only 60%, then there's no way that any reranker you implement could achieve better than that.

The general steps are to generate with beam search, iterate through the beam, score each of the options against the gold standard (this is the "cheating" part), and return the option with the best score.

**Fancy** Second, you will implement a real, non-oracle reranker over the beam. You are free to explore different techniques that you might find interesting.

What we suggest is to focus on a rule-based constraint: making sure the right expressions or literal constants are used. You might observe that many errors are of the form "*what states border texas*" and then the generated logical form contains the word "`california`". By preferring options in the beam that use the correct literal constants (`texas` in this case), you can do significantly better! You can use the provided `const_list` to get a list of literal constants.

**Your final "fancy" approach should get an exact match of above 51% on the development set, and your oracle approach should get above 55%.**

## Submission and Grading

You will upload your code *and your trained model* to Gradescope.

Your code will be graded on the following criteria:

1. Execution: your code should evaluate within the Gradescope time limit without crashing
2. Accuracy of your models on the development set for both parts
3. Accuracy on a hidden test for Part 1

**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 letter_counting.py
python sem_parsing.py --eval_from_checkpoint --decode_type ORACLE --model_load_path
checkpoint
```

```
python sem_parsing.py --eval_from_checkpoint --decode_type FANCY --model_load_path
checkpoint
```

## References

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *arXiv*.

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