CS 224N - Project Milestone
Compositional Pre-Training for Semantic Parsing with BERT

Arnaud Autef
Stanford Unversity
arnaud15@stanford.edu

Simon Hagege
Stanford Unversity
hagege@stanford.edu

Abstract

Semantic parsing - the conversion of natural language utterances to logical forms - is a typical natural language processing technique. Its applications cover a wide variety of tasks, such as question answering, machine translation, instruction following or regular expression generation. In our project, we investigate the performance of Transformer-based Encoder-Decoder models for semantic parsing. We compare a simple Transformer Encoder Decoder model built on the work of [3] and a Encoder Decoder model where the Encoder is BERT, a Transformer with weights pre-trained to learn a bidirectional language model over a large dataset. Our architectures will be trained on the semantic parsing data using data recombination techniques described in [2]. The main contribution of our work is the use of BERT and transformer-based structures in the semantic parsing setting and the analysis of their potential benefits on such tasks. We are mentored by Robin Jia from Stanford University's Computer Science Department.

1 Approach

In this project, we implement, train and evaluate two Transformer-based encoder decoder architectures on popular semantic parsing tasks. The first architecture is the classical encoder decoder architecture described in [3]. The second architecture uses BERT as the encoder, a pre-trained bidirectional language model with a Transformer architecture. We describe them below:

1.1 Semantic parsing inputs

In the semantic parsing tasks we consider, inputs are natural language sentences and target outputs are logical statements corresponding to their input. To feed input natural language sentences to our architectures, we take the following steps:

- Input sentence is *tokenized* using WordPiece [4]: sentence if broken down into its *tokens* which correspond to its different words and some sub-word units ("wordpieces"). This approach limits the likelihood of encountering unknown tokens in semantic parsing datasets.
- Each input is thus a list of WordPiece tokens, and we use d_{model} dimensional trainable look-up embeddings of those tokens to feed them to our Transformer encoders.

1.2 TSP: Transformer Semantic Parser

We name TSP our simple encoder decoder semantic parser and describe its different components.

Encoder The encoder is composed of N stacked Encoder Transformer layers, as described in [3]. Each Encoder Transformer layer is built as follows:

- Each input sequence in the batch is a tensor whose length is the number of tokens in the input sequence and where each element is a d_{model} dimensional WordPiece token embedding.
- The Transformer architecture does not use recurrence or convolutions. Therefore, to make sure that the representation of each token in the input tensor depends on its position in the sequence, a positional encoding is added to the input tensor.
- The positionally-encoded input is first fed to a Multi-Head, Self-Attention sublayer.
- The output is directed to a Feed Forward sublayer, which is a simple feed-forward neural network with one hidden layer and one output layer. The dimension of the input layer is d_{int} and differs from d_{model}.
- Tensor dimensions are kept to d_{model} in the Transformer layer. Between the sublayers, we apply residual connections, layer normalization, and dropout.

Input sequences are fed to the first Transformer layer. The outputs of each Transformer layer is the input of the next Transformer layer.

Positional Encoding of inputs Consider an input tensor of length L (number of tokens in the input sequence) and dimension d_model (dimension of the token embeddings). The positional encoding added to any such input tensor is a tensor with the same sizes and whose entries are sinusoid functions that will take different values depending on the dimension $1 \leq i \leq d_{model}$ and the position in the sequence $1 \leq pos \leq L$. The sinusoid formulas we use are the following, they come from a very detailed blog post about Transformers from Harvard NLP which helped us a lot in our implementation[1]:

$$PE(pos,\ 2i) = \sin\left(pos/10000^{2i/d_{model}}\right)$$
$$PE(pos,\ 2i+1) = \cos\left(pos/10000^{2i/d_{model}}\right)$$

Attention mechanisms Attention models link queries Q to key value pairs K, V. The output of this model is a weighted sum of the values, where the weights depend on the *compatibility* of the queries with the keys. The model attends to the values V depending on how relevant their keys K are to the queries Q. We described in more details multi-head, self-attention mechanisms with scaled dot-product attention in our project proposal.

Decoder The decoder is composed of N stacked Decoder Tranformer layers, each adding a Masked Multi-Head Attention sublayer to the Transformer layers we described for the Encoder. Each Decoder Transformer layer is built as follows:

- The target sequence is positionnally-encoded and fed to a Masked Multi-Head Attention sublayer. Here, we feed the target output of the Decoder, and the self-attention mechanism should respect the autoregressive nature of the decoding procedure: an autoregressive mask is applied to zero out weights of illegal positions in the attention output.
- This first embedding of the target becomes the query Q of a multi-head attention sublayer with keys K and values V equal to the output of the encoder.
- As before, a Feed-Forward layers completes the block. Residual connections, layer normalization and dropout are still applied between sublayers.

Model output The output the of N Transformer Decoder layers is fed to a linear layer followed by a *Softmax* to obtain a probability distribution over the tokens in the WordPiece vocabulary.

1.3 BSP: BERT Semantic Parser

We name BSP our encoder decoder semantic parser where we replace the simple Transformer Encoder of TSP by BERT, a Transformer sentence encoder, trained to learn a bidirectional language model.

[1]http://nlp.seas.harvard.edu/2018/04/03/attention.html

BERT overview

- **Architecture:** BERT's architecture follows the Transformer Encoder architecture we described above
- **Pre-training**: The BERT encoder is trained to learn a language model on large sentence datasets: BooksCorpus (800M words) and English Wikipedia (2,500M words). Two tasks are defined in [1] to learn a *bidirectional* language model and described in more details in our project proposal:
  - · *Masked Language Model*
  - · *Next sentence prediction*

In our project, we hope that the BERT Encoder will let BSP improve on TSP's performance. The two main assets of the BERT Encoder seem to be the large amount of data and bidirectional language modelling task from which it learns embeddings of natural language sentences. We hope that BSP will benefit from rich and relevant embeddings of its inputs from the beginning of the training, allowing it to learn better representations, faster than TSP.

1.4 Implementation with Pytorch

We based our implementation on the following sources:

- HuggingFace's pre-trained BERT GitHub repository [2]: we use their BERT tokenizer to split our inputs into tokens, and use their BERT pre-trained model as the Encoder of our BSP model.
- Harvard NLP's blogpost about transformers [3]: we directly use the code they expose for positional encodings and the masking of the Transformer Decoder inputs. Apart from those two methods, we coded our Tranformer Encoder and Decoder ourselves.
- The structure of the neural machine translation model in homework 5 inspired our implementation of the global Encoder-Decoder models (TSP and BSP), although the code inside each function is entirely different.
- Our code is made public on a GitHub repository[4].

2 Experiments

In this section, we describe our experimental approach and the results obtained so far.

2.1 Data

We use the dataset Geoquery [5], which is the standard for semantic parsing tasks. It basically contains at each line a single question (input) (`what is the highest point in florida ?`) and its logical utterance (output) (`_answer(A,_highest(A,(_place(A), _loc(A,B), _const(B,_stateid(florida))))))`. We split this dataset in three different files: one used as training set containing 600 data points, one for development set containing 100 data points and one for testing containing 280 data points.

Data recombination The compositionally pre-training process based on three different synchronous context-free grammars has enabled us to augment the data using three different methods: *entity*, *nesting* and *concat*. The entity rule swaps one entity (`florida` in our example) for any other (ex: `maine`). The nesting proceeds to the combination of several questioning tasks (`what is the highest montain in A` in one question and `what states border A'` become `what is the highest mountain in states border A'`). Eventually, the concat$-k$ rule proceeds to the combination of k different questions into one. Using these three rules, we have augmented by a factor 2 our training as well as dev sets and studied the results using each compositional method.

[2]https://github.com/huggingface/pytorch-pretrained-BERTdoc

[3]http://nlp.seas.harvard.edu/2018/04/03/attention.html

[4]https://github.com/simhag/Compositional-Pre-Training-for-Semantic-Parsing-with-BERT

2.2 Evaluation methods

To evaluate the performance of our models on GeoQuery, we will retain two evaluation metrics at the end of this project:

- Strict accuracy : the rate of model outputs that strictly match target outputs in the test set.
- Denotation match accuracy: the evaluation metric used in [2] and most of the literature on GeoQuery.

However, at this stage we are still to implement the denotation match metric. Also, our Transformer-based Encoder Decoder models output sequences of tokens for each input sentence example, but we are yet to implement a faultless method to map those token sequences to a string we could match with target logical forms. Therefore, for now we compared the sequences of tokens generated by our baseline model, with the sequences of tokens (the tokenization) of targets, rather than directly comparing string sequences.

In this context, we came up with three different metrics.

Strict evaluation The strict evaluation accuracy corresponds to the share of output tokens sequences ($\hat{y}_i$) which perfectly match the tokens sequence of the target query (y_i):

$$\frac{1}{n}\sum_i \mathbb{1}(y_i = \hat{y}_i)$$

Jaccard evaluation A much less strict evaluation metric, which consists in computing the *Jaccard similarity* between the model output - target query sets of tokens:

$$\frac{1}{n}\sum_i \text{Jac}(y_i, \hat{y}_i)$$

where

$$\text{Jac}(y_i, \hat{y}_i) = \frac{\#(Y_i \cap \hat{Y}_i)}{\#(Y_i \cup \hat{Y}_i)}$$
$$Y_i = set(y_i) \quad \hat{Y}_i = set(\hat{y}_i)$$

Strict Jaccard evaluation An evaluation metric closer to strict evaluation, corresponding to the share of model output - target tuples for which the set of their tokens perfectly match (Jaccard similarity is 1.):

$$\frac{1}{n}\sum_i \text{Jac\_strict}(y_i, \hat{y}_i)$$

where

$$\text{Jac\_strict}(y_i, \hat{y}_i) = \mathbb{1}(\text{Jac}(y_i, \hat{y}_i) = 1)$$

2.3 Experimental details

As a baseline model, we selected our TSP Transformer Encoder Decoder model with a reduced number of parameters. We limit ourselves to $N = 2$ Transorfmer layers in the Encoder and in the Decoder.

We trained the model for 200 epochs, using an Adam Optimizer with a learning rate of 10^{-3}. The number sublayer parameters are also reduced compared to [3], we set $d_{model} = 128$, $h = 8$, $d_k = 16, d_{int} = 512$. Dropout regularization is set at a rate $p_{drop} = 0.1$. We trained four different models. One base model on the genuine training and development sets from GeoQuery. One model on the training and development sets augmented by entity rule, with a factor 2. One model on the training and development sets augmented by nesting rule, with a factor 2. And a last model on the training and development sets augmented by a concat-2 rule.

We used two different decoding methods on the test sets: a classical greedy decoding method and a beam search with a beam size of $k = 5$.

The multi-head attention architecture enables efficient tensor computations and the number of parameters in this baseline model is kept low, so that we could train it locally on CPU, with a training time averaging 2.5 hours.

2.4 Results

We report the results obtained for our different models below:

Table 1: Results obtained on the test set

Augmentation method	Decoding method	Jaccard	$\text{Jac}_{\text{strict}}$	Strict
None	Greedy	0.898	0.529	0.471
	BEAM	0.897	0.532	0.475
Entity	Greedy	0.937	0.582	0.675
	BEAM	0.938	0.579	**0.682**
Nesting	Greedy	0.945	0.725	0.654
	BEAM	0.947	**0.732**	0.664
Concat−2	Greedy	0.952	**0.732**	0.657
	BEAM	**0.954**	**0.732**	0.661

The results are higher than what we expected for this simple baseline model, trained on a CPU with only 2 Transformer layers and a vanilla Transformer instead of a BERT Encoder. As expected, BEAM search improves results compared to Greedy decoding. Data recombination leads also seems to lead to higher accuracy, in line with the results from [2]. However, such large improvements are suspicious, and we believe they stem from a slight difference in the evaluation procedure which is hard to properly describe here in a succinct way: data recombination basically twitches GeoQuery samples, in the test set as well, and the WordPiece tokens of those twitched samples may be easier to predict. Such ambiguity will be raised when we implement the *denotation match* evaluation metric evoked at the beginning of the previous section.

3 Future Work

For the rest of the project we will focus on the following:

- Proper performances evaluation: Instead of comparing tokens sequences, we need to implement a *detokenize* method to compare strings directly, and implement the knowledge-based evaluation metric used in most of the literature.
- Larger TSP: Train on a Virtual Machine with GPUs our TSP model with higher parameters, especially $N = 6$ layers for the Encoder-Decoder Transformer, comparable to those used in [3].
- BSP: Train our BSP mode, with a pre-trained BERT Encoder instead of a vanilla Transformer.
- Label Smoothing and better optimizer: label smoothing is a way to regularize the outputs of our decoder. We will also use a Adam optimizer with varying (increasing then decreasing) learning rate to train our larger models, following ideas from [3].
- Analyses of results. Finally, we will investigate the role of data recombination rules with our two architectures and compare our performance with traditional RNN-based neural semantic parsing architectures. We will look deeper into the current literature to asses the interest of our results.

Acknowledgments

Throughout this project, we will be advised by Robin Jia from Stanford University's Computer Science department.

References

[1] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[2] Robin Jia and Percy Liang. "Data Recombination for Neural Semantic Parsing". In: *CoRR* abs/1606.03622 (2016). arXiv: 1606.03622. URL: http://arxiv.org/abs/1606.03622.

[3] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[4] Yonghui Wu and al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[5] Luke S. Zettlemoyer and Michael Collins. "Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars". In: (2005), pp. 658–666. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1%5C&smnu=2%5C&article%5C_id=1209%5C&proceeding%5C_id=21.