Semantic Parsing as a Seq2Seq Problem

Anish Acharya
University of Texas at Austin
anishacharya@utexas.edu

1 Introduction

Semantic parsing is the task of translating text to a formal meaning representation such as logical forms or structured queries. There are several decades of history associated with the classical NLP task of semantic parsing.

In this project we pose the problem of semantic parsing in a machine tranlation framework where the source is the standard text input while the target being the logical form. An intuitive way to think about it to think of logical form as another language and then treat this as a translation problem.

In this setting, we use a sequence to sequence style encoder-decoder architecture to do semnatic parsing. We particularly work on the geo-query dataset which in the downstream is used to do QA. The inputs are plain english form questions while the outputs/targets are logical forms that can be used to query knowledge-graphs to get answers.

2 Method

In all the experiments we tried different recurrent models to encode the source (input) sentence into a single vector often referred to as the context vector. This vector was then decoded by a second recurrent network which learns to output the target (output) sentence by generating it one word at a time.

In the rest of the section we briefly describe different components of our parser for completeness and also provide our modeling choices. (Sutskever et al., 2014)

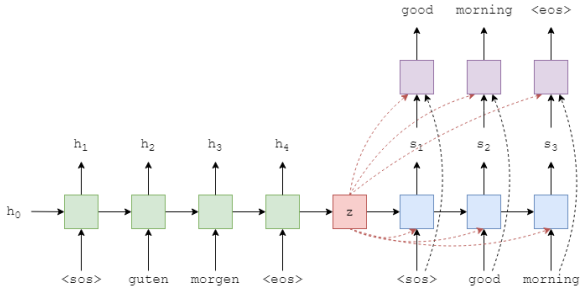


Figure 1: General Encoder Decoder Framework

2.1 Encoder

In the encoder stage the goal is to read the input sequence and embed it into a single context vector. Typically, for language tasks the encoder is some sort of a sequence model. While CNN and other models has also been used in literature, in our experiments we only tried sequential models like LSTM, GRU, vanilla-RNN.

2.2 Decoder

In a seq-to-seq task the decoder is often the most complex and challenging component. The decoder is usually another recurrent model that takes the context input from the encoder, current target timestep and prediction from the previous timestep to predict the next timestep.

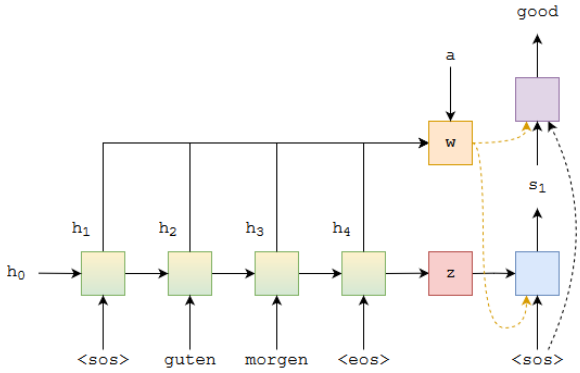


Figure 2: RNN Decoder

2.2.1 Teacher Forcing

Decoder can be viewed as a language model where given the encoded context and previous timestep prediction we predict the current time step output. During decoding/ text generation, since the states at previous time step is used to predict the next one and in the initial epochs when the model is weak, it is often beneficial (also helps in faster convergence) to feed the actual token from the last time step instead of the predicted token by the decoder.

2.2.2 Attention

Encoding the entire sentence into a fixed size vector often implies information loss due to longer sequences. While people have tried several

ideas like inverting the input sequence, using bi-direction encoding, it still does not completely solve the problem. However, a somewhat recent mechanism attention is able to alleviate the problem to some extent by combining an additional attention vector along with input and last state vectors to the decoder. Intuitively, it allows the decoder to weigh the encoded sequence based on the attention distribution. (Bahdanau et al., 2014; Vaswani et al., 2017)

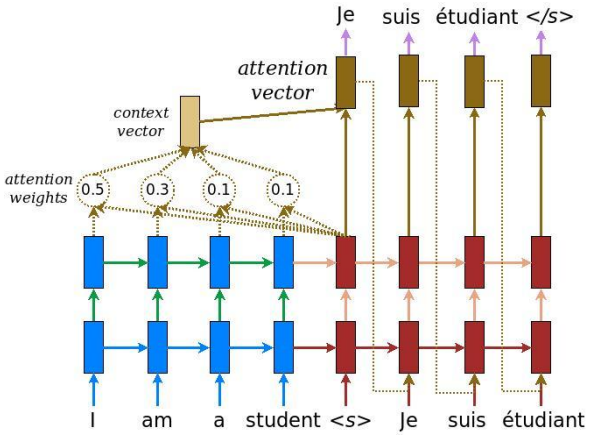


Figure 3: Attention Mechanism

2.2.3 Beam Search Decoding

Beam search decoding often is much more efficient than the greedy decoding. In greedy decoding the token with highest posterior is fed as an input to the next time-step to the decoder for prediction. However, in beam search instead of simply fetching the top we keep track of top-k predictions at each time-step and chose the one with the max (in a multiplication sense) confident path in the beam as our output at the end of decoding.

3 Experiments and Results

Our experiments can mainly be split into two direction of efforts:
(a) Trying different neural architectures for the encoder and decoder models.
(b) Trying ways to improve decoding.

3.1 Encoding

We tried three basic recurrent models as encoder - vanilla RNN, GRU and LSTM. While all of them performed similarly, LSTM seems to perform much better. However, GRU seems to converge faster. Additionally, we experimented with embedding layers, glove initialized encoders perform slightly better. Also, in terms of stacking multiple LSTMs we observed that stacked LSTMs actually out-performs single layer LSTMs and also converges much faster.

Model	Token Acc	Denotation Acc
GRU	0.632	0.09
stacked GRU + Beam Decode	0.707	0.275
LSTM	0.687	0.12
stacked BiLSTM + Beam Decode	0.731	0.326
BiLSTM + Attention	0.711	0.472
stacked BiLSTM + Attention + Beam Decode	0.752	0.525

3.2 Decoding

For the decoder, our model of choice were same as the encoder. While there is no reason to have similar models in encoder and decoder apart from making sure the hidden dimensions and the layers match, we observed mixing two different model actually made the model very unstable and the loss becomes very erratic. This can be an interesting avenue to explore further.
Additionally, we added attention to our seq-t-seq model. Instead of feeding just the encoded input and last states we also feed the attention distributed vector. As expected, attention actually improved the performance by many margins.
We also tried beam decoding, however in this case it does not seem to improve the results by a huge margin but only slightly.
We present a few of the results in the table above. The best accuracy was found using an encoder with 300 glove initialized trainable embedding layer, 4 LSTM layers stacked together and a 256 dim hidden state. followed by a decoder of same dimensions but a 100 dim embedding along with attention and beam width k =3.

4 Additional Experimental Setup

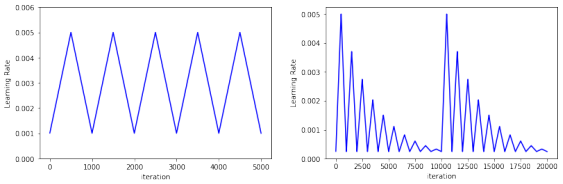
4.1 Cyclic Teacher Enforcing

While teacher enforcing worked very well in terms of converging faster, we observed a constant

teacher force ratio might not be a good idea. While high teacher enforcing is good in the beginning but might hurt inference. So, we propose a decayed teacher forcing ratio is often a good idea.
Surprisingly, we found that instead of a decayed teacher enforcing if we change the ratio by a factor that is correlated to the LR of the model it performs better. Since, we always use adaptive learning rate and the LR often decays slowly during epochs, if we actually slightly perturb teacher enforcing accordingly it performs better.
While this is not concrete, my observation was this particular fuzzy perturb works nicely.
At epoch 0:
Start with 0.7 (high) teacher ratio.
After each batch increase the ratio by lr decay. The intuition is as lr decays increasing the teacher enforcing actually might help the model learn a litle faster.
While decrease the teacher enforce ratio after each epoch by a small fraction.
Future work might include exploring the correlation between the teacher enforce ratio perturbation with the learning rate perturbation.

4.2 Cyclically Annealed Learning Rate

Due to the non-convex nature of the optimization surface of DNNs, gradient based algorithms like stochastic gradient descent (SGD) are prone to getting trapped in suboptimal local minima or saddle points. Adaptive learning rates like Adam and Adagrad try to solve this problem by tuning learning rate based on the magnitude of gradients. While in most cases they find a reasonably good solution, they usually can't explore the entire gradient landscape. Periodic random initialization or warm restarts often helps in better exploration of the landscape while showed that letting the learning rate(LR) vary cyclicaly (Cyclic learning rate(CLR)) 4a helps in faster convergence. Recently (Acharya et al., 2019) proposed a simulated annealing inspired LR update policy we call Cyclically Annealed Learning Rate (CALR). CALR is described in algorithm 1, (Fig. 4b). In addition to varying LR in a CLR style triangular windows, CALR exponentially decays the upper bound LR_{UB} of the triangular window from its initial value. When the learning rate decreases to a lower bound LR_{LB}, we increase the upper bound to its initial value LR_{UB} and continue with the LR updates. This exponential decay ensures slow steps on the optimization landscape. Intuitively, the proposed warm restarts, motivated by simulated annealing, make CALR more likely to escape from local minima or saddle points and enables better exploration of the parameter space. CALR should have similar but less aggressive effect as random initialization or warm restarts of LR. While exponentially decayed small cyclic windows let CALR carefully explore points local to the current solution, cyclic temperature increase helps it jump out and explore regions around other nearby local minima.
Throughout all the experiments we have used CALR in conjunction with Adam resulting in much faster convergence while the accuracy was not significantly improved.



(a) CLR (b) CALR

Figure 4: Comparison of CLR and CALR update Policy

References

Anish Acharya, Rahul Goel, Angeliki Metallinou, and Inderjit Dhillon. 2019. Online embedding compression for text classification using low rank matrix factorization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6196–6203.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Algorithm 1 Cyclically Annealed Learning Rate Schedule

procedure CALR(Iteration, Step Size, LR_{LB}, LR_{UB})
    $LR_{UB} \leftarrow LR_{UB}(init)$
    $LR_{LB} \leftarrow LR_{LB}(init)$
    for each EPOCH **do**
        $LR_{UB} \leftarrow LR_{UB} \times \exp(Decay)$
        if $LR_{UB} \leq LR_{LB}$ **then**
            $LR_{UB} \leftarrow LR_{UB}(init)$
        for each Training BATCH **do**
            LR←CLR(Iteration,StepSize, LR_{LB}, LR_{UB})
    return LR

procedure CLR(Iteration, Step Size, LR_{LB}, LR_{UB})
▷ this procedure implements a triangular window of width StepSize and height $LR_{UB} - LR_{LB}$
    $Bump \leftarrow \frac{LR_{UB} - LR_{LB}}{StepSize}$
    $Cycle \leftarrow (Iteration) \bmod (2 \times StepSize)$
    if $Cycle < StepSize$ **then**
        $LR \leftarrow LR_{LB} + (Cycle)$
    else
        $LR \leftarrow LR_{UB} - (Cycle - StepSize) \times Bump$
    return LR