Abstractive Text Summarization using Attentive Sequence-to-Sequence RNNs

Elliott Jobson emjobson@stanford.edu
Abiel Gutiérrez abielg@stanford.edu

Abstract

In this work, we aimed to emulate the baseline of state-of-the-art abstractive text summarization models, with the intention of exploring different attention mechanisms upon having a decent working baseline. We decided to get inspiration specifically from the sequence-to-sequence recurrent neural networks model built by the IBM Watson team (Nallapati et al., 2016), which achieved outstanding results by implementing several unique features. We built a total of three iterations, adding improvements with regards to the word embeddings, encoder-decoder complexity, and attention. For the third model, we implemented a bilinear attention mechanism, which improved the rate of training loss decrease.

1 Introduction

1.1 Background

Summarization refers to the task of creating a short summary that captures the main ideas of an input text. For our project, we focus on abstractive summarization, which generates the summary through paraphrase. By comparison, extractive summarization works by extracting only words found in the input.

Models for abstractive summarization fall under a larger deep learning category called sequence-to-sequence models, which map from an input sequence to a target sequence. Although sequence-to-sequence models have been successfully applied to other problems in natural language processing, such as machine translation, there remains a lot of room for improvement for state-of-the-art models in abstractive summarization. Despite the fact that state-of-the-art models are able to achieve high ROUGE scores (the standard metric for evaluating abstractive summarization models) on summaries for small inputs, models often lose their ability to summarize key points once inputs become large. Although hierarchical attentive models have had some success in summarizing large inputs, they still have a long way to go. Thus, the task of document summarization is an open problem in natural language processing.

1.2 Plan

We set out with the goal of producing a model for abstractive summarization, starting with the RNN encoder-decoder baseline model implemented by the IBM Watson team (Nallapati et al., 2016). From there, we hoped to explore the effectiveness of different methods for attention in abstractive summarization. Specifically, we planned to implement global attention using the dot product scoring function ($h_t^T h_s$), bilinear form ($h_t^T W_a h_s$) as proposed by (Luong et al., 2015), and a new format in which an output

projection from the hidden states of the RNN encoder to scalar "importance scores" is used in the attention scoring function.

We used the Unannotated Gigaword Corpus, an archive of newswire text, to train our data.

2 Related Work

In this section, we'll establish the baseline used in models for abstractive summarization, then walk through modifications to the baseline that have achieved state-of-the-art results. Despite the clear differences between abstractive summarization and machine translation, the attentional RNN encoder-decoder model proposed in (Bahdanau et al. 2014) has become standard in abstractive summarization. For both tasks, the encoder generates a representation of the input, while the decoder uses these encodings to create the final outputs.

2.1 Baseline

As a baseline model, Nallapati et al., (2016) proposes the use of a bidirectional GRU encoder, a unidirectional GRU decoder, an attention mechanism over the source's hidden states, and a softmax layer over the vocabulary to generate target words. Nallapati then adds a variety of features such as the large vocabulary trick, feature-rich encoding of key words, and use of pointer networks to model rare words; we won't dive into these in order to focus on attention.

2.2 Large Vocabulary Trick

Speeding up convergence and addressing the computational bottleneck caused by the softmax computation over the entire vocabulary, the "large vocabulary trick," proposed by Jean et al., (2014) restricts the decoder vocabulary of each mini-batch to words in the batch's source documents. It then adds words from the target dictionary until the decoder's vocabulary reaches a set size. This approach focuses the model on words that come from the source, making it effective for summarization.

2.3 Out of Vocabulary Pointer

Training the model requires restricting the decoder vocabulary, so it will inevitably run into words it does not recognize. Many models handle this by outputting an "UNK" token in place of the unknown word. Instead, Nallapati et al., (2016) allowed the decoder to choose, at each timestep, between producing a word in regular fashion and generating a pointer to some source word, which is then outputted in the summary.

3 Approach

All three of the models we implemented were based on an encoder-decoder RNN baseline. For our different models, we experimented with attention and word embedding initializations. For abstractive summarization, it is commonplace to use either GRU or LSTM cells for the RNN encoder and decoder. We elected to use LSTM cells for their extra control via their memory unit, although many top models use GRU cells for their cheaper computation time (Nallapati et al., 2016).

3.1 Preprocessing and Dataset

We trained our model on sentence-headline pairs from the Unannotated English Gigaword Corpus, which is commonly used for training models for abstractive summarization. During preprocessing, we pulled out 700,000 pairs of headlines and first sentences, and we trained our model to predict the headline given the first sentence.

3.2 Model Basics

Model 1:

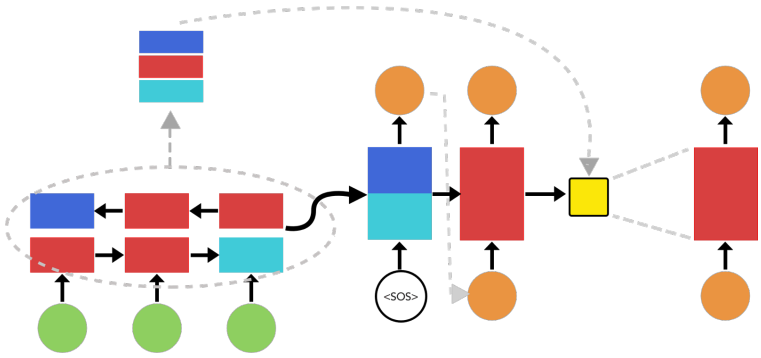
The first model we implemented was a simple unidirectional encoder-decoder LSTM, with randomly initialized word embeddings. As we will mention later, we soon realized that it would be wiser to use pre-trained word embeddings, so we quickly iterated upon this model to create Model 2.

Model 2:

Next, we implemented a model with a bidirectional LSTM encoder and a unidirectional LSTM decoder.

Model 3:

For our last model, we used a bidirectional LSTM encoder, and we added global attention to the LSTM decoder. We used the attention scoring functions specified in section 3.4. The final (forward and reverse) hidden states of the encoder are concatenated and are used as the decoder's first hidden states.



Model 3: Attentive Encoder-Decoder LSTM

3.3 Word Embeddings

In Model 1, we randomly initialized the word embeddings and allowed the model to update them during training. In doing so, we hoped that the learned word vector representations would become more suited for summarization than pre-trained Word2Vec or GloVe vectors. However, we quickly realized that computational and time constraints would prevent the realization of this goal, as it would take a high number of iterations on a large dataset to develop accurate word embeddings for the task.

Given that training word embeddings on a large dataset was not an option, we loaded pre-trained GloVe vectors to create our embedding matrix for Model 2 and Model 3. To simplify the implementation, we used the same embeddings and vocabularies for both the encoder and decoder. However, other models such as the one proposed by Rush et al., (2015) train different embedding matrices for different tasks within, thus improving the vector's notion of semantics within the context in which a word appears.

3.4 Attention

For encoder-decoder neural networks, the use of attention allows for the creation of a context vector at each timestep, given the decoder's current hidden state and a subset of the encoder's hidden states. For global attention, the context vector is conditioned on all of the encoder's hidden states, whereas local attention uses a strict subset of the encoder's hidden states. For our Model 3 implementation, we utilized two different scoring functions, which are used as the weights when averaging the hidden states to produce the context vector.

$score(h_t, h_s) = h_t^T h_s$ (dot product form)

$score(h_t, h_s) = h_t^T W_a h_s$ (bilinear form)

The scoring functions are then used as follows:

$a_t(s) = softmax(s)$

Applying the softmax function to the raw scores creates a probability distribution over the encoder's hidden states, which are then used to create the context vector as follows:

$c_t = \sum_s a(s)h$

c_t is a context vector at timestep t with the same dimensionality as the decoder hidden states. Finally, we use h_t and c_t to compute the next hidden state in the decoder, h_{t+1}:

$h_{t+1} = tanh(W[h_t ; c_t] + b)$

3.4 Handling Varying Sequence Lengths

Because input and output sequence lengths vary, we standardized all inputs to be some max length by padding short sentences with PAD tokens and cutting off longer sentences. This padding was taken into account in both the scoring and loss functions, preventing the model from updating parameters that should not be updated.

3.5 Generating Summaries

At test time, the model's decoder feeds its output as a word embedding input into the next decoder cell. For an LSTM, the output at time t = h_t. Because using hidden states with dimensionalities equal to the size of the model's vocabulary would take too long to train, the outputs of the LSTM decoder cells are actually dense vector representations of the final outputs. By using an affine transformation in the decoder from output h_t, we generate prediction word predictions as follows:

$y_t = Wh_t + b$

$W \in R^{(V, H)}$

Projecting the dense vector through the affine transformation allows us to create a probability across the entire vocabulary size, and then take an argmax in order to be able to index into the embedding matrix and retrieve the input for the next time step.

3.6 Training and Testing

At training time, our model decoders take in article truth embeddings as input at each timestep. This allows the model to learn to output predictions given some inputs x, and prediction history context, y_c. During testing, however, the model's decoder feeds its output as a word embedding input into the next decoder cell.

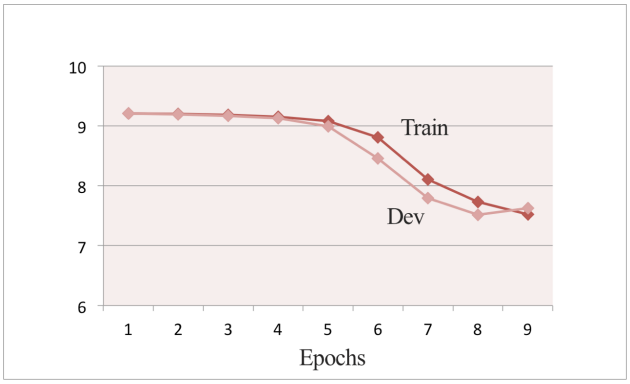
We used Adadelta, an alternative to the Adagrad algorithm, for our gradient descent algorithm. We chose Adadelta because it handles sparse data well, making larger updates to parameters with small frequencies and smaller updates to parameters with large frequencies. This becomes really useful when handling large vocabulary sizes, since common words start to appear much more often than non-common words.

3.7 Scoring and Loss Function

To train the model, we minimize the negative log probability of prediction word **y** given an input x and context y_c.

$loss_t = -log\ p(y_t \mid x, y_c)$

$CE(p, q) = -\sum_w q(w)\ log\ p(w) = -$ log p(w)



Model 1 Loss

4 Experiments

After implementing and successfully training and testing baseline models, we hoped to begin to explore new attention functions. Specifically, we intended to explore a possibility proposed to us by our mentor Abi – to improve the ability of the model to identify key information by allowing the encoder hidden states to output scalar "importance scores," which would be used in calculating the attention scores. We hoped to use these scalars by either directly modifying the attention scores, or by concatenating them to their corresponding encoder hidden state vector.

We had planned to use ROUGE to evaluate our results, but unfortunately, we lacked sufficient time to train a competitive model.

We did, however, test a very different strategy on our last attempt. Rather than training on 100,000 training data points, with 5,000 vocabulary size, and 10 epochs, we switched to 1,000 data points, with 3,000 vocabulary size, and 600 epochs. By epoch 80 we were seeing predicted summaries whose first word made sense according to the ground truths, even though the first word was not directly copied from either the article or the headline.

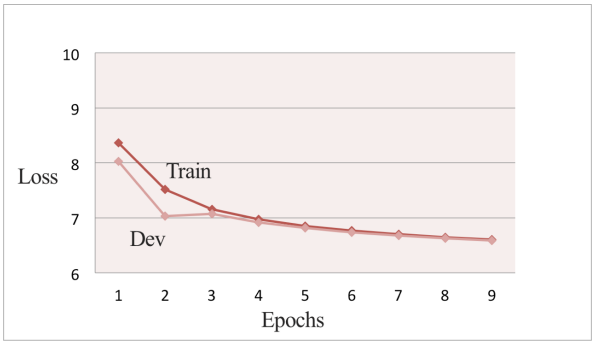
5 Limitations

Our biggest limitation was the lack of time left after finishing the implementation of our attentive encode-decoder RNN model. For lack of a better word, abstractive summarization is a messy task – training and truth data are not well-aligned, compared to

the near one-to-one alignment for machine translation. Therefore, the model must learn to interpret nuances in order to map from input text to summary. As such, models that have achieved success in this task often train for days with a high number of trainable parameters (i.e. from a large vocabulary size, or large hidden state size). As an example, IBM Watson's model (Nallapati et al., 2016) took 8 days to train on Tesla K40 GPUs. Given that we wished to iterate continuously in order to tune hyperparameters, it was difficult to train the model for sufficient time to the point where outputs were meaningful.

6 Results

Though we successfully managed to implement the attentive encoder-decoder RNN summarizer used as a baseline in the model by Nallapati et al. (2016), we lacked sufficient training time, trained our Model 3 on bad data for most of its training time, and ran into multiple problems with the Azure GPU.

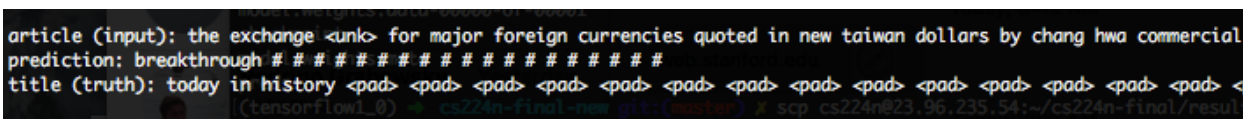


Model 3 Loss

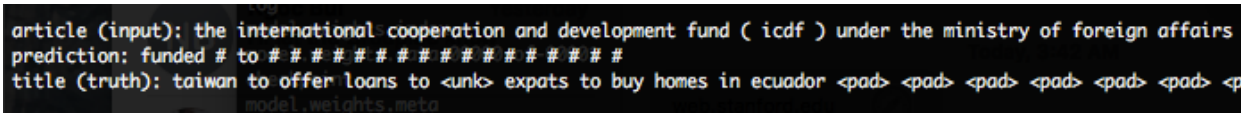
The first time we attempted to train Model 3, we trained it for 10 epochs on 100,000 sentence-headline pairs. We used 200-dimensional encoder and decoder cells, 200-dimensional GloVe vectors, and a vocabulary size of 5,000. The loss decreased from 8.667 to 6.5843, but upon viewing the predictions on the development set, we realized that the model had begun to output purely UNK tokens, for the maximum possible length output. To diagnose the problem, we revisited the data upon which we were training our model, and the method by which we initialized the model's vocabulary.

To initialize a vocabulary of size V, we took the V most common words from the entire preprocessed dataset and used them to create our embedding matrix. However, the V ≤ 10,000 most common words in the entire Unannotated English Gigaword Corpus is still much smaller than the number of unique words in our initial 100,000 example training set. Therefore, many of the truth values were represented as UNK tokens. Consequently, UNK was by far the most common word in the truth values, and the model learned to minimize loss by repeatedly outputting UNK.

Without sufficient time to train our model on a large dataset so that it could generalize well to unseen data, we decided to train our attentive encoder-decoder RNN model on a small dataset, where the vocabulary of the model is identical to the unique words in the training and development sets. This got rid of the issue where the UNK token got printed repeatedly, and even started printed words that made sense, even though these results were evaluated only at epoch 80 of 600:



The word “breakthrough” is predicted for a ground truth summary “today in history”, and “funded” is predicted for a ground truth summary of “taiwan to offer loans to <unk>...”



Although we won’t have time to run through the 600 epochs by the deadline, these results confirm our earlier suspicions -- that this task requires a lot of training time. This also means that our model is definitely headed in the right direction.

7 Conclusion

This was our first experience creating a big data project from scratch, and the amount of learning we have accomplished is incomparable to any other project we have done. Creating all the starter code was a big challenge; despite being able to lean on the code of the previous assignments for this class, we had to understand what each function call did. It was nice to play around with features of TensorFlow and Python that we simply took for granted on earlier assignments, such as the logger, the saver, the arg parser, and the prog bar, among other things.

We had never created scripts to process big amounts of text input, so thinking about the time and memory ramifications of every design decision was a really interesting procedure. When using TensorFlow, we grappled with really poor documentation and strange limitations within the contrib libraries that we had to solve ourselves. We ended up modifying the source code several times in an attempt to debug a problem or two. The most fun part was definitely the exploration of the model once we had the baselines working. Unfortunately, we weren’t able to do this as much as we’d like, but now we know how important it is to leave time to tinker with all possible variables -- as shown above, our last iteration with a small data set and a much more substantial number of epochs seems to be going in the right direction through 80 out of 600 epochs. We are confident that if we run our final model on this data set through the entire epochs (which we’ll do shortly after this submission), it will do a much better job at outputting a meaningful summary.

Finally, getting a taste for what a Computer Science research project looks like has been immensely gratifying. Reading and contrasting different papers to know what new features to add to your model provides for a different learning framework than the one provided in the regular teaching environment. We look forward to taking other classes involving artificial intelligence and deep learning, and use the knowledge gained here to reach working baseline models much quicker, such that we’ll be able to do much more exploring.

7.1 Contributions

Both of us contributed evenly to the project.

References:

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." *[1409.0473] Neural Machine Translation by Jointly Learning to Align and Translate*. N.p., 19 May 2016.

Jean, Sébastien, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. "On Using Very Large Target Vocabulary for Neural Machine Translation – Sampled Softmax."*The Neural Perspective*. N.p., 26 Oct. 2016.

Luong, Thang, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation." *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*(2015).

Nallapati, Ramesh, Bowen Zhou, Cicero Dos Santos, Caglar Gulcehre, and Bing Xiang. "Abstractive Text Summarization Using Sequence-to-sequence RNNs and Beyond."*Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning* (2016).

Rush, Alexander M., Sumit Chopra, and Jason Weston. "A Neural Attention Model for Abstractive Sentence Summarization." *[1509.00685] A Neural Attention Model for Abstractive Sentence Summarization*. N.p., 03 Sept. 2015.