Abstractive Summarization with Global Importance Scores

Shivaal Roy
Department of Computer Science
Stanford University
shivaal@cs.stanford.edu

Vivian Nguyen
Department of Computer Science
Stanford University
vnguyen2@cs.stanford.edu

Abstract

Abstractive text summarization offers the potential to generate human-like summaries because of its ability to select words from a general vocabulary, rather than being limited to the input text like other automatic summarization methods. However, due to the larger vocabulary, a common difficulty with abstractive summarization is choosing the right words to focus on when generating the summary. In this work, we explore the effects of explicitly incorporating a notion of word importance into our seq2seq network at encoding time. We introduce importance in two ways: (i) through tf-idf scores concatenated to our input vectors, and (ii) by modifying our attention scoring mechanism with learned weights during the encoding step.

1 Introduction

Machine text summarization can be performed in two ways: extractively or abstractively. Extractive summarization creates a condensed version of the input text by only using words from the source text to create the summary. Abstractive summarization, on the other hand, is not limited to words from the input and instead generates a summary based on semantic understanding of the source text. It has the ability to paraphrase, compress, and generalize. Currently, the majority of computer text summarization using deep learning is extractive, but this approach is fundamentally limited by the vocabulary of the input text. Abstractive summarization has the ability to create richer summaries due to the lack of constraints, but for this reason poses a more difficult challenge. Our work explores abstractive summarization and expands upon its current techniques by taking into account the inherent importance of each word when generating summaries.

The intuition is that non-stop words and infrequent words, such as proper nouns, are more likely to be important and should therefore appear in the summary. To incorporate this idea, we experiment with two methods: tf-idf scores concatenated to word feature vectors and encoder-generated importance scores multiplied to the attention mechanism scores. Both of these modifications affect the encoding step, the reasoning being that even before the network begins to decode, it should have an idea of which words should receive greater consideration when generating text.

2 Related Work

Sequence-to-sequence neural networks map a source text sequence to a target text sequence. Recent successful applications of this model follow an encoder-decoder framework and have been applied to tasks such as neural machine translation [1][3][10] and speech recognition [1]. Following this work, Rush et al. [9] introduced the idea of using this model for the task of abstractive summarization. Prior to Rush et al.'s work, methods included summarization with a statistical noisy-channel model [2]; syntactic transformation of parsed texts [3]; and grammatical usage of context-free and dependency parsing [11].

The attention-based encoder-decoder model used by Bahdanau et al. [1] for machine translation guided the work of Rush et al. Networks used for abstractive summarization have evolved from feed-forward neural network language models [9] to convolutional recurrent neural networks [4]. In most work, the neural network utilizes LSTM or GRU cells and a beam-search decoder [4][9], but Nallapati et al. [7] also expands further upon current models with a bidirectional encoder and a switching generator-pointer decoder to model rare words.

3 Approach

The overarching task for text summarization is to create a conditional language model that gives us the distribution $p(\mathbf{y}_{i+1}|\mathbf{x}, \mathbf{y}_C; \theta)$, where $\mathbf{x}$ is our input and $\mathbf{y}_C$ is a window of size C of the words preceding $\mathbf{y}_{i+1}$. With neural language models, we're able to learn this distribution directly, as opposed to computing

$$\arg\max_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} p(\mathbf{x}|\mathbf{y})p(\mathbf{y})$$

indirectly by learning $p(\mathbf{x}|\mathbf{y})$ and $p(\mathbf{y})$.

Given this architecture, we can expand more upon our conditional language model. The input text to the model is represented as $\mathbf{x} = [x_1, ..., x_M]$, and each word x_i can be found in the vocabulary V. The target text is represented as $\mathbf{y} = [y_1, ..., y_N]$. Because this is a summarization task, $N < M$. Furthermore, we want to find the values of the vector $\hat{\mathbf{y}}$, which represents a summary of N words, that will maximize the conditional probability $P(\hat{\mathbf{y}}|\mathbf{x}; \theta)$, given our parameters θ. This conditional probability can be expanded as follows:

$$P(\hat{\mathbf{y}}|\mathbf{x}; \theta) = \prod_{t=1}^{N} p(y_t|\{y_1, ..., y_{t-1}\}, \mathbf{x}; \theta)$$

3.1 Models

3.1.1 Baseline: Attentive Bidirectional RNN with LSTM Cells

For our baseline, we began with the Google Brain Tensorflow `textsum` model [8]. It is a sequence-to-sequence model made up of two neural networks: an encoder and a decoder.

Encoder Architecture

The encoder's task is to read in tokens from the input sequence and to generate a fixed-dimension vector C that encapsulates the entire sequence. Because condensing sequences of different lengths to the same fixed-dimensional vector is a difficult task, we use multiple layers of LSTM cells. The decoder will use the hidden states from the topmost layer for its attention mechanism in order to construct context vectors c_i at each decoding timestep i.

Another problem is that a regular seq2seq model only considers the words that precede the current timestep, but we want to take into account dependencies in both directions. Therefore, we use a bidirectional RNN. This means that for each cell, the output for time step t is a concatenated vector of the forward and backward vectors $[o_t^{(f)}; o_t^{(b)}]$.

Decoder Architecture

The decoder's task is represented by the language model described above. It must be able to keep track of the words it has generated and of the input sequence in order to generate the output sequence. The first hidden state of our single-layer decoder is initialized with the last hidden state from the topmost layer of our encoder, thereby taking the input sequence into account. The decoder maintains what it has generated by feeding each generated word back into the LSTM unit at the following timestep. Furthermore, we employ an attention mechanism to generate a context vector c_{i-1} to be used at each timestep i during decoding. The context vector is generated as follows:

$$c_i = \sum_{j=1}^{M} \alpha_{i,j} h_j$$

where

$$\alpha_{i,j} = \frac{exp(e_{i,j})}{\sum_{k=1}^{M} exp(e_{i,k})}$$

Here, h_j is the topmost hidden state of the encoder at timestep j, and $e_{i,j}$ is the score generated by the attention mechanism. In the tensorflow seq2seq library, $e_{i,j}$ is implemented as:

$$e_{i,j} = \text{softmax}(V^{(e)^T} \circ \tanh(h_j^T W^{(e)} + h_i^T U^{(e)}))$$

Following the creation of the output sequence, the final objective is to minimize the sampled-softmax loss of our model on the training set.

$$L = -\sum_{i=1}^{S} \sum_{t=1}^{M} logp(y_t^{(i)} | \{y_1^{(i)}, ..., y_{t-1}^{(i)}\}, \mathbf{x}^{(i)}; \theta)$$

where S is the size of the training set.

Beam Search

To generate our summaries, we use the most commonly used technique in neural machine translation and text summarization: beam search. The beam search implementation in the decoder maintains the top K candidates at each time step. In order to proceed to the next time step, the decoder finds the top K next steps for each candidate, and then selects the top K candidates from the $K * K$ potential candidates it considered.

Model Optimizations

The existing Tensorflow `textsum` model uses a gradient descent optimizer with a linearly decaying learning rate. Following material learned in class, we switched the gradient descent optimizer to Adam optimizer to adapt learning rate to word frequency. In addition, we introduced dropout in between layers of our bidirectional RNN and added L2 regularization on the matrices of our models (but not biases) to prevent overfitting.

3.1.2 Extension 1: Tf-idf Scores Concatenated to Word Feature Vectors

Tf-idf scores are a commonly used NLP statistic to indicate how important a word is. Therefore, we compute a tf-idf score for each word and concatenate it to the embedded word vectors. To compute the tf-idf scores, we precalculated inverse document frequency values for each word in the vocabulary using the training data. Then, we feed in these values to our model and multiply it with the term frequencies, which are computed in real-time. For words not found in our vocabulary, we assign it a high idf score to account for its infrequency.

3.1.3 Extension 2: Encoder-generated Importance Scores Multiplied to Attention Scores

To generate an importance score from our encoder, we applied an output layer to the hidden states in the topmost encoding layer. Each hidden state is multiplied by a weighted row vector, summed with a bias term, and then fed through a ReLU. We use ReLU since β doesn't need to be squished between 0 and 1, or -1 and 1, and so ReLU will prevent us from saturating the output layer.

$$\beta = \text{relu}(h_j^T W^{(\beta)} + V^{(\beta)})$$

With our importance score β, we now multiply it back to the hidden state at each time step in our topmost encoding layer. This modified hidden state is then incorporated in the same attention mechanism as stated above. We call the new attention score $e'_{i,j}$:

$$e'_{i,j} = \text{softmax}(V^{(e)^T} \circ \tanh(\beta h_j^T W^{(e)} + h_i^T U^{(e)}))$$

Again, h_j is the hidden state of the topmost layer of our encoder at timestep j, and h_i is the hidden state of our decoder at decoding timestep i. β allows us to scale the encoding state according to our learned importance of the word and impacts the score computed by the attention decoder when determining the context vector c_{i-1}.

Furthermore, to accurately evaluate our models, our model with the third extension (encoder-generated importance scores) also had the second extension (tf-idf scores concatenated to the word feature vectors). Although seemingly counterintuitive, we must do so because of the natural implementation of the extensions. Tf-idf scores are concatenated to word feature vectors, and the encoder-generated importance scores are multiplied to the attention scores. In order to directly compare the tf-idf extension with the encoder-generated importance score extension, we would have to apply them at the same part of the model.

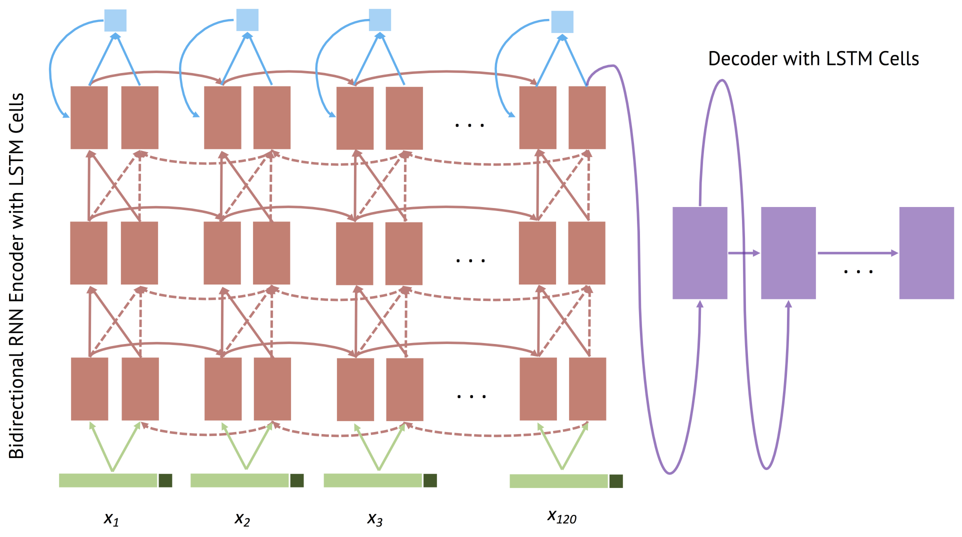

Figure 1: Representation of our model with tf-idf scores concatenated to the word feature vectors and encoder-generated importance scores multiplied to attention scores.

4 Experiments

4.1 Data

We used the annotated Gigaword corpus [6], which is maintained by the Linguistic Data Consortium at UPenn and contains 10 million article-headline pairs from seven different news sources like the New York Times and the Washington Post. Following the practice seen in Rush et al. [9] and Chopra et al. [4], we limited the input to the first sentence of each article due to complexity and time constraints. The underlying assumption is that these popular news sources oftentimes begin their articles with a descriptive first sentence that pertains to the entire article. The reference output is the headline of the article. Building on top of the script used by Rush et al., we extracted headline-article pairs from the Gigaword dataset and then split our data into training, validation, and test sets, having 4.7 million, 400K, and 400K pairs respectively.

In our preprocessing, we discarded all headline-article pairs that were either too short (fewer than 2 words) or too long (over 30 words for the headline or 120 words for the article). Pairs where the headline and article didn't have any non-stopwords in common were also removed. Digits were replaced with the # character. Subsequently, we used the training set to build a vocabulary list, and words seen fewer than five times were replaced with `<unk>`.

Lastly, because our model takes in articles in batches and since articles are not of the same length, we pad each article with a special `<PAD>` token until they are the max length allowed. At train time, we use a mask to prevent these tokens from contributing to the loss.

4.2 Hyperparameter Search

Our baseline model had many hyperparameters, so we began by conducting a search across possible hyperparameter configurations. In order to test our different hyperparameter settings quickly, we reduced our dataset from 4.7 million training pairs down to 50K.

Hyperparameters that we modified were learning rate (η), epsilon value for Adam optimizer (ϵ), dropout rate, L2 regularization weight, batch size, number of encoding layers, number of hidden units, size of word embeddings, and max gradient norm. Due to the large number of hyperparameters, it wasn't feasible to use grid search. Therefore, we randomly selected from reasonable values to create 10 different sets of hyperparameters.

Table 1: Values of randomized hyperparameters for a hyperparameter tuning search.

ID	η	ϵ	DROPOUT	L2	BATCH	LAYERS	UNITS	EMB	NORM
1	0.15	0.1	0.9	0.01	16	4	256	256	8
2	0.15	0.1	0.9	0.001	16	4	128	256	2
3	0.15	1.0	0.5	0.1	16	2	128	128	5
4	0.15	0.1	0.7	0.001	64	4	128	128	8
5	0.05	0.01	0.9	0.1	16	4	128	256	8
6	0.1	1.0	0.5	0.01	64	4	256	256	5
7	0.2	0.01	0.7	0.001	128	4	128	128	10
8	0.15	0.1	0.5	0.01	64	4	256	256	8
9	0.15	0.1	0.9	0.01	32	4	256	256	8
10	0.15	0.1	0.7	0.001	128	4	256	256	8

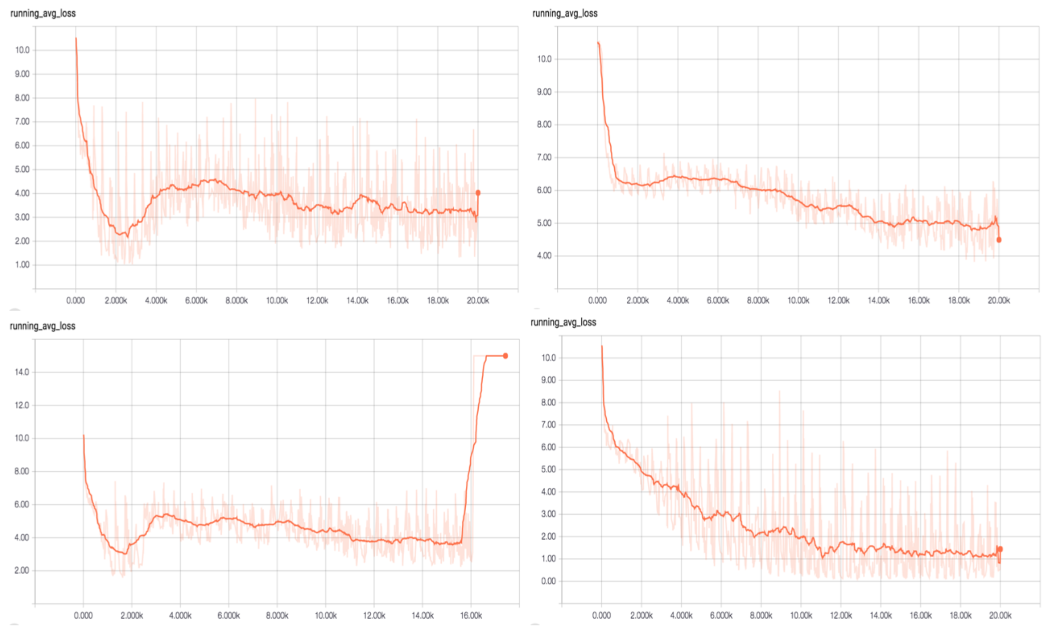

Figure 2: Training loss graphs for models 5, 6, 7, 8 (clockwise from top left).

Ultimately, following our hyperparameter search, we chose the set of parameters that produced the lowest perplexity on the validation data. The hyperparameter values are displayed in Table 2.

Table 2: Optimal hyperparameter values.

η	ϵ	**DROPOUT**	**L2**	**BATCH**	**LAYERS**	**UNITS**	**EMB**	**NORM**
0.15	0.1	0.5	0.001	16	4	256	256	8

4.3 Training

After choosing an optimal configuration of our hyperparameters, we scaled up our training set and trained each of our three models. However, we quickly realized that we would not be able to feasibly train and iterate on a dataset of 4.7 million samples, so instead we randomly selected 200K samples for our training set. We observed similar training times across our 3 models running on the Tesla M60 GPU, each taking about 90 minutes per epoch. With a batch size of 64, we ran each model for about 15K steps in order to train for 5 epochs. This took 7 to 8 hours per model.

We tracked our evaluation loss by running the model on our validation set every 1K steps.

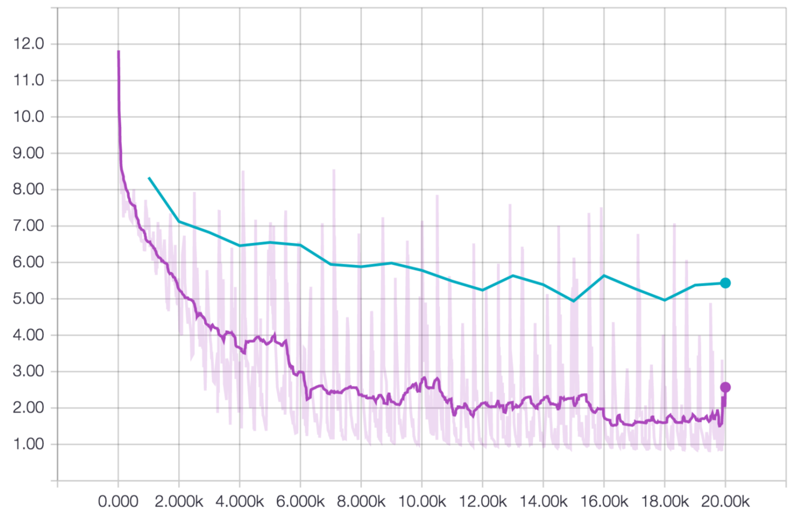

Figure 3: Training vs evaluation loss for our baseline++ model (train = purple, eval = blue)

4.4 Evaluation

In order to evaluate our model, we ran our generated summaries against the reference summaries using ROUGE, or Recall-Oriented Understudy for Gisting Evaluation. Specifically, we used the ROUGE-1 and ROUGE-2 metrics, which calculate precision, recall, and F1 scores for unigrams and bigrams, respectively.

Table 3: ROUGE-1 and ROUGE-2 scores on the baseline, extension 1, and extension 2 models on the test set.

	ROUGE-1	**ROUGE-2**
BASELINE	13.71	4.73
BASELINE+	14.68	4.84
BASELINE++	14.49	4.67

From our results, we're unable to properly evaluate the impact of our extensions, since the baseline performance did not come close to state-of-the-art.

Article: the wall street journal , the asian edition of the us-based business daily , has appointed a new managing director , former hachette <unk> executive christine brendel , a statement said tuesday .
Headline: wall street journal asia names new managing director
Generated headline: <unk> of asia calls for new statement

Article: <unk> their families and supporters mounted a massive demonstration friday in the <unk> valley to defend the region 's industrial heart and soul .
Headline: ##,### demonstrate with human chain to defend german coal industry
Generated headline: demonstrations of german guards against enemies

5 Discussion

From both our development set loss and our final ROUGE-1 and ROUGE-2 scores, we see that our model performed sub-optimally compared to published work in abstractive summarization, with typical ROUGE-1 and ROUGE-2 scores being above 30 and 20, respectively. While our model overfits, despite the use of dropout and L2 regularization, we believe the deviation in train and test results is better attributed to the lack of words in our vocabulary.

We generated our vocabulary from all the words seen in the training set, which for a training set of 200K samples, consisted of 73K tokens. Additionally, a large part of abstractive summarization is developing the language model, which requires not only a token existing in the vocabulary, but also fully learning embeddings for each word. Our vocabulary was significantly smaller than the ones used in Rush et al. and Chopra et al. which were truncated at 200K tokens. Our motivation for explicitly using importance scores was to raise the likelihood of using a non-stop word in our generated summary. The 127K tokens not in our vocabulary (assuming our vocabulary of 73K tokens is a subset of the 200K vocabulary used in papers) are likely to be non-stop words, and so would be words that we were looking to raise the importance of in the first place. Thus, while overfitting was likely an issue, we believe the underlying problem was having a reduced vocabulary, and therefore not being able to handle many new words seen at test time.

If we had more time, the first thing to do would be to train our models on the full 4.7 million training pairs. Earlier, our model took 90 mins per epoch for a training set of 200K samples, so for the full training set, our model would take about 36 hours per epoch. Training on the full dataset would allow our model to cover a larger vocabulary, and thus allow us to extend better to unseen article-headline pairs.

Another way we could attempt to get around the limited vocabulary problem is to start with pre-trained GloVe word embeddings. However, it's uncertain how useful these would be since many of the words in our articles were proper nouns, which are not covered very well by available GloVe embeddings.

6 Conclusion

We attempted to explicitly incorporate the inherent importance of words through two mechanisms on top of an attentive, bidirectional multilayer LSTM. First, we concatenated tf-idf scores of words to their embeddings and fed the modified inputs into the encoder. Additionally, we added an output layer on top of our topmost encoding hidden layer to learn a weight for a word, and then incorporate that weight into the context vector c_{i-1} while decoding at timestep i. We trained on a reduced version of the Gigaword dataset to compare our three models, but our results do not clearly indicate if our two extensions offer improvements to the baseline. Given more time, we would train on the full Gigaword dataset, which would allow us to have more conclusive results about the efficacy of our models.

References

[1] Bahdanau, D., Cho, K. & Bengio, Y. (2014) Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR,abs/1409.0473*.

[2] Banko, M., et al. (2000) Headline Generation Based on Statistical Translation. *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 318-325.

[3] Cho, K., et al. (2014) Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *Proceedings of EMNLP 2014*, pages 1724-1734.

[4] Chopra, S., Auli, M. & Rush, A.M. (2016) Abstractive Sentence Summarization with Attentive Recurrent Neural Networks. *HLT-NAACL*.

[5] Cohn, T. & Lapata, M. (2008) Sentence Compression beyond Word Deletion. *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 137-144. ACL.

[6] Graff, D., et al. (2003) English Gigaword. *Linguistic Data Consortium, Philadelphia*.

[7] Nallapati, R., et al. (2016) Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond. *arXiv*.

[8] Pan, X. & Liu, P. (2016) Sequence-to-Sequence with Attnetion Model for Text Summarization. *GitHub*

[9] Rush, A.M., Chopra, S. & Weston, J. (2015) A Neural Attention Model for Abstractive Sentence Summarization. *EMNLP*.

[10] Sutskever, I., Vinyals, O. & Le, Q. (2014) Sequence to Sequence Learning with Neural Networks. *Advances in Neural Information Processing Systems*, pages 3104-3112.

[11] Woodsend, K., et al. (2010) Generation with Quasi-Synchronous Grammar. *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 513-523. ACL.

Team Members' Contributions

Both members of the group contributed equally. We coded and wrote everything as a pair.