Exploring Teacher Forcing Techniques for Sequence-to-Sequence Abstractive Headline Summarization

Corbin Albert

Supervisor: Dr. Natasa Milic-Frayling

A dissertation presented for the degree of
Masters of Computer Science and Artificial Intelligence

Department of Computer Science
University of Nottingham
United Kingdom
September 14, 2017

Abstract

Every internet user today is exposed to countless article headlines. These can range from informative, to sensationalist, to downright misleading. These snippets of information can have tremendous impacts on those exposed and can shape ones views on a subject before even reading the associated article. For these reasons and more, it is important that the Natural Language Processing community turn its attention towards this critical part of everyday life by improving current abstractive text summarization techniques. To aid in that endeavor, this project explores various methods of teacher forcing, a technique used during model training for sequence-to-sequence recurrent reural network architectures.

A relatively new deep learning library called PyTorch has made experimentation with teacher forcing accessible for the first time and is utilized for this purpose in the project. Additionally, to the author's best knowledge this is the first implementation of abstractive headline summarization in PyTorch. Seven different teacher forcing techniques were designed and experimented with: (1) Constant levels of 0%, 25%, 50%, 75%, and 100% teacher forcing probability through the entire training cycle; and (2) two different graduated techniques: one that decreased linearly from 100% to 0% through the entire training cycle to convergence, and another that graduated from 100% to 0% every 12.5% of the training cycle, often corresponding with learning rate annealing. Dozens of generative sequence-to-sequence models were trained with these various techniques to observe their differences.

These seven different teacher forcing techniques were compared to one another via two metrics: (1) ROUGE F-scores, the most common metric used in this field; and (2) average loss over time. Counter to what was expected, this project shows with statistical significance that consistent 100% and 75% teacher forcing produced better ROUGE scores than any other metric.

These results confirm the use of 100% teacher forcing, the most widely used technique today. However, this throws into question an important assumption by many leading machine learning researchers that dynamic, graduated teacher forcing techniques should results in greater model performance. Questions of ROUGE metric validity, response to more complicated model parameters, and domain specificity are encouraged for further analysis.

Acknowledgements

There are too many people to name I have depended on throughout the completion of this MSc, but I shall try to name but a few.

First and foremost I must thank Alexandra Din, who showed such incredible love, support and selflessness throughout my postgrad.

To my parents, without whom I could never have had such a tremendous opportunity at personal growth.

To Soofi Din, for being the greatest friend and conversationalist I have ever known. And to Amanda Din, for always pushing me to do my best and keeping me on my toes.

Quite especially to Jeremy Howard of fast.ai. Without his deep learning MOOC, I could never have gained the requisite knowledge to understand and build deep neural networks.

To my classmates turned friends at The University of Nottingham for sharing in your passion for this great field of Computer Science.

To The University of Pennsylvania's Linguistic Data Consortium for their generous scholarship for the Annotated English Gigaword Corpus so paramount to the implementation and success of this project.

To my Supervisor, Dr. Natasa Milic-Frayling, for her much appreciated guidance and encouragement.

And to all the faculty of The University of Nottingham whom had the great misfortune have having to put up with me.

Contents

List of Figures

List of Tables

1 Introduction

Article headlines play an increasingly important role in today's information-dense world. Their intent is to be a highly digestible summary of the information contained within the associated article so the reader can decide whether the content will be of interest. But they also need to hook the reader in and grab their attention. In the days of newspaper media, this may have only applied to front-page headlines, enticing a potential reader to buy the paper. In the age of online journalism, however, every article is an opportunity to generate revenue by having readers visit their site. Setting aside the 24-hour news cycle from any one particular publication, resources like Reddit, Twitter and Facebook enable and indeed profit from their users seeing dozens if not hundreds of headlines every day.

This becomes a problem when people start are so inundated with information that they propagate headlines uninformed of the contents within. Take, for instance, a recent report that uncovered that 59% of articles on Twitter were never clicked or read before being shared [4]. Even for issues that an individual deems important, the headline is often the only exposure to the information that said person has had. Headlines can also cause substantive misconceptions for those who do not read the article as well [5].

Furthermore, in the case of misleading headlines, evidence has shown that, even for individuals that read the associated articles, biases are created from the first impressions that headlines form [6]. Even if the misleading element(s) of the headline is addressed within the article, the reader is still more likely to be misinformed after fully reading the article content [6]. The first impressions that headlines provide are nearly impossible to overcome. It truly is difficult to overstate their importance.

Suffice it to say, headlines play a crucial role in our daily digestion of information and deserve the attention of the Natural Language Processing (NLP) community. Unfortunately, we are still many years away from being able to tackle some of the more pressing concerns revolving around headlines and their effect on the public. We are currently still in the early stages of generating headlines from article content, but this means that there are many interesting questions to explore.

Therefore, this research project hopes to contribute to the field of abstractive text summarization (Sec. 2.2) by exploring one of the open questions. Current undertakings within this area of interest have used attentional sequence-to-sequence models (seq2seq; Sec. refseq2seq), a type of Recurrent Neural Network (RNN; Sec. 2.4) to achieve the best results thus far. But teacher forcing (Sec. 2.6), a method used to help train seq2seq models, has not been explored in great depth, nor have the capabilities been realized until recently. Therefore, this project focuses on studying the effects of various teacher forcing techniques for the task of headline summarization, in an attempt to help guide the direction of future seq2seq research.

2 Literature Review

2.1 Introduction

Research on abstractive text summarisation (Sec. 2.2) for article headlines has been primarily undertaken by industry in recent years, with progress being made by Facebook [7, 8], Google [9], and IBM [10]. As stated in Sec. 1, these approaches all implement attentional seq2seq (Sec. 2.5) models, which generate a sequence of words from a different sequence of words. An important open question in seq2seq design, however, revolves around a concept called teacher forcing. To understand the scope and implementation of this work, a review of text summarization, deep neural networks, recurrent neural networks, seq2seq attentional models and teacher forcing techniques is provided.

2.2 Text Summarization

Computer Scientists have been working on automated text summarization techniques since 1958 [11]. Within the subfield, there are two broad categories: extractive and abstractive. Extractive summarization attempts to evaluate the most important words, phrases, and/or sentences within an article and reconstruct a short summary using computational linguistics to place words in the correct order for the summarization. This, however, often leads to very unnatural sounding headlines and relies entirely on words

present within the article [12]. There are simply too many nuances in languages to account for every possible syntactic structure.

Abstractive summarization, by contrast, attempts to view patterns within the article relative to other articles it has seen. It then attempts to generate a summary based upon these patterns and does not rely upon any of the words in the article for constructing its summary. Abstractive summarization is much more akin to human-esque comprehension, reads more naturally and has eluded researchers for decades. In his 2002 descriptive paper, Radev goes so far as to say "true abstractive summarization remains a researcher's dream" [12]. With the renaissance that deep learning has witnessed within the past few years, however, this is beginning to look like a possibility. As such, this project will focus on abstractive summarization contribution.

2.3 Deep Neural Networks

DNNs are one of the most widely used machine learning tools today. Some of their most notable achievements have been the identification of objects in images [13], language comprehension and translation[14, 15], and even mastering some of the world's most difficult board games [16]. Their development was inspired by the synaptic connections of biological neurons, which have continued to serve as inspiration [17]. Neurons receive information from environmental input, such as sight, and proceed to pass and receive electrical signals to and from other neurons via axons to process the input. Eventually, after thousands of passes through different neurons (of which there are approximately 100 billion in the human brain) [18], an understanding of the input is arrived at.

DNNs can be applied to all three broad machine learning categories: supervised (SL), unsupervised (UL) and reinforcement (RL). The most common is SL, in which the expected output of the model is known and can be used to compare against and adjust the machine learning algorithm's hypothesis. Furthermore, in practice, DNNs have two broad categories: recurrent and convolutional. The former ensures a maintenance of state for successive timesteps, while the latter, used primarily in image recognition and generation, is adept at discovering complex patterns. As seq2seq is a supervised recurrent method,

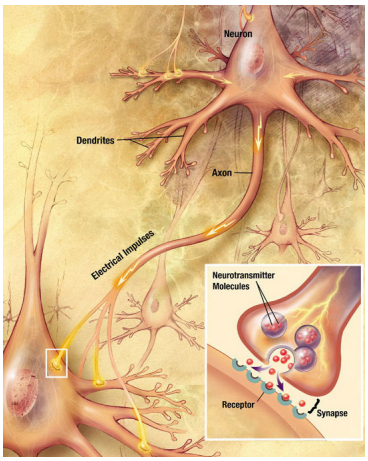


Figure 1: Example of a biological neuron. Neurons carry information along their axons to other neurons using synapses. These neurons then transfer this electrical signal to another neuron and so forth. From [1].

this discussion will be focused on the respective machine learning techniques.

As mentioned previously, DNNs draw inspiration from biological neurons, containing an input layer, hidden layers, and an output layer with numerous nodes in each. Every node, aside from those in the output, relays information to every other node in the proceeding network layer. These nodes simulate the neurons, with connections between them representing axons and synapses.

First, there is an input layer. This is where the pixel values of an image, or the embeddings (See Sec. 2.4.4) of a word in the source sentence of a translation would be fed. There are a series of hidden layers of varying, arbitrary width. A neural network is deep if it has multiple layers (though how many designates an architecture as deep is up for debate). Finally there is an output layer, which could be anywhere from a single node representing a prediction for the next word in a sentence, to a series of nodes detailing what percentage it believes this image is a shoe, a football or a saxophone. So the number of nodes in the input layer and output layer are predetermined based upon the problem you are trying to solve, but the intermediate architecture is up to the neural network designer.

2.3.1 Forward Propogation

An important insight into understanding DNNs is realizing that they are simply a set of matrix multiplications followed by transformation functions. To begin, the input layer passes its values, multiplied by a set of weights, to every node in the next layer. These

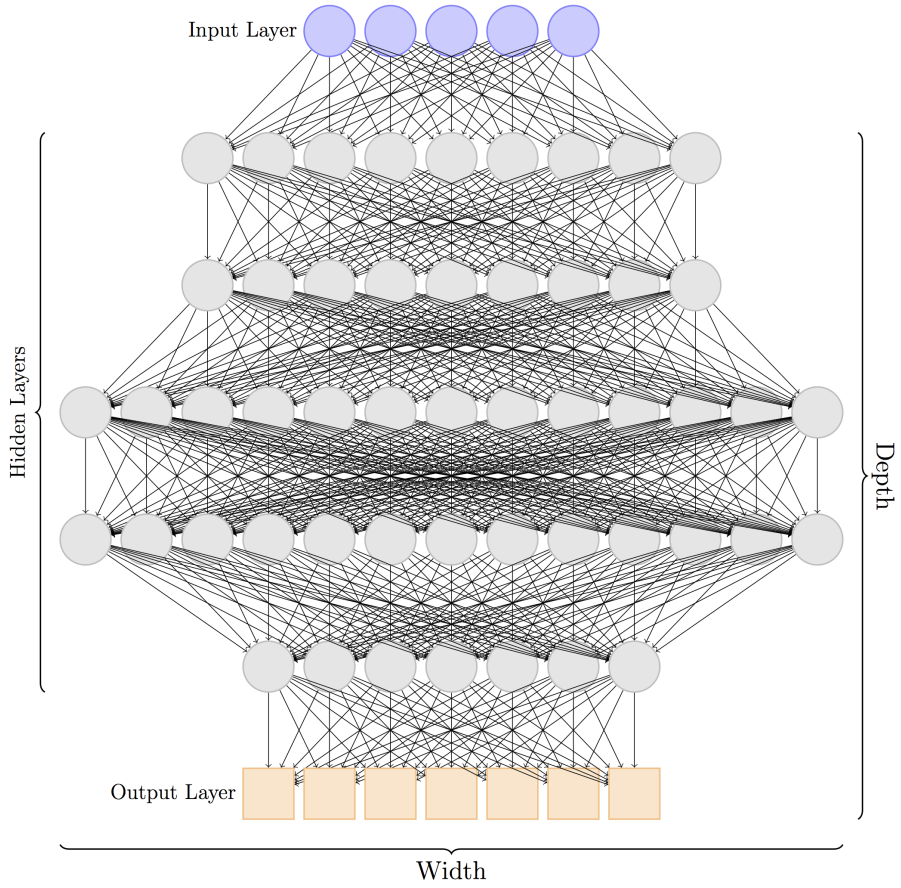


Figure 2: A six layer deep neural network. NB:ANN Architectural naming convention does not include the input layer [2]

weights can intuitively be thought of as a level of importance (this is not accurate, but can help in comprehension). Each node now contains a new value. This value is then transformed, generally using a method called ReLU, which is short for Rectified Linear Unit. This simply means the greater of 0 and the value, or $max(0, x)$ [19].

At this point, the input has been transformed into the first hidden layer. This process is repeated by each successive layer until you get to the output layer. The activation function for the output layer is going to be different and tailored to the specific problem. In the case of seq2seq problems, a sigmoid function, $f(x) = \frac{1}{1+e^x}$, will output a percentage value for what the neural network believes is most likely to be the next word in the sequence. The maximum value will then be selected. So by the time we have reached the output layer, a long string of various matrix multiplications and ReLU transformations has occured. The output layer is simply a function of many other functions of functions. Arriving at the output completes the forward propogation of a single training example.

The reason for the high degree of effectiveness of DNNs is that they can model every function at extremely high dimensionalities [20]. This was the crucial insight into being able to leverage neural networks for such a wide range of problems. Simply by multiplying by a set of weights and transforming with an activation function several times, extremely complex functions can be modeled to a high degree of accuracy.

2.3.2 Cost Calculation

The magic of neural networks is in the weights that each successive layer is multiplied by before being transformed by the activation function. The correct values for these weights are what DNNs are trying to learn. When DNN models are first created, these weights are initialized randomly, so the model will have no predictive capability whatsoever at this point. The prediction will be compared to the actual value, or the next word in the sentence, as is the case with seq2seq. The cost function is generally cross entropy loss, which has the benefits of being positive and trending towards zero as the loss decreases [20]. The specifics of the cost function are not important, but it is critical to understand that when a prediction is wrong, the value of the cost function is greater than zero. By adjusting the weights in each layer, this cost function can be minimized.

2.3.3 Backpropogation

To ensure the weights are changed in the correct direction, backpropogation is used. After the cost function has been calculated, we can take the derivatives of the function described by the current neural network to find the slope of the cost function for all current weights. By using the chain rule to propogate backwards through the entire function, we can find the slope of the cost. If there is a slope to the cost function, this means the cost can be less than it currently is and a small step in the right direction towards minimizing this cost can be achieved.

2.3.4 Gradient Descent

Now that the derivatives have been calculated, a small step towards the minima of the cost function is taken by modifying the weights in the correct direction as deteremined by backpropogation. The size of these steps is determined by a learning rate, which generally must be lowered as the model becomes more and more accurate. There are various methods for calculating this, and intricacies such as momentum to accelerate the learning process, but these are not important for understanding the core concepts. If learning is done one training example at a time, it is called stochastic gradient descent. Eventually, after several epochs (Forward Propagation, Cost Calculation, Backpropogation and Gradient Descent on the entire training dataset), the DNN will have learned the optimal set of weights to minimize the cost function and make accurate predictions or classifications.

2.3.5 DNN Conclusion

Stated simply, DNNs are a series of layers that contain weight matrices that each modify the previous layer before undergoing a transformation. Eventually this produces an output that at first is very wrong, but by taking the derivative of the cost function applied to every layer and minimizing the loss accordingly, DNNs can be trained to be highly accurate machine learning tools. The example presented above is a simple, fully-dense DNN with only one input layer and no sense of state. The previous prediction has no bearing on the next prediction. For understanding the sequence of words in an article, however, a sense of state is important—a problem solved by RNNs [21].

2.4 Recurrent Neural Network

RNNs are so-called because they feed the state of the previous timestep calculation into the next state. This can best be visualized in Figure 3. Here, you can see a sequence of inputs, $X_{0...t}$, that are fed into a hidden state that derives information from the hidden states of previous timesteps.

This combination of new input and old state being input into the new hidden state allows for the model to develop a sense of context from the previous timesteps. To under-

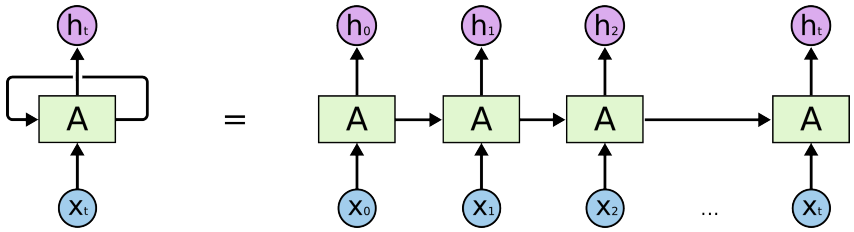


Figure 3: A depiction of a simple, single layer RNN. On the left is the traditional representation which can be unrolled as displayed on the right side of the image. From [3] with permission.

stand this concept of timesteps, imagine the following input sentence (the first sentence is often all that is used in modern headline length article summarization [9, 7, 8, 10]): "New research from XYZ suggests that quick brown foxes have been jumping over lazy dogs at an alarming rate." Each individual word would be the input at it's respective timestep, so X_0 = 'New', X_1 = research, and so forth. In NLP, there are two types of RNNs: character and word. In the case of a character-RNN, this sequence would have 107 timesteps, one for each subsequent character, including spaces and punctuation. A word-RNN would have 20 timesteps. As in DNNs, the hidden states that these inputs are fed into have a set number of nodes that contain an associated weight. Because they continually recieve context from previous timesteps, RNNs can learn spelling, punctuation, and grammatic and syntactic structure without ever being explicitly trained [22].

The original conception of an RNN contained only a simple transformation, as displayed in Figure 4.

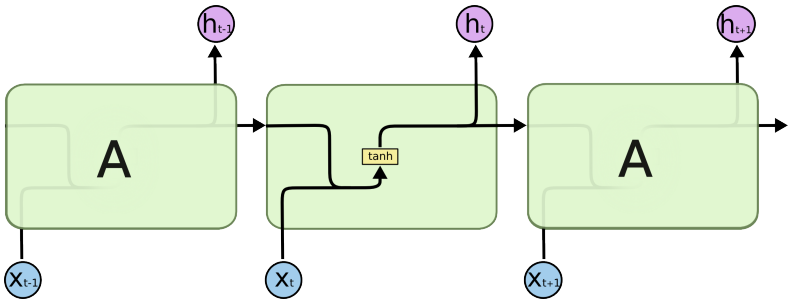


Figure 4: The inner-workings of the hidden state for traditional RNNs. From [3] with permission.

In theory, this simple structure should have worked, but in practice, problems such as exploding gradients prevented them from converging (minimizing the loss function

to a stable, low overall cost) during training [23, 24]. As a result, much research was undertaken to solve this issue, leading to the development of Long Short-Term Memory (LSTM) networks [25], which have since become the defacto standard [22].

2.4.1 Long Short-Term Memory

LSTMs are a significantly more complex type of RNN, as seen in Figure 5.

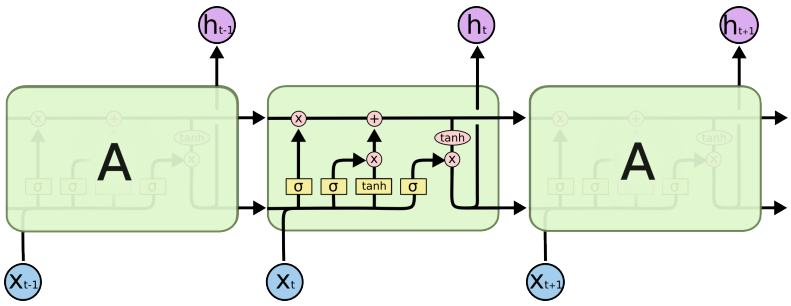
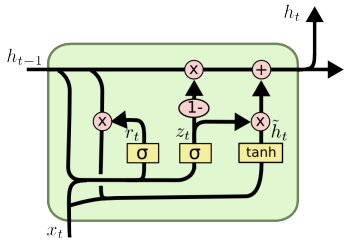


Figure 5: The original LSTM as described in [23]. From [3] with permission.

The details are not significantly important, but the main takeaway is that LSTMs essentially contain miniature neural nets that dictate how much information should flow from certain outputs to the next, called sigma (σ) gates. These σ gates output a value between one and zero to determine the amount of information released to the main hidden state channel, represented by the horizontal bar running across the top of Figure 5 [23, 3]. In this way, a more complex sense of state can be achieved.

2.4.2 Gated Recurrent Unit

The LSTM architecture, developed in 1997, was the only RNN architecture used in practice until 2014 [22] when the Gated Recurrent Unit was developed by [26]. It simplified the LSTM by combining certain functions and getting rid of others.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 6: The GRU as described in [26]. From [3] with permission.

Again, the specifics are not important, but it is important to know that these have become a popular option for RNN architectures. As regards effectiveness, both LSTMs and GRUs seem comparable on average and appear very task dependent [27, 28]. However, because GRUs are slightly simpler and consequently take less time to train, these are more often used than LSTMs.

2.4.3 Bidirectional RNN

RNNs as pictured in Figure 4 only send state in one direction. So, the first word has no context for later words in the sentence. This is ultimately unhelpful, as oftentimes the words picked at the beginning of a sentence are dependent upon the rest of the sentence structure. To address this, bidirectional RNNs were developed [29] as displayed in Figure 7.

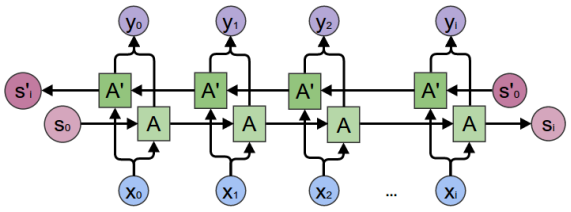


Figure 7: A bidirectional RNN. From [3] with permission.

A bidirectional RNN is just a standard RNN merged with another RNN running through the sequence backwards to ensure that each timestep contains state information about each element both preceeding and proceeding it. These have become the defacto RNN structures used in seq2seq.

2.4.4 Word Embeddings

As a final detail to RNNs, it should be noted that an index to a word is not typically fed into the RNN. Instead, an embedding [30] is created for each word. An embedding is a vector of arbitrary dimensionality that is used to describe characteristics of the word. The values in these vectors will ultimately allow the model to understand whether a word is a noun or an adjective, syntactic structure between words, and any other nuance of language that it can determine from seeing words in differing contexts millions of

times [31]. This allows for a word to have dimensionality, which can be thought of as the characteristics of the word. So instead of feeding a word index to an RNN, a vector of learned weights for each individual word is used as the input.

Understandably, the same word across domains often has the same contextual meaning, so it is not necessary to train all words from scratch. Resources like Stanford NLP's GloVe Embeddings [32] and Google Brain's word2vec [33] have trained word embeddings on dozens of thousands of words across extremely large databases. This allows DNN Engineers to save valuable time training their networks if the Embeddings have already been learnt. It will likely require some tuning after the model has converged, but this will be minimal.

2.4.5 RNN Conclusion

RNNs are used to pass a sense of state from timestep to timestep. These, like any DNN architecture, can be stacked on top of one another for increasingly complex pattern recognition and generation, but due to the complexity of their state-maintenance, they take a very long time to train. There are likely some critical pieces of the puzzle we have yet to solve as regards RNNs, but this is the current state of top-performing NLP RNN architectures. A few more levels of complexity are necessary for seq2seq, which will be dissected presently.

2.5 Sequence-to-Sequence

Seq2seq [34] is the primary method used in NLP for turning one sequence of characters into another sequence. Developed in 2014, it is used heavily in Neural Machine Translation [35, 36], speech recognition [37], video captioning [38], and of course, abstractive text summarization [7, 8, 39, 9, 10]. They consist of two primary pieces—the encoder and decoder, as shown in Figure 8.

The first work on headline-length summarization by [7] used a convolutional neural network (CNN) for the encoder and a context-sensitive feedforward network for the decoder. This was expanded upon by [8], which maintained the use of the CNN for the

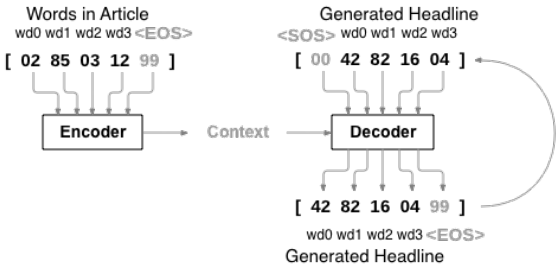


Figure 8: A basic seq2seq model for translating French into English. From [40] with permission.

encoder, but switched to an RNN for the decoder. Today's highest performing architectures, however, use RNNs for both the encoder and decoder [41, 10, 9].

2.5.1 Encoder

The encoder takes the input sentence and converts it into a vector representation, denoted by *Context* in Figure 8. By training on thousands, if not millions, of examples the encoder will have learnt how to convert a sentence into a meaningful vector representation, symbolizing the "essence" of the input sentence. Obviously, the larger this vector is, the more complex and precise the representation is for any given input.

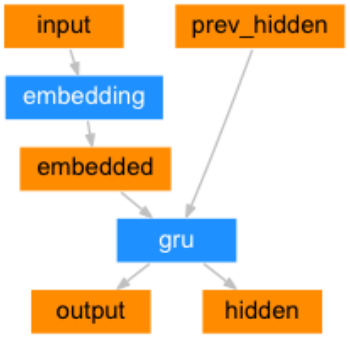


Figure 9: Diagram of an RNN encoder network. From [40] with permission.

While the simple RNN in Figure 4 produced an output at every timestep, this RNN simply continues to feed its previous state into the next timestep to create the context vector.

2.5.2 Decoder

The context vector created by the encoder is then fed as the input to a decoder RNN, the functions of which can be visualized in Figure 10.

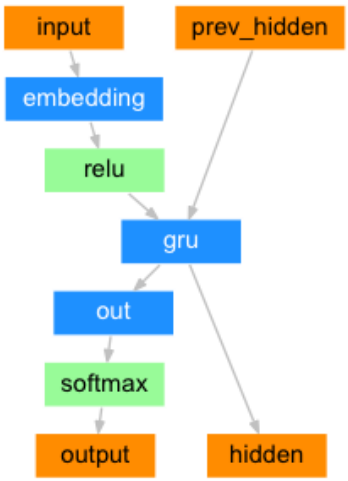


Figure 10: Diagram of an RNN decoder network. From [40] with permission.

At the first timestep of the decoder, a start-of-sentence token, $< SOS >$,will be fed as an additional input. This signals the decoder to predict the first word of the output. In a fully trained seq2seq model, this first word prediction would then be fed into the second timestep, along with the context vector to generate the next word in the sequence. It will continue to do this until the model decides to generate an end-of-sentence token, or $< EOS >$. This then produces the resultant sequence in the seq2seq model.

However, before it has been trained, none of the weights will contain meaningful information for proper sequence generation. This means that during training, you must feed the correct input to the decoder for each timestep and allow it to calculate its loss based upon its prediction and what the next word should have been.

2.5.3 Attentional Decoder

These simpler encoder-decoder models worked relatively well, but a problem persisted in which the decoder could not tell which words from the input of the encoder would have a specific impact on the decoder's output [35]. This is best understood by thinking about translation, in which words at the beginning and end of a sentence can have an impact

on, say, the gender of a noun or types of prepositions. This is shown in Figure 11, in which the relative importance of certain input words on their corresponding output is visualized.

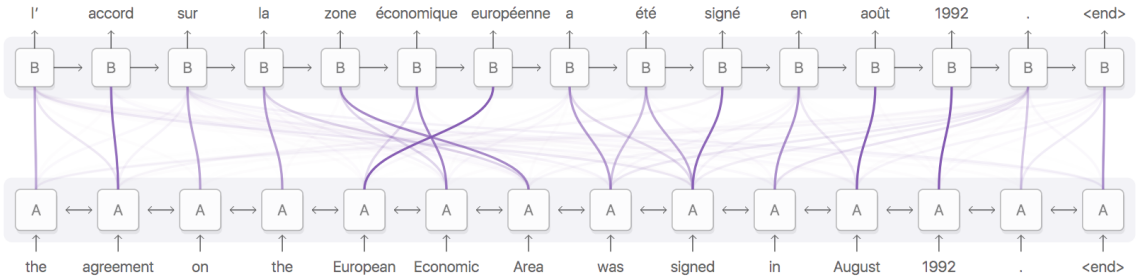


Figure 11: When translating, certain words in the input will have a disproportionate effect on words in the output. From [3] with permission.

To address this, an attentional framework was developed in 2014 [35]. The original attentional model can be thought of as a simple neural net between the encoder and decoder that attempts to determine the strength of the influence that words in the starting sequence will have on a particular output [35]. This was further developed upon in 2015 by implementing different scoring functions [42], the specifics of which are outside the scope of this literature review. Attentional decoders have proven their effectiveness and are used in all sophisticated seq2seq models today, including headline generation.

2.5.4 Seq2Seq Conclusion

RNN encoders fed into attentional RNN decoders represent the architectural framework of the state-of-the-art. Note that the encoder and decoder RNNs can a multiple layers deep by feeding the output of an encoder into another encoder and vise versa. For example, the Neural Machine Language architecture behind Google's Translate service uses 8 encoders and 8 decoder [36]. These allow for far more complex characteristic representations at the cost of far greater training time and computation requirements.

2.6 Teacher Forcing

The last piece to understand is the concept of teacher forcing (TF) [43]. In Figure 8, there is a grey arrow on the right side of the diagram pointing from the *output* of the decoder back to the *input* of the decoder. This is how a trained RNN will generate a sequence. However, when a model's weights have not yet been trained, this is highly ineffective as the output at any given timestep will almost certainly be incorrect. The decoder will then attempt to modify its weights for subsequent time steps based upon an incorrect prior state, which theoretically should result in much longer training time. To combat this, teacher forcing was developed [43] which instead feeds the correct, labeled output back into the decoder regardless of how it predicted the previous timestep.

It was noted, however, that models which were constantly fed the correct prior state were not functioning as well at inference time. It was theorized that this was because the model had developed a dependency on receiving the correct prior state [44], but it wasn't until 2015 that this was addressed with a technique called scheduled sampling [44]. In the models that have been created for headline generation, however, TF has been a binary decision—either always feed the decoder the input or don't at all. This is because the deep learning frameworks used in state-of-the-art models, namely tensorflow and torch, all require pre-defining the neural network's computational graph before runtime, which does not allow the flexibility required for dynamic teacher forcing [45].

However, a relatively new deep learning library called PyTorch builds the computational graph at run-time [46] and is consequently far more customizable [45]. As a result, dynamic teacher forcing can now be implemented and it is the aim of this dissertation to explore the effect of these various TF techniques on model performance.

3 Methodology

To construct the initial seq2seq model, the PyTorch neural network library for python was used. Recall from Section 2.6 that other state-of-the-art solutions are not able to implement dynamic teacher forcing methods due to the fact that the computational graph

must be pre-defined before inference.

Specifically, I used an open source seq2seq pytorch jupyter notebook for neural machine translation[1] as the base model. Several modifications were made to this model, including the implementation of bidirectional RNNs and various teacher forcing methods, as well as redefining input and output parameters for the specific task of headline-length abstractive summarization.

3.1 Data and Preprocessing

The dataset used for this project and all other state-of-the-art models for this task is The University of Pennsylvania Linguistic Data Consortium's Annotated English Gigaword Corpus[2], graciously provided via scholarship, which contains over four million article-headline pairs from seven news publications between 1994 to 2010. Specifically, the LDC data was structured as 1,011 gzipped XML files, each file representing a month of articles from one of the news sources. Because the data was delivered via an external hard drive and I did not have access to a University computer, the dataset had to be processed on a 2015 Macbook Pro. Unfortunately, this meant that the 154 largest XML files had to be discarded due to the computers inability to store their parsed XML trees in memory. It was decided that the dataset should be pared back to $\tilde{1}$00,000 pairs, so 120 articles were sampled randomly from each of the remaining 857 files.

To address the file format, the LXML library was used to unzip and parse the XML tree. After randomly shuffling all of the child nodes of the XML tree, a series of regex substitutions were used to strip the unnecessary part-of-speech tags which accompanied every word in the dataset to transform the necessary data into raw text. It must be noted that all state-of-the-art headline generation models currently only use the first sentence of the article as input, as this is generally found to be sufficient to generate headlines [9]. I maintained this convention within this project, so any reference to an article is only to the first sentence of said article in practice.

One drawback of the LDC dataset is that there is a non-negligable amount of nonsen-

[1]Available at: https://github.com/jph00/part2/blob/master/translate-pytorch.ipynb
[2]https://catalog.ldc.upenn.edu/ldc2012t21

sical headlines that also happen to be very long. To combat this, I restricted the length of headlines to 30 words and articles to a length of 50 words, the same criteria used in the highest-performing model [9]. Any pairs that did not meet both of these criteria were discarded, resulting in a 20.7% discard rate. The remaining 79.3% randomly sampled, fully preprocessed article/headline pairs were then placed into respective lists. The code for extracting the data can be found in Appendix A.

The sentences were then turned into word indexes within a vocabulary list for every headline/article pair, keeping the 200,000 most common words and substituting all other words with an $< UNK >$ token, for unknown. This practice is in line with the highest-performing state-of-the-art model [9]. A $< SOS >$, start of sentence pad, or index 1, was added to the start of every pair. Then each sentence was padded with the $< PAD >$ token, or index 0, on the ends to make each input and output the same length. The data was then split 90% training, 10% test. Lastly, each word was also linked with its associated 100-dimensional GLoVe embedding (See Sec. 2.4.4) if an embedding existed for the word. Otherwise, a random initialization was created for the word to be trained by the model. Unfortunately, due to licensing constraints, I cannot share the contents of any of the data, but the processed, indexed pairs can be seen in Figure 12.

```
art_train[0], hdln_train[0]
```

```
(array([    3,   134,     6,    82,   183,    19,  1120,    23,    29,
         4725,    43,  5378,  1009,     8,  2012,    25,  3703,  3754,
          995,    21,    18,    36,    34,  2814,  2229, 11172,     9,
            3,  3015,   693,   242,   124,  3995,    54,  2468,   104,
            9,     3,   311,   271,    83,     4,     0,     0,     0,
            0,     0,     0,     0,     0]),
 array([  759, 11954,  1034,     8,    25,  3280,     4,  1949,   751,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0]))
```

Figure 12: A preprocessed headline and article pair with zero-padding.

3.2 Difficulties and RNN Architecture Adjustments

When initially setting out to research this issue of teacher forcing, several large issues presented themselves. First and foremost, training DNNs can take significant resources,

some using more than 300 GPUs simultaneously [16]. For my specific task,attempts at adhereing to state-of-the-art model parameters revelaed that fully training such a model from scratch would take over 10 days per experiment on a single GPU, confirmed by [10]. Furthermore, because teacher forcing is a method used during model training, pretrained models could not be used. Therefore, modifications were required to achieve results within a reasonable time.

The current state-of-the-art headline generation model uses four bidirectional encoder layers, four bidirectional decoder layers, a word embedding size of 128, and 256 hidden nodes per LSTM [9]. The greater one can increase any of these dimensions, the greater the model's learned complexities between inputs, outputs and intra-word relationships at the cost of computational requirement for both training and evaluating. This model was trained on multiple GPUs (though the exact number is not specified) and appeared that it would take weeks to converge on a single GPU through a short test run. Because of this, the same general seq2seq architecture was maintained, but all dimensionalities reduced to the following: one layer encoder, one layer decoder, word embedding size of 300, and hidden size of 128 nodes per LSTM. With these parameters, models appear to converge within approximately four hours on the University's Titan X (Pascal) GPU instance (Thadeus machine). Due to this decrease in model complexity, one cannot hope for comparative results to state-of-the-art, so instead, results for various teacher forcing methods will only be judged relative to one another.

As a final precaution, it must be noted that these models are learning sequence generation from headlines that have been chosen by the publications, meaning they are still subject to bias. The idea, however, is that by generalizing well enough amongst a large dataset, biases will naturally be mitigated. I think this is likely incorrect, and that Recurrent Neural Networks will pick up trace biases by the very nature of their capability to understand and interpret nuance to such an incredible degree of dimensionality. That said, developing effective abstractive techniques is the first step in a long road towards the mitigation of editorial bias in online news, and seq2seq is currently at the forefront of development in this arena.

3.3 Experiment Set-up

Recall from Section 2.3.4 that a learning rate is picked by the DNN engineer to change all the weights within the network. This learning rate needs to be decreased as the model gets closer and closer to convergence in a process called learning rate annealing. It was found that 40,000 cycles of generating a random batch of 128 examples, annealed every 5,000 - 10,000 cycles, was adequate for model convergence. The first two sets of 5,000 batches had a learning rate of 0.003; the second two sets of 5,000 had learning rates of 0.001; the third two sets of 5,000 batches had learning rates of 0.0003; the next set of 5,000 had a learning rate of 0.0001; the last set of 5,000 had a learning rate of 0.00003. This progression can be seen clearly in the `multi_train` function in Appendix B.

I trained seven different models on these teacher forcing (TF) methods: graduated, graduated-reset, 100%, 75%, 50%, 25%, and 0%. For the graduated model, the chance of teacher forcing started at 100% and decreased linearly over the course of the entire 40,000 batches of training until eventually 0% teacher forcing was allowed by the last batch. Graduated-reset would gradually decrease the amount of teacher forcing over every 5,000 batches, so full teacher forcing was reset to 100% and tapered to 0% for each of the 8 sets of 5,000 batches. The differences can be further investigated by referencing the `trainEpochs` and `fullgraduated_trainEpochs` functions in Appendix B. The various percentages listed above provided the associated teacher forcing probability throughout the entire training period.

To ensure that the training and test splits were not disproportionately influencing one TF method over another by chance, I trained and tested every model 3 different times on randomized data splits. In doing so, I was able to ascertain that all TF methods on all three tests were statistically similar, and consequently any of the test results could be used for further analysis. See Section refstat for details.

3.4 Evaluation Metrics

When evaluating the effectiveness of various teacher forcing methods, there are two main metrics to condiser: loss over time and ROUGE.

3.4.1 ROUGE

Firstly, to judge generative performance, the metric used in all previous explorations of headline-length text summarization [7, 8, 10, 9, 47]: Recall-Oriented Understudy for Gisting Evaluation, or ROUGE [48], will likewise be used for this project. There are two ROUGE metrics of interest: ROUGE-1, ROUGE-2. For the purposes of this task, ROUGE-1 measures the overlap of each word between the generated summary and the label. ROUGE-2 measures the overlap of adjacent words between the generated summary and the correct label. The top ROUGE scores for each state-of-the-art abstractive model can be compared in Table 1. By measuring these values, we can ascertain how close the

		ROUGE-1	ROUGE-2
Rush	[Facebook]	29.78	11.89
Chopra	[Facebook]	33.78	15.97
Nallapati	[IBM]	35.30	16.64
Liu	[Google Brain]	42.56	23.12

Table 1: ROUGE-1 and ROUGE-2 scores for top performing abstractive models

generated output was to matching words and word pairs from the beginning. While this metric is quite unsophisticated and does not well convey actual comprehension, it is the metric used by all other headline-length abstractive summarization researchers to judge performance and consequently is the metric chosen to be used here.

3.4.2 Loss Over Time

Evaluating visualizations of loss over the training period is another important metric in judging the effectiveness of various TF methods. Recall from Section 2.6 that the reason teacher forcing became a topic of discussion in the first place was the idea that, at the beginning of model training, the model would be generating the wrong output nearly every time. If this incorrect output was constantly fed into the decoder state to predict the next time step, all training afterwards is unhelpful. This would result in far longer convergence times. So, the amount of time it takes for a model to converge is a critical part of the motivation behind teacher forcing. Consequently, the average loss over time was recorded and used to compare the various teacher forcing techniques.

3.5 Statistical Analysis and Data Visualization

To compare and contrast the results of the various TF methods from the three different tests, I used the Kruskal-Wallis Test [49], which tests whether a number of samples are statistically probable to originate from the same distribution. These results can be found in Section 4. As all TF methods over all three tests returned probable to have come from the same distribution, it is safe to assume that all TF methods from all three tests are representative of a general trend, and consequently further analysis only needed to be carried out on one of the tests. To visualize the skewed ROUGE distributions, I created histograms and Q-Q plots using R.

For computing average ROUGE values, I could not discern the methodology used by any of the state-of-the-art researchers. As such, I used what I felt was appropriate given the data distribution. That said, for skewed datasets, the median is typically used and was initially attempted for this analysis. However, the Wilcoxon pseudo-median values gave wildly inaccurate results for ROUGE-2. It would appear that non-normal median-distributed data was not properly accounting for ROUGE values of zero, a statistic that is very important to this specific problem. Consequently, bootstrapping [50], a mean-centric resampling technique, was used to compute these averages instead.

To test for statistically significant differences between the data, the Wilcoxon test [51] was used. This is a method used for paired difference testing on non-normal data distributions. The values reported are probabilities (p-values) that the differences in median of the compared TF methods is 0. This means that smaller p-values make it less of a possibility that the compared means are in fact the same and no statistical significance exists between them. A p-value ≤ 0.05 (5.0E-02) makes the differences statistically significant. Wilcoxon probabilities were calculated relative to three different benchmarks: all TF methods relative to no TF, all TF methods relative to 100% TF, and graduated compared to reset-graduated as a matter of academic interest.

Lastly, a look at loss over time is done as a second evaluation metric, again with only one of the test results. The loss and time information was collected in lists during training. These were then plotted using matplotlib, a MATLAB-esque plotting program

for python and heavily used in jupyter notebooks. These plots have been provided in Section 4.2.

4 Results

After running all seven teacher forcing methods 3 times each, the results for each TF method across the different tests were compared using Kruskal-Wallis, as described in Section 3.5.

4.1 ROUGE Scores

Before diving into ROUGE scores, it is important to understand the distributions of the ROUGE scores for the purposes of statistical analysis. The data is right (positively) skewed, as visualized in Figures 13 and 14.

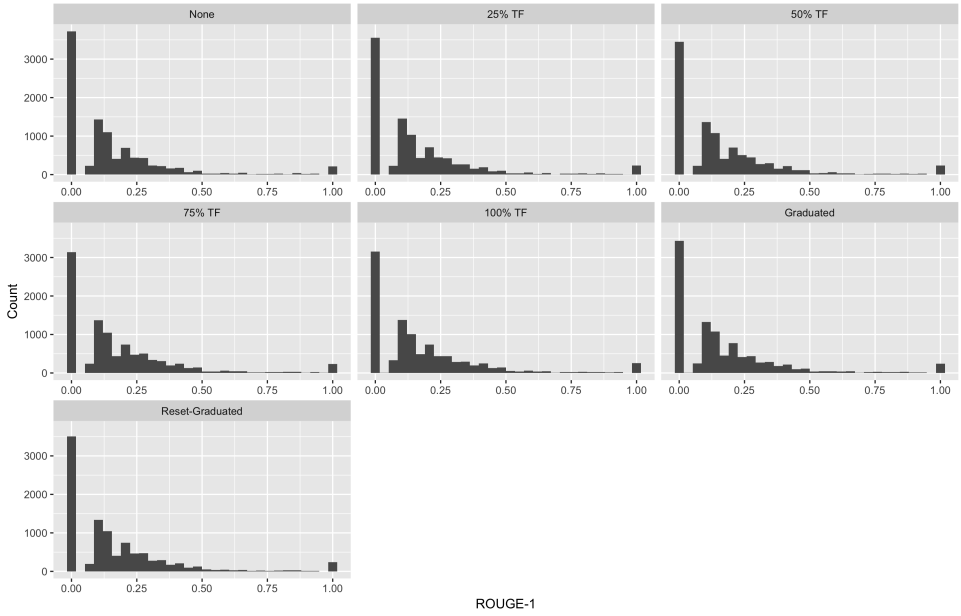


Figure 13: Histograms of ROUGE-1 scores from all TF methods of Test 2

Data non-normality can also be visualized via Q-Q Plots, as demonstrated for ROUGE-1 scores in Figure 15. A normal distribution would lie primarily near the regression line with dots randomly dispersed on either side [52].

Due to this skewedness, a Kruskal-Wallis test was used to compare the ROUGE results

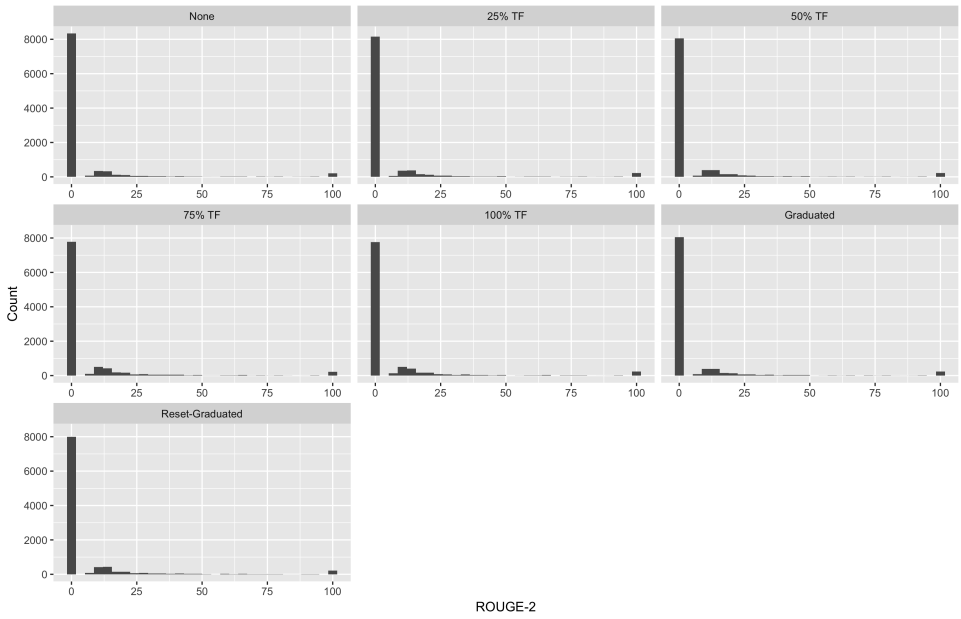


Figure 14: Histograms of ROUGE-2 scores from all TF methods of Test 2

for each TF method across the three different tests. The results of the Kruskal-Wallis test can be viewed in Table 2.

TF Method	ROUGE-1	ROUGE-2
0%	0.481	0.262
25%	0.132	0.201
50%	0.162	0.307
75%	0.465	0.493
100%	0.632	0.577
Graduated	0.175	0.770
Reset-Grad	0.861	0.402

Table 2: Kruskal-Wallis ROUGE-1 and ROUGE-2 p-values for various TF methods across three tests

Recall that values above 0.05 indicate a lack of statistical difference between the distributions of the data. Because all data fell within tolerance, we can safely assume no individual test benefitted from random chance. From here forward, analysis has only been conducted on the test 2. Having visualized the distribution of ROUGE scores and consolidated the analysis, we can now incorporate the appropriate metric for comparing TF methods: Wilcoxon averages and significance testing.

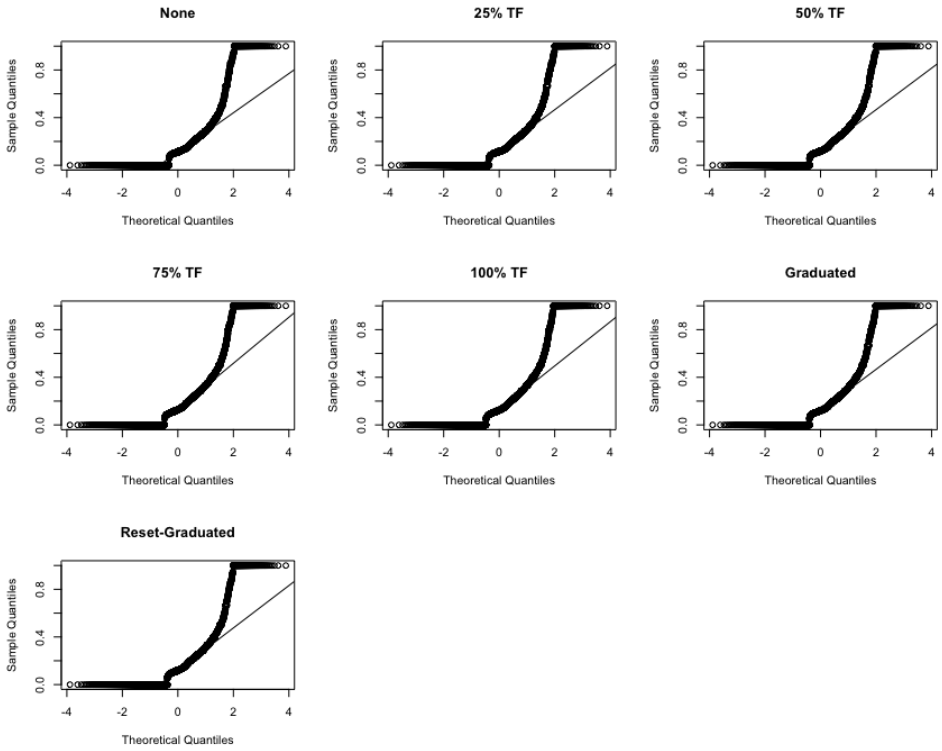


Figure 15: Q-Q Plots of ROUGE-1 scores for each TF Method of Test 2

4.1.1 Averages

Tables 3 and 4 show the averages for ROUGE-1 and ROUGE-2, respectively. It bears repeating that, due to the necessary decrease in seq2seq architecture complexity, state-of-the-art results should not be expected. Rather, we are only hoping to compare various TF methods across the same datasets.

TF Method	Lower Bound	Upper Bound
0%	15.27	16.09
25%	16.04	16.86
50%	16.60	17.45
75%	17.76	18.59
100%	17.58	18.45
Graduated	16.65	17.49
Reset-Grad	16.52	17.36

Table 3: ROUGE-1 bootstrapping averages for various TF methods

Perhaps surprisingly, these results appear to indicate that the highest levels of constantly held TF, 100% and 75%, resulted in the best possible ROUGE scores. This would appear to contradict the accepted wisdom that continual high levels of TF cause genera-

TF Method	Lower Bound	Upper Bound
0%	4.997	5.688
25%	5.510	6.241
50%	5.692	6.403
75%	6.194	6.893
100%	6.223	6.964
Graduated	5.692	6.421
Reset-Grad	5.743	6.433

Table 4: ROUGE-2 bootstrapping averages for various TF methods

tive degradation. To validate these, though, we need to run paired Wilcoxon tests to test for significance.

4.1.2 Paired Wilcoxon Tests

We will run paired Wilcoxon tests on three different reference tests: all TF methods relative to no TF, all TF methods relative to 100% TF, and graduated compared to reset-graduated. Recall from Section 3.5 that the values reported are probabilities (p-values) that the differences in μ of the compared TF methods is 0. P-values > 0.05 represent no statistically significant difference. We will first evaluate p-values with respect to no TF.

TF Method	ROUGE-1	ROUGE-2
25%	2.59E-3	9.77E-4
50%	2.46E-08	3.29E-07
75%	2.20E-16	2.20E-16
100%	2.20E-16	2.20E-16
Graduated	1.43E-08	1.75E-7
Reset-Grad	8.39E-08	6.57E-10

Table 5: ROUGE paired Wilcoxon tests for various TF methods as compared to no teacher forcing

It is immediately obvious that there are substantive differences between all TF methods relative to no TF, the least of which, unsurprisingly, being 25% TF. Coupled with the fact that all TF methods had higher average confidence intervals, it is safe to assume that all TF methods under the parameters of the experiment score a higher ROUGE than not using TF. A simple "greater-than" Wilcoxon test confirms this to be true. Next we will evaluate all TF methods compared to 100% TF, the industry standard at the moment.

Surprisingly, the differences in averages for 75% and 100% in Section 4.1.1 seem to

TF Method	ROUGE-1	ROUGE-2
0%	2.20E-16	2.20E-16
25%	2.73E-10	1.15E-10
50%	1.81E-04	3.55E-06
75%	0.153	0.883
Graduated	2.58E-04	5.78E-06
Reset-Grad	8.14E-05	3.47E-4

Table 6: ROUGE paired Wilcoxon tests for various TF methods as compared to 100% teacher forcing

be substantiated by these paired Wilcoxon tests. We can see that 100% teacher forcing is significantly different and of higher average confidence interval (See Sec. 4.1.1) than all TF methods besides 75%. A "greater-than" Wilcoxon test confirms that these two TF methods have a statistically significantly higher mean than all other TF methods. Under the simplified conditions of this experiment, conventional wisdom is statistically incorrect. Because this project originally set out to research the effect of reset-graduated and graduated TF methods, we will analyze the statistical differences next, displayed in Figures 7 and 8.

TF Method	ROUGE-1	ROUGE-2
0%	1.43E-08	1.75E-07
25%	7.79E-03	5.55E-2
50%	0.933	0.912
75%	3.72E-07	1.24E-05
100%	2.58E-04	5.78E-06
Reset-Grad	0.778	0.343

Table 7: ROUGE paired Wilcoxon tests for various TF methods as compared to graduated

TF Method	ROUGE-1	ROUGE-2
0%	8.38E-08	6.57E-10
25%	1.82E-2	4.23E-3
50%	0.842	0.292
75%	8.78E-08	6.31E-04
100%	8.14E-05	3.74E-04
Graduated	0.778	0.343

Table 8: ROUGE paired Wilcoxon tests for various TF methods as compared to reset-graduated

As can be seen, graduated and reset-graduated TF methods have no statistically significant difference in generative performance. Perhaps unsurprisingly, both graduated

and reset-graduated TF score similarly to 50% constant TF, as these methods will, on average, have contributed TF approximately 50% of the time. This is in contrast to the idea of presumed benefit from gradual tapering of TF, however.

Now that we have analyzed these various averages and their statistical differences, we will move on to loss over time.

4.2 Loss over Time

The original motivation behind TF was to speed up training. Consequently, we will evaluate the speed of convergence for all seven TF methods in all three tests conducted.

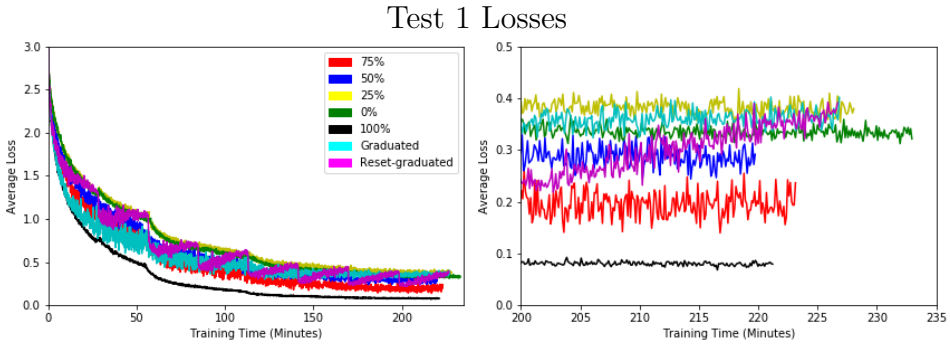


Figure 16: Average loss over time for every TF method for test 1

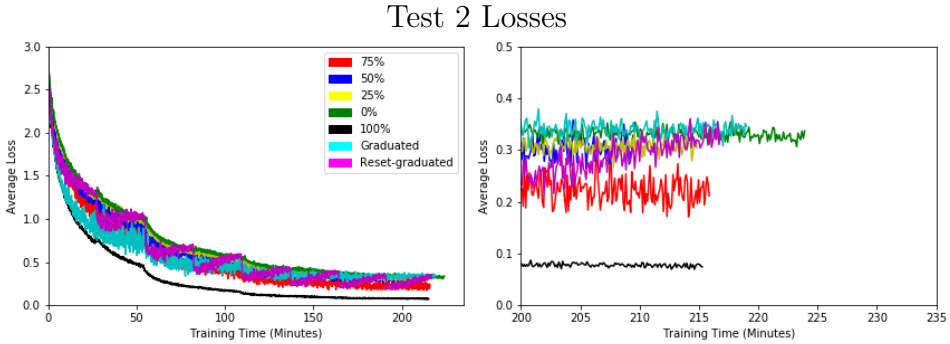


Figure 17: Average loss over time for every TF method for test 2

There is a definite pattern amongst these three plots. 100% TF converges noticably faster than all other TF tests. Additionally, 100% TF had the lowest loss, unsurprising given the ROUGE findings. 75% TF is consistently the second lowest for coverged loss. The rest all fall within the same general vacinity of one another. One interesting observation is how the Kruskal-Wallis scores showed no statistical difference between any of

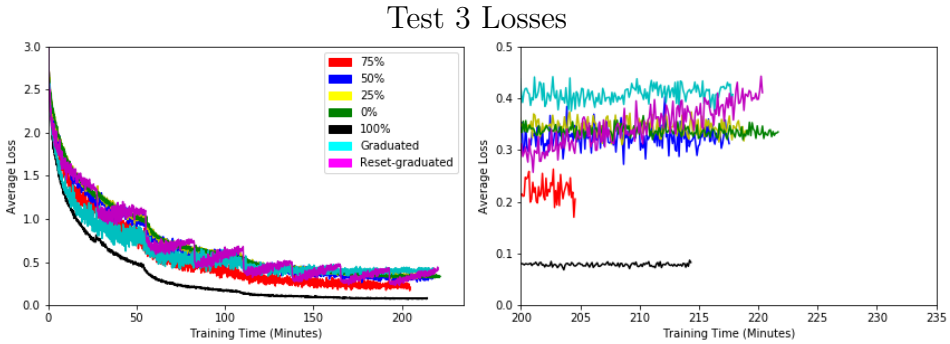


Figure 18: Average loss over time for every TF method for test 3

the ROUGE tests, yet many of the models from these techniques fluctuate by as much as 0.05. This would indicate that ROUGE outcomes are not particularly susceptible to such fluctuations at this loss level.

A second interesting observation is that converged loss proximity does not align with statistically significant over-performance on the ROUGE metric. According to the analysis on test 2 in Section 4.1.1, graduated, reset-graduated, 50% and even 25% all statistically outperformed 0% TF, yet all final loss values are roughly the same. Indeed, pulling the last recorded average loss for the functions results in the following losses:

- Graduated: 0.346
- Reset-graduated: 0.329
- 50%: 0.327
- 25%: 0.305
- 0%: 0.337

This discrepancy between significant out-performance on ROUGE and comparative final loss values raises questions of whether ROUGE is a reliable metric, or whether the traditional cost function for these models needs could be better suited to specifically maximize ROUGE scores. However, given the lack of sophistication involved in ROUGE, I'm willing to bet the former. These observations raise some interesting new questions for future research.

5 Conclusion

In the Information Age, headlines have a ever-increasing presence in our lives. They show up in mobile notifications, in search engines, and all of the world's most popular websites. Due to the influence these hyper-condensed summaries can have, it is important to try to take steps towards ameliorating some of the negative consequences of their ubiquity. That said, the NLP community still has a lot of work to do in order to get to a point where they can start to address some of these pressing issues computationally. In these beginning stages, though, there are some interesting questions to be answered, not least of which is the question of optimal training parameters. As seen through this research, picking the correct TF method can have a statistically significant impact on generative results, as well as training time and converged loss.

The results of this experimentation have shown statistically significant ROUGE result improvement from using 75% – 100% teacher forcing over any other methods tested. With the flexibility that PyTorch provides, there are an infinite number of TF strategies, but it's only a matter of time until we find the best way to automate and optimize it. For now, it is still a manually set variable, and typically only at 100% out of necessity due to the restrictive nature of today's most popular deep learning libraries. For the abstractive summarization researcher who measures their success via ROUGE, however, this does not appear to be a problem. As is the case with many research projects, more questions were likely created than solved, but this opens the door to future research opportunities.

Sequence-to-sequence modelling is becoming more and more sophisticated as techniques and computability improve. Still, even now, there are many open questions that need addressing and I hope that the findings of this research can be utilized within the community.

6 Considerations for Future Work

I believe the groundwork laid by this research leads to some interesting new questions, all of which are cause for further investigation. Some future projects may include:

- Running the models with greater computational resources to processes full dataset, add more encoding and decoding layers, increase hidden size, etc. See how model sophistication impacts the results found here.
- Try a different domain. Seq2seq models are used very frequently in Neural Machine Translation (NMT) as well. Running the various TF experiments on a new domain could uncover some interesting results.
- Invent new TF techniques, such as graduated steps every n epochs.
- Try running lower TF percentages only when a model has converged as a previous arbitrary TF percentage.
- Implement the TF rate into PyTorch's autograd to learn, at any given iteration, whether or not TF should be allowed. Essentially turn teacher forcing into one of the tunable parameters that the model learns in order to best assist itself in minizing the model's loss.
- Development of a more appropriate metric to judge computational comprehension than ROUGE

References

[1] National Institute of Aging and N. i. o. H. National Institute on Aging, "Alzheimer's disease: Unraveling the Mystery," tech. rep., National Institute of Aging, 2011.

[2] Stanford, "CS231N 2.1 Convolutional Neural Networks: Architectures, Convolution / Pooling Layers," 2016.

[3] C. Olah, "Understanding LSTM Networks," 2015.

[4] M. Gabielkov, A. Ramachandran, A. Chaintreau, M. Gabielkov, A. Ramachandran, A. Chaintreau, A. Legout, S. Clicks, M. Gabielkov, and A. Chaintreau, "Social Clicks : What and Who Gets Read on Twitter ? To cite this version : Social Clicks : What and Who Gets Read on Twitter ?," *SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 1, pp. 179–192, 2016.

[5] R. Wenzlaff, R. M. Kerker, and A. E. Beattie, "Incrimination through innuendo: Can media questions become public answers?," *Journal of Personality and Social Psychology*, vol. 40, no. 5, pp. 822–832, 1981.

[6] U. K. H. Ecker, S. Lewandowsky, E. P. Chang, and R. Pillai, "The effects of subtle misinformation in news headlines.," *Journal of experimental psychology. Applied*, vol. 20, no. 4, pp. 323–35, 2014.

[7] A. M. Rush, S. Chopra, and J. Weston, "A Neural Attention Model for Abstractive Sentence Summarization," *In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, no. September, pp. 379–389, 2015.

[8] S. Chopra, M. Auli, and A. M. Rush, "Abstractive Sentence Summarization with Attentive Recurrent Neural Networks," in *NAACL-2016*, pp. 93–98, 2016.

[9] P. Liu and X. Pan, "Text summarization with TensorFlow," 2016.

[10] R. Nallapati, B. Zhou, C. N. dos Santos, C. Gulcehre, and B. Xiang, "Abstractive Text Summarization Using Sequence-to-Sequence RNNs and Beyond," *Proceedings of CoNLL*, pp. 280–290, feb 2016.

[11] H. P. Luhn, "The Automatic Creation of Literature Abstracts," *IBM Journal of Research and Development*, vol. 2, no. 2, pp. 159–165, 1958.

[12] D. Radev, E. Hovy, and K. McKeown, "Introduction to the special issue on summarization," *Computational linguistics*, vol. 28, no. 4, pp. 399–408, 2002.

[13] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," 1989.

[14] T. Mikolov, M. Karafiát, L. Burget, and J. Cernocký, "Recurrent neural network based language model.," *Interspeech*, 2010.

[15] T. Mikolov and S. Kombrink, "Extensions of recurrent neural network language model," *Icassp*, pp. 5528–5531, 2011.

[16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[17] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[18] A. Goodfellow, Ian, Bengio, Yoshua, Courville, "Deep Learning," 2016.

[19] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," *Proceedings of the 27th International Conference on Machine Learning*, no. 3, pp. 807–814, 2010.

[20] M. A. Nielsen, "Neural networks and deep learning," *URL: http://neuralnetworksanddeeplearning.com/.(visited: 01.11.2016)*, 2017.

[21] J. L. Elman, "Finding Structure in Time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[22] A. Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks," *Web Page*, pp. 1–28, 2015.

[23] J. Hochreiter, *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, Technische Universitat, Munchen, 1991.

[24] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long Term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[25] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–80, 1997.

[26] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014.

[27] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An Empirical Exploration of Recurrent Network Architectures," *Proceedings of The 32nd International Conference on Machine Learning*, vol. 37, pp. 2342–2350, 2015.

[28] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," 2016.

[29] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of-Words and Phrases and their Compositionality," *CrossRef Listing of Deleted DOIs*, vol. 1, pp. 1–9, 2000.

[31] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A Neural Probabilistic Language Model," *The Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

[32] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1532–1543, 2014.

[33] T. Mikolov, G. Corrado, K. Chen, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, pp. 1–12, 2013.

[34] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Nips*, pp. 1–9, 2014.

[35] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation By Jointly Learning To Align and Translate," *Iclr 2015*, pp. 1–15, 2014.

[36] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *ArXiv e-prints*, pp. 1–23, 2016.

[37] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-End Attention-based Large Vocabulary Speech Recognition," in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pp. 4945–4949, 2016.

[38] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko, "Sequence to sequence - Video to text," in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 11-18-Dece, pp. 4534–4542, 2016.

[39] K. Lopyrev, "Generating News Headlines with Recurrent Neural Networks," *arXiv*, pp. 1–9, 2015.

[40] S. Robertson, "Translation with a Sequence to Sequence Network and Attention," 2017.

[41] B. Hu, Q. Chen, and F. Zhu, "LCSTS: A Large Scale Chinese Short Text Summarization Dataset," *In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, no. September, pp. 1967–1972, 2015.

[42] M.-T. Luong, H. Pham, and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation," *Emnlp*, no. September, p. 11, 2015.

[43] R. J. Williams and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

[44] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks," *NIPS*, pp. 1–9, 2015.

[45] J. Howard, "Introducing Pytorch for fast.ai," 2017.

[46] S. Chintala, "Autograd: automatic differentiation," 2017.

[47] A. See, P. J. Liu, and C. D. Manning, "Get To The Point: Summarization with Pointer-Generator Networks," *ACL 2017*, 2017.

[48] C. Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Proceedings of the workshop on text summarization branches out (WAS 2004)*, no. 1, pp. 25–26, 2004.

[49] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[50] A. C. Davidson and D. Kuonen, "An introduction to the bootstrap with application in R," *Statistical Computing & Statistical Graphics Newsletter*, vol. 13, no. 1, pp. 6–11, 2003.

[51] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, p. 80, 1945.

[52] C. Ford, "Understanding Q-Q Plots," 2015.

Appendix A Data Extraction

```
import os, re, random, glob, pickle, collections, bcolz, numpy as np, ker
from tqdm import tqdm
from random import shuffle
from lxml import etree
from itertools import islice

MAX_HDLN_LEN = 30
NUM_ART_SENTS = 1
MAX_ART_LEN = 50 * NUM_ART_SENTS

pattern = re.compile(r'''
                    # Don't care about matching beginning of string
(                   # Start of Conditional (...|...)
                    ## First conditional
(\(\s)*             # It could start with 1 or 0 '(' followed by possible
\(+                 # One or More opening parentheses
\-?                 # Possible starting '-' (-[LR]RB-)
[A-Z\.,\'\`:]+      # One or More Capital letters and various punctuation
\-?                 # Possible ending '-' (-(Right|Left) Parenthesis-)
\$?                 # Zero or 1 '$'
\s                  # Must be followed by whitespace

|                   ## Second Conditional
\(\s\$\s            # ( $
|                   ## Third Conditional
\)                  # Also replace ')' with ''
|                   ## Fourth Conditional
```

```
\n
|                    ## Fifth Conditional
by\b\w+\b\w+$)       # by Author
''', re.VERBOSE)

def process_data(sentence):
  sentence = pattern.sub('', sentence)
  sentence = re.sub(r'(\`\`)|(\'\')', '"', sentence)
  sentence = re.sub(r'-RRB-', ')', sentence)
  sentence = re.sub(r'-LRB-', '(', sentence)
  sentence = re.sub(r'\.\.\.', ' ellipsis ', sentence)
  sentence = re.sub("([\d\"().,;:/_?!   ])", r' \1 ', sentence).replace(
  sentence = re.sub(r'ellipsis', '...', sentence)
  sentence = re.sub(r'  *', ' ', sentence)
  sentence = sentence.lower().split()
  return sentence

def extract_headline(hdln):
  for headline in hdln.itertext():
      headline = process_data(headline)
      return headline

def extract_art_txt(txt, n_sents):
  sentences = []
  for sentence in islice(txt.itertext(), n_sents * 2):
      if sentence is not '\n':
          sentence = process_data(sentence)
          sentences += sentence
  return sentences
```

```
def write_tab_seperated(file, headlines, articles, articles_per_file):
  global included, total, overall_total
  tree = etree.parse(file)
  root = tree.getroot()

  tot_arts = len(root)
  overall_total += tot_arts
  print(tot_arts," articles in this file. Extracting 100 article / headli
  rand_indicies = list(range(0,len(root))); shuffle(rand_indicies)

  num_processed = 0
  for rand_idx in rand_indicies:
      child = root[rand_idx]
      hdln = child.find('HEADLINE')
      txt = child.find('TEXT')

      if hdln is not None and txt is not None:
          total += 1
          headline = extract_headline(hdln)
          article = extract_art_txt(txt, n_sents = NUM_ART_SENTS)
          if len(headline) > MAX_HDLN_LEN or len(article) > MAX_ART_LEN:

          headline = ' '.join(headline)
          article = ' '.join(article)
          del child; gc.collect()
          num_processed += 1; included += 1
          if num_processed == articles_per_file: break
          headlines.append(headline)
```

```
            articles.append(article)
    del tree; gc.collect()

DATA_PATH = '/Volumes/Alex Hard Drive/anno_eng_gigaword_5/data/xml/*'
all_files = glob.glob(DATA_PATH); shuffle(all_files)

articles = []; headlines = []
included = 0
total = 0
overall_total = 0

with open('test.txt', 'w') as f:
    f.write('')

for file in tqdm(all_files):
    print('Parsing file: %s' % file)
    write_tab_seperated(file, headlines, articles, articles_per_file=120)
    percent_procd = included/total
    print('Total articles processed: {0} of {2} ({1:.1f}%)'.format(included

with open('articles.pkl', 'wb') as f:
    pickle.dump(articles, f)

with open('headlines.pkl', 'wb') as f:
    pickle.dump(headlines, f)
```

Appendix B Training Functions

```
def train(inp, targ, encoder, decoder, enc_opt, dec_opt, crit, teacher_forc
    decoder_input, encoder_outputs, hidden = encode(inp, encoder)
    target_length = targ.size()[1]

    enc_opt.zero_grad(); dec_opt.zero_grad()
    loss = 0

    if random.random() < teacher_forcing_ratio:
        for di in range(target_length):
            decoder_output, hidden = decoder(decoder_input, hidden, encode
            loss += crit(decoder_output, targ[:, di])
            decoder_input = targ[:, di]

    else: # feed output for next input
        for di in range(target_length):
            decoder_output, hidden = decoder(decoder_input, hidden, encode
            loss += crit(decoder_output, targ[:, di])
            topv, topi = decoder_output.data.topk(1);
            decoder_input = Variable(topi.squeeze()).cuda()

    loss.backward()
    enc_opt.step(); dec_opt.step()
    return loss.data[0] / target_length

  def trainEpochs(encoder, decoder, n_epochs, start_time, times_list, avg_
                  print_every=200, lr=0.01, plot_loss_every=20, teacher_forci
    print('LEARNING RATE: %f' % (lr))
```

```
print_loss = 0 # Reset every print_every
plot_loss = 0

enc_opt = optim.Adam(req_grad_params(encoder), lr=lr)
dec_opt = optim.Adam(decoder.parameters(), lr=lr)
crit = nn.NLLLoss().cuda()

for epoch in range(n_epochs):
    fra, eng = get_batch(fr_train, en_train, 128)
    inp = long_t(fra)
    targ = long_t(eng)

    try:
        isinstance(teacher_forcing, (str, float, int))
    except:
        raise TypeError

    if teacher_forcing == 'graduated':
        teacher_forcing_ratio = 1 - epoch/n_epochs
    elif teacher_forcing == 'full':
        teacher_forcing_ratio = 1
    elif teacher_forcing == 'none':
        teacher_forcing_ratio = 0
    elif teacher_forcing <= 1 and teacher_forcing >= 0:
        teacher_forcing_ratio = teacher_forcing
    else:
        raise ValueError

    loss = train(inp, targ, encoder, decoder, enc_opt, dec_opt, crit, t
```

```
        print_loss += loss
        plot_loss += loss

        if epoch % print_every == 0 and epoch is not 0:
            print('%s\t%d\t%d%%\t%.4f' % (time_since(start_time, epoch / n
                                          epoch, epoch / n_epochs * 100, p
            print_loss = 0

        if epoch % plot_loss_every == 0 and epoch is not 0:
            times_list.append(calc_minutes(start_time))
            avg_loss_list.append(plot_loss / plot_loss_every)
            epochs_list.append(epoch)
            plot_loss = 0

def multi_train(encoder, decoder, times_list, avg_loss_list, epochs_list,
    start_time = time.time()
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li
    trainEpochs(encoder, decoder, 5000, start_time, times_list, avg_loss_li

def fullgraduated_trainEpochs(encoder, decoder, total_epochs, start_time,
                              print_every=200, lr=0.003, plot_loss_every=2
    print("LEARNING RATE: %f" % (lr))
    print_loss = 0 # Reset every print_every
```

```
plot_loss = 0

enc_opt = optim.Adam(req_grad_params(encoder), lr=lr)
dec_opt = optim.Adam(decoder.parameters(), lr=lr)
crit = nn.NLLLoss().cuda()

for epoch in range(total_epochs):
    fra, eng = get_batch(fr_train, en_train, 128)
    inp = long_t(fra)
    targ = long_t(eng)

    teacher_forcing_ratio = 1 - epoch/total_epochs

    loss = train(inp, targ, encoder, decoder, enc_opt, dec_opt, crit, t
    print_loss += loss
    plot_loss += loss

    if epoch % print_every == 0 and epoch is not 0:
        print('%s\t%d\t%d%%\t%.4f' % (time_since(start_time, epoch / to
                                      epoch, epoch / total_epochs * 100
        print_loss = 0

    if epoch % plot_loss_every == 0 and epoch is not 0:
        times_list.append(calc_minutes(start_time))
        avg_loss_list.append(plot_loss / plot_loss_every)
        epochs_list.append(epoch)
        plot_loss = 0

    if epoch == 10000:
```

```
        lr = .001
        print("LEARNING RATE: %f" % (lr))
        enc_opt = optim.Adam(req_grad_params(encoder), lr=lr)
        dec_opt = optim.Adam(decoder.parameters(), lr=lr)
    elif epoch == 20000:
        lr = .0003
        print("LEARNING RATE: %f" % (lr))
        enc_opt = optim.Adam(req_grad_params(encoder), lr=lr)
        dec_opt = optim.Adam(decoder.parameters(), lr=lr)
    elif epoch == 30000:
        lr = .0001
        print("LEARNING RATE: %f" % (lr))
        enc_opt = optim.Adam(req_grad_params(encoder), lr=lr)
        dec_opt = optim.Adam(decoder.parameters(), lr=lr)
    elif epoch == 35000:
        lr = .00003
        print("LEARNING RATE: %f" % (lr))
        enc_opt = optim.Adam(req_grad_params(encoder), lr=lr)
        dec_opt = optim.Adam(decoder.parameters(), lr=lr)
```

Appendix C Testing

```
def evaluate(inp):
    decoder_input, encoder_outputs, hidden = encode(inp, encoder)
    target_length = maxlen

    decoded_words = []
    for di in range(target_length):
        decoder_output, hidden = decoder(decoder_input, hidden, encoder_out
        topv, topi = decoder_output.data.topk(1)
        ni = topi[0][0];
        if ni==PAD: break
        decoded_words.append(en_vocab[ni])
        decoder_input = long_t([ni])

    return decoded_words

def test(f_description):
    parent_dir = './Dissertation/rouge_eval/'+f_description
    ref_dir = parent_dir+'/reference/'
    system_dir = parent_dir+'/system/'

    if not os.path.exists(ref_dir): os.makedirs(os.path.dirname(ref_dir), e
    if not os.path.exists(system_dir): os.makedirs(os.path.dirname(system_

    for idx in range(len(fr_test)):
        real_sent = [en_id2w[t] for t in en_test[idx] if t != 0]
        if real_sent:
            with open(ref_dir + 'news%d_reference%d' % (idx, idx), 'w') as
```

```
            f.write(' '.join(real_sent))
    else:
        continue


ids = long_t(fr_test[idx]); ids = ids.unsqueeze(0)
translation = evaluate(ids)
with open(system_dir + 'news%d_system%d' % (idx, idx), 'w') as f:
    f.write(' '.join(translation))
```