Deep Semantic Hashing

Matthew Findlay
Isaac Jorgensen
Esai Morales
Kevin Velcich

June 12, 2018

Acknowledgements

Thank you to Dr. Wang for teaching COEN 281: Pattern Recognition and Data Mining, teaching us crucial information in the field. Additionally, thank you for the help and guidance for this project. Additionally, we would like to thank the researchers who have provided valuable and important insights for our solution to the semantic hashing problem.

Contents

List of Tables and Figures

Introduction

It is our goal to develop a method which utilizes a Recurrent Neural Network with long short-term memory for hashing documents while maintaining the integrity of the structure of the text and also preserving its semantic meaning.

Similarity search is an increasingly popular problem. Its objective is to identify similar documents given an input document or query, to be used in applications such as search engines, document detection, collaborative filtering, caching, etc. The first proposed methods for similarity search provided accurate and reliable results. However, as a result of the explosive growth of data on the Internet, traditional similarity methods have resulted less impactful. Many of these original methods require computations in high dimensional matrix spaces that become impossible to compute given the large amount of information to be searched. To address this issue, hashing was introduced. Hashing documents provides a method of reducing a document's dimensionality allowing similarity computations to be performed much more quickly and efficiently in a lower dimensionality space. However, an optimal method of determining the best hashing function continues to be heavily researched and improved upon. The best hashing functions aim to not only maintain the similarity, but identify and preserve the semantics of a document as well.

In COEN 281: Pattern Recognition and Data Mining, we covered locality-sensitive hashing (LSH), a method used in similarity search which addresses the shortcomings of traditional similarity search methods. LSH reduces the dimensionality of its input with the objective of preserving the similarity between a set of items. This allows for efficient computing of similar items. This project relates to the course as it analyzes the drawbacks to LSH and proposes a method of hashing which in addition to maintaining similarity, captures the semantic meaning of the items.

There are many methods that exist for hashing documents based off of the structure of the text in those documents. The general use of this is for finding similar documents. However, simply hashing documents based on structure is not enough. For example, consider the following two short texts: “I bought an Apple mouse” and “I bought an apple and a mouse”. These two strings are extremely similar in structure. With most traditional hashing methods, these sentences would hash to the same or nearly the same location. However, in actuality, these sentences have two completely different meanings. While these are short texts, the same principle applies to full-fledged documents, just on a larger scale. It is for the reason provided in the example that current approaches are not ideal. In essence, they ignore the core component behind all text, its meaning. By taking semantics into account, our method of hashing documents will be able to group similar texts more accurately. There has been a limited amount of research into doing this specifically. Additionally, while semantic hashing was originally introduced in 2009, there has not been enough research which

combines the promising and modern technology of deep learning with semantic hashing. To further improve upon what work has been done for semantic hashing, we intend to explore the potential use of Recurrent Neural Networks to aid in the hashing of especially long documents. Recurrent Neural Networks are designed to perform with sequences of information and are frequently used for language applications such as text translation. For this reason, we believe utilizing a recurrent neural network with long short-term memory is a perfect candidate for combining semantic hashing with deep learning and can provide a hashing mechanism which can capture semantic meaning with high accuracy.

Current methods of document/text similarity comparison through hashing are using the wrong metrics for similarity. We will correct that by combining the disciplines of semantic hashing and neural networks to successfully hash and compare documents based on their semantic meaning.

Theoretical Bases and Literature Review

The objective of our project is to create a model which produces hash codes for documents which maintain similarity between hash codes of documents both in structure and in meaning. In order to evaluate this, we use the following formula to calculate the performance of our results:

$$Performance = \frac{\sum_{i=1}^{100} R_i}{100}$$

Where R_i is whether document i is relevant. Precision measures how many selected items are relevant. To evaluate the performance of our hash codes, for each test document in our test data, for the top one hundred hash codes with the closests hamming distance, how many are actually similar/relevant documents. For each method we are comparing with, the total performance or average precision reveals how well the hashing function preserves the similarity and semantics of a document when hashing.

So much information in the world has been documented and archived. With the advent of the internet, the amount of such information has exploded. Given so much work already done, it simply does not make sense to reinvent the wheel each time we need to understand or create something. For this reason, having the ability to search currently existing databases of information in an efficient and effective manner is vital. To achieve this, we must be able to accurately compare text and documents.

From this problem has arisen the solution of document comparison for similarity through the use of hashing. The usefulness of semantic hashing, as it's called, rests on the fact that using a methods like TF-IDF and standard text hashing merely account for the structure of a document through word ordering or frequency while leaving out meaning, the most important component of any text. By accounting for the semantic meanings of words and phrases, we will be able to match documents that are similar in content, not just structure.

Our solution will be implementing semantic hashing, similarly to how the articles we researched did. The key difference will be that we intend to use Recurrent Neural Networks (RNN). The major benefit of the RNN model that we will implement is that it works well on long documents, meaning that it is able to interpret the definition of a word or phrase even if the appropriate context was given much earlier in the document. This key difference should make our implementation even more robust than the semantic hashing methods that have come before.

Related Research

Self-Taught Hashing for Fast Similarity Search

As previously stated, the number of applications that semantic hashing has is very large and continues to grow with advancements in the internet. The premise of Semantic Hashing is the creation of very compact binary sequences used to represent a document. Ultimately, similar documents will be mapped to similar codes that are within a very short hamming distance of each other. Once these documents have been mapped, comparisons for finding the most similar documents rely on simple XOR operations among codes that easily fit into main memory. Additionally, the semantic hashing methodology yields a very pragmatic approach for finding new similar documents. The practical tasks of Information Retrieval, more specifically tasks such as plagiarism detection, involve checking the similarity of a single document, or input query, against a very large set of documents. In most cases, finding all similarities among all documents is not merely practical or useful at all. However, finding the most similar documents to the input query is in most cases, the most useful application. Let's say we have a document that can be represented with the very short binary code of '1111'. If this document is the input query, then we can retrieve the most similar documents from a corpus with shortest hamming distance of 1 which would include those mapped to hash codes of '1110' or '0111'.

Ultimately, this article addresses the problem of designing binary codes for unseen documents from the corpus. The approach is to use unsupervised learning to learn the hash codes of a corpus of documents. From this stage, the use of a supervised approach can be applied to the learning of a new hashcode for any new query document. In essence, this allows the hash function to be trained according to the corpus and tested against any input query. However, the research paper simply proposes a framework that can be used for new query documents. Although the approach hopes to accelerate similarity searches, its focus is on speeding up the process for a new document query, unseen by the corpus.

Semantic Hashing using Tags and Topic Modeling

In their paper “Semantic Hashing using Tags and Topic Modeling”, researchers Wang, Zhang, and Si identified two major issues with existing semantic hashing. The first issue identified is that tag information is often not fully utilized or incorrectly used. In most real world applications, documents are frequently associated with tags, which provide context on the meaning and usage of a document. These tags can be extremely useful when trying to capture semantic meaning. Many implementations completely ignored associated document tags or alternatively used them limitedly. Ignoring the descriptive document tags can certainly hinder the performance of a hashing method. Alternatively, for the methods that do utilize the document tags, they often do not account for documents where tags are missing. A real application requires the allowance of tags to be present or missing, as documents frequently are absent of tags.

The second issue identified was that many methods use document similarity in the original keyword feature space when constructing the hashing function. The main issue with this is that they fail to capture semantic meaning. While documents with similar words are likely to be similar, the converse is not necessarily true; documents with differing words may be semantically similar. Many methods would categorize two different documents under the same topic as different if they have a large vocabulary gap.

To address these issues, the researchers proposed a hashing method which takes into account both the documents tags as well as computing document similarity through the topic of a document rather than the words in the document itself. In order to incorporate document tags, they had to account for two main challenges. The first is that they have to find a way to incorporate the document tags into the hashing function. Secondly, document tags may be missing, so they need to account for situations for incomplete or missing tags. Wang, Zhang, and Si solved the first problem by matrix factorization with a latent factor model. For each tag, a latent variable is introduced, indicating the correlation between tags and hashing codes. A tag consistency component is then calculated using the following equation:

$$\sum_{i=1}^{l}\sum_{j=1}^{n} \left\| \boldsymbol{T}_{ij} - u_i^T y_j \right\|^2 + \alpha \sum_{i=1}^{l} \left\| u_i \right\|^2$$

Where T_{ij} is a binary label for the i-th tag and the j-th item, $u_i^T y_j$ is a weighted sum revealing the relationship between the i-th rag and the j-th document, and the second term is introduced to avoid overfitting with a fixed hyperparameter alpha. To address the second issue, a confidence matrix is constructed, where each document has a value for each tag. When that value is large, then we can trust the tag information for that document. In order to avoid calculating document similarity in the original keyword feature space, the proposed method is by running latent dirichlet allocation (LDA) through the documents to discover a set of latent topics. Each document has a distribution describing how each topic relates to that document. Therefore documents with similar semantic meaning will have similar distributions.

The overall algorithm functions by following the steps as listed. First is the running of LDA through the training documents to discover a set of topics and calculate the distributions for how each document relates to those topics. Next, the confidence matrix is constructed describing how much a tag is trusted for each document. Next the optimal hash codes are computed by minimizing the distance between similar documents (ones with similar distributions). Next, the following equation is used to find an optimal hashing function W:

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} \sum_{j=1}^{n} \|y_j - \boldsymbol{W}x_j\|^2 + \lambda\|\boldsymbol{W}\|_F^2$$

$$\Rightarrow \boldsymbol{W}^* = \boldsymbol{Y}^T\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}$$

This function aims to minimize the distance between the optimal hash codes computed and the outputted hash codes from the hash function for each document.

In order to evaluate how well their proposed hashing method functions, they compared their algorithm to five other top-performing hash functions with four different data-sets. In order to measure their performance, the top 100 nearest neighbors for a given document were found, and the precision (relevant/total) was calculated. Their results showed that their proposed method, SHTTM outperformed all previous methods they compared against. Table 1 displays the exact numbers retrieved for each dataset along with the different hashing methods for different bit sizes.

	ReutersV1					*Reuters*				
Methods	8 bits	16 bits	32 bits	64 bits	128 bits	8 bits	16 bits	32 bits	64 bits	128 bits
SHTTM	**0.559**	0.636	**0.712**	**0.781**	**0.815**	0.716	**0.741**	**0.755**	**0.758**	**0.753**
SSH [31]	0.531	**0.641**	0.664	0.722	0.727	**0.718**	0.722	0.724	0.731	0.739
STH [39]	0.507	0.568	0.643	0.646	0.694	0.703	0.715	0.731	0.740	0.734
SH [34]	0.514	0.556	0.617	0.631	0.658	0.641	0.694	0.711	0.703	0.716
PCAH [19]	0.431	0.557	0.601	0.638	0.641	0.652	0.687	0.693	0.646	0.631
LSH [7]	0.289	0.382	0.444	0.557	0.652	0.569	0.592	0.624	0.653	0.678
	20Newsgroups					*WebKB*				
Methods	8 bits	16 bits	32 bits	64 bits	128 bits	8 bits	16 bits	32 bits	64 bits	128 bits
SHTTM	**0.576**	**0.644**	**0.648**	**0.684**	**0.697**	**0.546**	**0.558**	**0.581**	**0.611**	**0.625**
SSH [31]	0.557	0.576	0.624	0.649	0.665	0.514	0.536	0.543	0.562	0.583
STH [39]	0.526	0.585	0.615	0.623	0.651	0.421	0.449	0.495	0.532	0.538
SH [34]	0.510	0.579	0.574	0.607	0.622	0.504	0.513	0.536	0.541	0.547
PCAH [19]	0.471	0.513	0.541	0.568	0.616	0.471	0.524	0.531	0.568	0.570
LSH [7]	0.389	0.431	0.474	0.557	0.646	0.339	0.377	0.389	0.387	0.401

Table 1: Precision of the top 100 retrieved documents on the four datasets with different hashing bits for STTM, compared against other top performing semantic hashing methods.

The proposed method provides very compelling reasons for including tag information when hashing. However, their method of using LDA to discover topics, while intuitive and high performing, has shortcomings. The main issue is that a document is categorized on the basis of its topic(s). However documents can be much more nuanced; topics can have subtopics and LDA may not capture the different subtopics. While it proved to be a reliable method for semantic hashing, we believe that a more accurate method should be able to capture more precise semantics rather than the topics of a document.

Understanding Short Texts through Semantic Enrichment and Hashing

Finding similar documents in large data sets is hard enough, especially when measuring similarity based on semantic meaning instead of text content. This becomes even more difficult when dealing with short texts. Short text does not provide the same contextual clues or themes that a full fledged document might provide. Because of that, the writers of this article decided to implement a special case of semantic hashing specifically for short text.

The proposed solution in this article consists of two main parts: enriching the short text, and hashing the text with a Deep Neural Network (DNN). By enriching the short text, the authors hoped to magnify the semantic meaning of the words in that text. They achieved this by using conceptualization and co-occurring terms.

Conceptualization evaluates the probability of word taking on one of its definitions at that moment. This probability determines how likely a word is to mean one specific definition. This is based off of the few words that surround the target word for context. Because this is not very strong in a short text, conceptualization is used in conjunction with co-occurring terms. A score is assigned to a combination of words based on how often that combination of words occurs together based on their semantic meanings in the current context. Together, these methods magnify the semantic meaning of the short text.

Once these values have been calculated, they are used in a DNN to hash the original text. This DNN takes the text as a vector and passes it through 3 auto-encoders. These auto-encoders are what take into account the previously calculated scores and values to train the DNN. By passing the text through this network, they are able to create a binary hash of the text that has been weighted based off of the calculated values. As a result each binary hash vector should be different yet still true to the original text's meaning.

The authors of this paper did test their method afterwards. They first tested the semantic enrichment portion first. By comparing it against methods that sites like Wikipedia used, they were able to prove that their method is more accurate than what is currently being utilized on the web. In addition, they also tested the DNN and compared

against methods such as TF-IDF. Over all, the new implementation laid out in the paper was more successful and accurate than anything that currently exists.

Deep Semantic Hashing with Generative Adversarial Networks

This paper proposed a method for deep semantic hashing which uses a generative adversarial network for image search. Typically in semantic hashing, supervised models which learn a hash function from labeled images tend to outperform unsupervised models. The paper lists the reasoning for this is because similarity criteria does not always depend on the contents but sometimes can depend on image annotators. However, the main issue with this is that retrieving labeling images is not a simple task and they stress the importance to develop a model that does not require labeled images.

The goal of their paper was to generate synthetic images for learning a hash function while leveraging general adversarial networks. The intuition behind general adversarial networks is to train a conditional generator and discriminator that are able to produce an image given class labels.

The main model produced was deep semantic hashing with general adversarial networks (DSH-GANs). A real input image must have labels that allows the control of the generator to produce similar images. Additionally, non-relevant labels need to be selected so the generator can produce dissimilar images. Finally, the images will go

through a deep convolutional neural network to learn to low level features of the image. The output of the convolutional neural network is fed into three different networks: a hash stream, classification stream, and an adversary stream. The hash stream uses a maximum margin as a rank loss function in order to learn a hashing function. The classification stream forces images with similar labels to have similar hash codes, and conversely, images with separate labels will have dissimilar hash functions. Lastly, the adversary stream is apart of the GAN.

To test their proposed semantic hashing method, they conducted extensive evaluations on two image datasets. They then calculated the accuracy of their DSH-GANs model compared to 11 other models, showing that theirs outperforms.

Method	CIFAR-10 (MAP)				NUS-WIDE (MAP)			
	12-bits	24-bits	32-bits	48-bits	12-bits	24-bits	32-bits	48-bits
DSH-GANs	**0.735**	**0.781**	**0.787**	**0.802**	**0.838**	**0.856**	**0.861**	**0.863**
DSH-GANs$^-$	0.726	0.769	0.772	0.783	0.823	0.847	0.845	0.854
DPSH	0.713	0.727	0.744	0.757	0.794	0.822	0.838	0.851
NINH	0.552	0.566	0.558	0.581	0.674	0.697	0.713	0.715
CNNH	0.439	0.476	0.472	0.489	0.611	0.618	0.625	0.608
KSH+CNN	0.446	0.502	0.518	0.516	0.746	0.774	0.765	0.749
ITQ+CNN	0.212	0.230	0.234	0.240	0.728	0.707	0.689	0.661
SH+CNN	0.158	0.157	0.154	0.151	0.620	0.611	0.620	0.591
LSH+CNN	0.134	0.157	0.173	0.185	0.438	0.586	0.571	0.507
KSH	0.303	0.337	0.346	0.356	0.556	0.572	0.581	0.588
ITQ	0.162	0.169	0.172	0.175	0.452	0.468	0.472	0.477
SH	0.127	0.128	0.126	0.129	0.454	0.406	0.405	0.400
LSH	0.121	0.126	0.120	0.120	0.403	0.421	0.426	0.441

Table 2: Accuracy in terms of MAP. The MAP performance is calculated on the top five thousand returned images.

Table 2 displays the precise results found for each hashing method. As shown DSH-GANs performs very well. Outperforming most methods significantly. The only comparable method was DPSH.

While we found their usage of a general adversarial network had promising results, the basis of their research seemed too restricted. The strongest methods should be able to excel whether information is labeled or unlabeled. However, their approach into incorporating deep learning into the problem of semantic hashing shows that deep learning should not be overlooked and helped guide our approach in this problem.

Our Solution

Our proposed solution to this problem is to utilize long short-term memory in recurrent neural networks in order to create a model to accurately preserve the semantics of a document when generating hash codes. Recurrent neural networks are a form of neural networks where internal states are used. Simply put, recurrent neural networks can use information from previous iterations when computing an output. Recurrent neural networks were designed to perform with sequences of information and are frequently used for language applications such as text translation. However recurrent neural networks alone have a few issues including that maintaining states is computationally expensive as well as gradient issues when training. A popular method to solve these issues in large scale data applications is long short-term memory (LSTM). LSTM is an abstraction that is used alongside recurrent neural networks and is used to maintain, update, and regulate the states of the network models. Therefore, for semantic hashing, we believe using a recurrent neural network with long short-term memory will hold promising results.

Firstly our proposed solution is unique from most semantic hashing methods as it attempts to combine deep learning with semantic hashing. Deep learning is a subfield of machine learning that has recently emerged because of the complex computations that were previously impossible. Deep learning has continuously proved itself to be a viable solution for various problems with extremely strong results. Few methods of semantic

hashing have connected the problem with deep learning, so only limited research has been performed. However, deep learning with semantic hashing is not unexplored. Limited research has proven its strength and viability as a solution for semantic hashing. However, utilizing recurrent neural networks with long short-term memory has been absent in the realm of semantic hashing.

We believe our solution will be superior as the inclusion of recurrent neural networks has strong intuition behind it. As previously mentioned recurrent neural networks were designed specifically for sequential data, and are very powerful when used with text. The inclusion of long short-term memory allows this method to function with large scale data, which semantic hashing requires. Recurrent neural networks with long short-term memory were design to perform for problems very similar to semantic hashing. Our deep learning model is precisely selected and constructed for the exact problem.

Hypothesis

Current semantic hashing methods fail to take word orderings into account when creating hash codes for documents. The bag of words and TF-IDF scorings do an excellent job at representing documents as a distribution, however, documents are not interpreted by humans as distributions of words when being read. Therefore, we propose an RNN based semantic hashing method that will take sentence orderings into account when creating unique hashcodes. We hypothesize that sentence ordering has a significant impact on the semantic meaning of documents, and that our hashing function must take sentence ordering into account to properly capture semantic meaning. We will test an RNN based hashing architecture with the positive hypothesis that it will perform better than non-RNN hashing architectures.

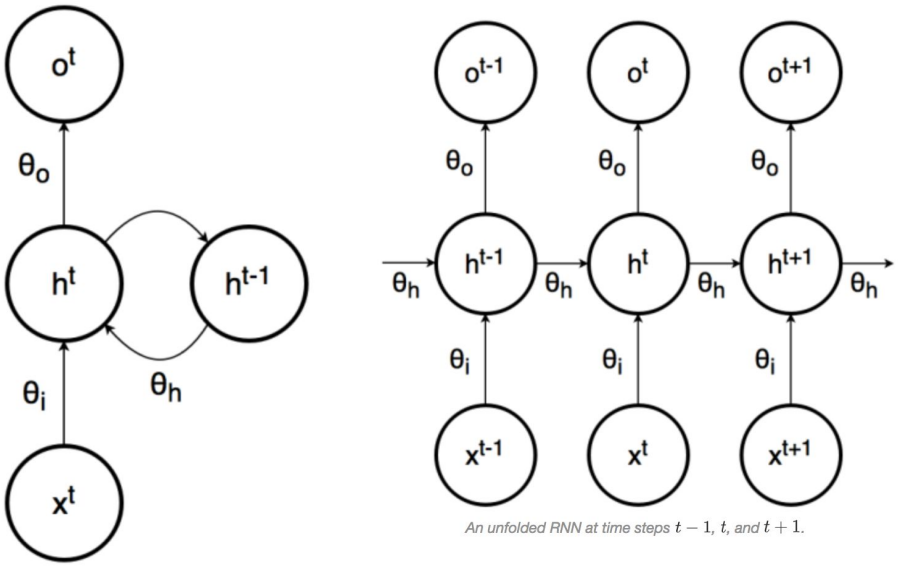
Methodology

The following four public datasets with be used for input data. 1. Reuters Corpus Volume 1 (RCV1). RCV1 is a collection of manually labeled newswire stories provided by reuters. There are a total of 800,000 stories and 103 classes. 2. Reuters21578[2] , this is a collection of 10,788 documents with 90 classes on news topics that have been manually labelled. 3. 20Newsgroups3 dataset which is a collection of 18,828 news articles contained 20 unique classes. 4. TMC4 which is a dataset of air traffic reports provided by NASA. This dataset contains 28,515 documents that are all multi labelled.

Each of the four datasets will be randomly split into 3 subsets: training (80%), testing (10%), and validation (10%). The training set will be used to learn the mapping of document to hash code. The validation set will be used for classifier hyperparameter tuning of our model. The validation set is necessary to avoid hyperparameters that are bias towards our dataset. The testing set will be used to compare our performance against other models. The testing set must be separate as we do not want out mapping to overfit to the data we provide it. Essentially, if we do not have a seperate testing set our model may ‘memorize’ the optimal hash codes for the documents it has seen, meaning it will have poor performance on new documents.

In order to build a model that understands the semantic meaning from sentence orderings we will leverage Recurrent Neural Networks with Long Short Term Memory[1].

Recurrent neural networks allow for neural networks to have a sense of state along with outputs, giving them the ability to work with temporal datasets (**Fig 1.)**

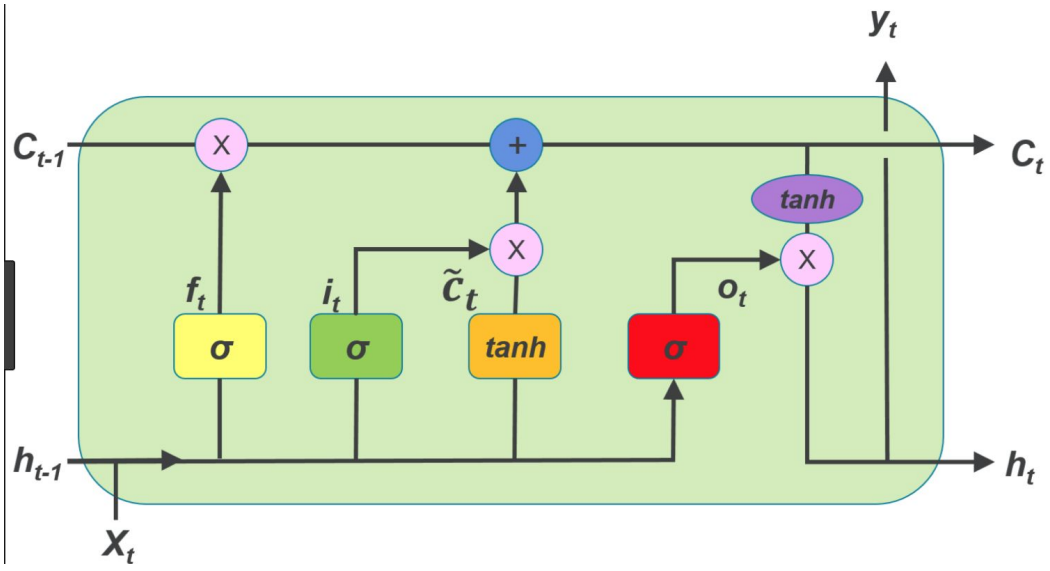


$$o^t = f(h^t; \theta)$$
$$h^t = g(h^{t-1}, x^t; \theta)$$

Figure 1. The simplest RNN architecture. The Output at time t is a function of the current state of the RNN, which is then a function of the previous state and the input into

the RNN. The architecture on the left shows a graphical representation of the RNN, while the architecture on the right is an ‘unrolled’ RNN architecture used for visualization purposes.

The issue with the RNN architecture above is that the information from previous states is lost over several timesteps[2]. Because of the long term memory issues experienced with RNNs, we proposal using an LSTM[3] architecture in our model (**Fig 2.)**. LSTMs are a special kind of RNN capable of learning long term dependencies, they have been widely adopted by the deep learning community because they offer much better performance than typical RNNs. LSTMs work like RNN’s with the repeated unfolded structure, however, they contain 4 neural network layers within them instead of 1.



$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

- σ_g: sigmoid function.
- σ_c: hyperbolic tangent function.
- σ_h: hyperbolic tangent function

Figure 2. The architecture of an LSTM. Several gates are used to decide if states need to be dropped or altered. C_t represents the cell state. F_t is the forget gate, which decides if Cell state needs to be forgotten. I_t is a gate that can add new information to the Cell state. O_t decides on the output of the LSTM.

We plan on inputting sentences one at a time into the LSTM. At the last time step, the output will be our hashed code. This is similar to a ‘many to one’ LSTM architecture (**Figure 3)**. Sentences will be converted to vectors using TF-IDF and bag of words models. Essentially, the LSTM will first hash a sentence, then hash the resulting sentence and hash code repeatedly until the final hashed code is outputted. We think this will produce better semantic meaning than the naive method of feeding and entire document representation into the model at once.

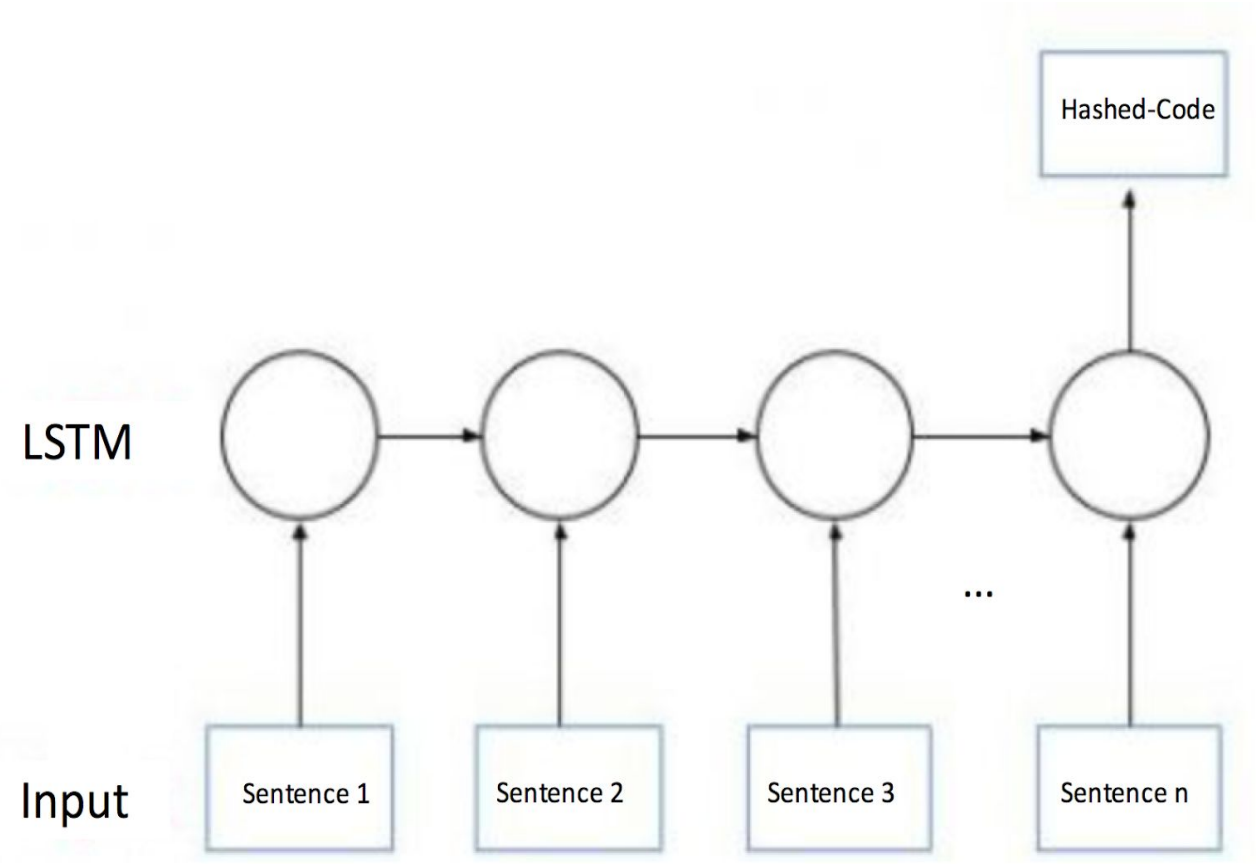


Figure 3. Architecture of our proposal model. Sentences are fed one at a time into a LSTM model trained to generate hashed outputs.

In order to back propagate error through the LSTM, we will reconstruct our sentences from the hashed-code, and use the reconstruction error as a loss function in the RNN.

We plan to implement our model using Python and the Keras deep learning package. Keras is a high level neural-networks API that runs on top of tensorflow. Keras

will ease our development time while still allowing for the dataflow optimizations experienced when using tensorflow.

To test our hypothesis, we will calculate the precision of the top 100 retrieved documents on the 4 datasets we are using. We will compare our performance to the performance of baselines including : Locality Sensitive Hashing (LSH)[4] , Spectral Hashing (SpH)[5], Self-taught Hashing (STH)[6], Stacked Restricted Boltzmann Machines (Stacked RBMs)[7], Supervised Hashing with Kernels (KSH)[8], and Semantic Hashing using Tags and Topic Modeling (SHTTM)[9].

Implementation

In order to implement the design that we came up with, we first needed to select what data sets and tools we were going to use. For our data set, we chose to use the 20 Newsgroups data set. The specific set that we used was pre-labelled by topic. The tools we relied on included Word2Vec, to create vectors of hashes of each document. In addition we used Keras on a Tensorflow backend to build the Recurrent Neural Network (RNN).

Our implementation begins by training an RNN with the 20 Newsgroups data set. Our RNN model is “seq2seq”, a model provided by Tensorflow, which includes recurrent neural networks in it’s list of specialties. The seq2seq model relies on on a DataUtils class which performs all of the data (i.e. 20 Newsgroups) and auxiliary setup necessary to run the model. The model is created and trained on the 20 Newsgroups training set. The data, in this case, are documents labelled by topic. By feeding this into our RNN with their labels as is standard in supervised learning, we teach the model to generate hash codes with the help of Word2Vec. This is performed in the hidden layer of our RNN. Word2Vec is a pre-trained shallow neural network that creates vectors from characters or words. The vectors that are output are intended to be as close in numerical similarity as the original documents are in topic similarity.

Data Analysis and Discussion

To test the accuracy of our model, we measure the precision of both the LSH categorizations and our semantic hashing categorizations. In the case of LSH, we took the cosine distance of all of the vectors, and grouped together the top one hundred. By taking the ratio of the largest similarly labelled group divided by one hundred, we were able get an accuracy score for how well LSH worked. We implemented the same method on our semantic hashing, however, we used hamming distance instead cosine. After running this test multiple times (Table 3), we noticed that our implementation outperformed LSH consistently. The following table shows a sample precision for 10 different document queries.

LSH	Semantic Hashing
0.231021	0.225643
0.051623	0.124423
0.171394	0.210243
0.142467	0.160547
0.231191	0.254238
0.070721	0.129378
0.160164	0.202190
0.150156	0.199121
0.221101	0.176368
0.184203	0.172601

Table 3. Results from 10 tests performed to compare the two methods.

As shown in Table 3, Semantic hashing outperformed LSH marginally. The overall average mean precision for the top 100 documents retrieved for semantic hashing ended up at roughly .185 while LSH received an average mean precision of .165. This is about a 15% improvement from LSH.

In our hypothesis, we predicted that semantic hashing would perform better than non-RNN hashing architectures. Based on our results gathered in Table 3, we found that our implementation of semantic hashing can outperform LSH, a basic version of hashing. However, when comparing LSH to other semantic hashing methods, the top performers typically perform around 100-200% better than LSH. So while our implementation is not a useful alternative to top hashing methods, it was able to outperform several famous hashing methods. Because of this, we believe that our project shows a lot of promise in utilizing RNNs for semantic hashings, and with more time and resources, RNNs could potentially prove to be an optimal solution to semantic hashing. Overall, using word orderings for semantic hashing proved to show working results, however due to limited time, we were unable to optimize the RNN. While we were unable to prove our hypothesis that RNNs would outperform other top semantic hashing methods, our results provide evidence that this may be true.

Conclusion and Recommendations

Summary and Conclusions

Our results, to some degree, demonstrate the effectiveness of the application of Recurrent Neural Networks, specifically, Long short-term memory, on Semantic Hashing. Let's begin by analyzing the results that were gathered. While our results were consistently better than LSH, the ratio of correctly grouped hashes was not very high. It ran between 16-18%. We were successful in developing a better method of finding similar documents, but not to the extent that we initially anticipated.

Recommendations for Future Studies

There exist many possibilities and directions where our studies on deep semantic hashing could be driven towards. Specifically, we would like to emphasize a few future improvements that we hypothesize could vastly improve our system's performance. Possibilities for improvement include, but are not limited to, increasing the training time and size, improving the algorithmic efficiency of our code, and feeding inputs word-by-word or sentence-by-sentence.

The first improvement, deals with increasing the training time for our model. Due to time constraints, deadlines, and work setbacks, it was impossible to train the neural network for an ideal time frame, at least not one which we were satisfied could perform as well

as we had initially predicted. This is one of the reasons why we believe that our precision was low. Furthermore, our current training time was around 8 hours, however, the machine specifications and environment were perhaps not up to par with state-of-the-art, industry standards. Some libraries which we used, like Tensorflow, require or benefit from a high amount of memory or GPUs to handle large datasets adequately. Therefore, given more time and better conditions, we believe our overall precision would benefit.

Another topic of further research and improvement is the algorithmic efficiency of our model. Perhaps our methods are not as optimized heavily and therefore it is necessary to continue working and improving the algorithm and code so that it runs in an acceptable time frame.

A third area for future recommendations and additions is to get the Recurrent Neural Network to predict the original documents from a simple word2vec input. This would essentially reconstruct the document given the word2vec and could have many potential uses in a variety of applications.

Finally, we feel that the results could be greatly improved if the RNN was fed input in a word-by-word or sentence-by-sentence fashion. This would preserve the structural integrity of the text and would inherently aid in maintaining semantics throughout.

Bibliography

1) Has¸im Sak, Andrew Senior, Franc¸oise Beaufays, Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. *INTERSPEECH,* 2014
2) Claudio Gallicchio, Short-Term memory of Deep RNN. *26th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*, Bruges (Belgium), 25-27 April 2018
3) Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, Jürgen Schmidhuber LSTM: A Space Search Odyssey. *IEEE Transactions on Neural Networks and Learning Systems (Volume: 28, Issue: 10, Oct. 2017)* Pages: 2222 - 2232
4) M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *In Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.
5) Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2009.
6) D. Zhang, J. Wang, D. Cai, and J. Lu. Self-taught hashing for fast similarity search. In *SIGIR*, pages 18–25. ACM, 2010.
7) R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

8) W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang. Supervised hashing with kernels. In *CVPR*, pages 2074–2081. IEEE, 2012.

9) Q. Wang, D. Zhang, and L. Si. Semantic hashing using tags and topic modeling. In *SIGIR*, pages 213–222. ACM, 2013.

10) Z. Qiu, T. Yao, Y. Pan, and T. Mei. Deep Semantic Hashing with Generative Adversarial Networks. In *SIGIR,* pages 225-234. ACM, 2017.

11)“20 Newsgroups.” *Home Page for 20 Newsgroups Data Set*, qwone.com/~jason/20Newsgroups/.

Appendix

A. Source Code

a. dataonefile.py

```
import os

path = '/mnt/c/Users/Kevin/Desktop/Deep-Semantic-Hashing/20news-bydate-train'

total_files = 0
file_output = open('20news.data', 'w')

for (dirpath, dirnames, filenames) in os.walk(path):
   total_files = len(filenames)

   for filename in filenames:
       if filename == '.DS_Store':
           continue

       filedir = os.path.join(dirpath, filename)
       category = filedir.split('/')[-2]

       with open(filedir, 'r') as content_file:
           content = content_file.read()
           file_output.write(category + ', ' + repr(content))
           file_output.write('\n')

print 'Converted ' + str(total_files) + ' documents into single file \'20news.data\''
```

b. LSH.py

```
import binascii
import random

num_hashes = 128
num_docs = 377
```

```
shingle_length = 8
data_file = '20news.data'
doc_shingle_sets = {}
doc_id_to_category = {}
next_prime = 379 #
http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php



def get_index(row, col):
 if row == col:
   sys.stderr.write("Incorrect Access")
   sys.exit(1)

 if col < row:
   temp =row
   row = col
   col = temp

 return int(row * (num_docs - (row + 1) / 2.0) + col - row) - 1

def generate_hash(k):
 # Create a list of 'k' random values.
 rand_list = []
  while k > 0:
   # Get a random shingle ID.
   rand_idx = random.randint(0, doc_count)
   # Ensure that each random number is unique.
   while rand_idx in rand_list:
     rand_idx = random.randint(0, doc_count)

   # Add the random number to the list.
   rand_list.append(rand_idx)
   k = k - 1
 return rand_list

data_file = open(data_file, "rU")

docs = []
```

```
doc_count = 0
all_shingles = set()

for i in range(0, num_docs):
 doc = data_file.readline()
 category = doc.split(',')[0]
 doc = doc.split(',')[1:]
 doc = ','.join(doc)
 doc = doc.split('\'')[1:-1]
 doc = '\''.join(doc)
 doc = str(doc)
 docs.append(doc_count)

 shingles_in_doc = set()
 shingles = [doc[i: i + shingle_length] for i in range(len(doc))][: -shingle_length]

 for shingle in shingles:
     crc = binascii.crc32(shingle) & 0xffffffff
     shingles_in_doc.add(crc)
     all_shingles.add(crc)

 doc_shingle_sets[doc_count] = shingles_in_doc
 doc_id_to_category[doc_count] = category
 doc_count += 1
 data_file.close()

a = generate_hash(num_hashes)
b = generate_hash(num_hashes)

sigs_list = []
for doc_id in docs:
 shingle_id_set = doc_shingle_sets[doc_id]
  doc_sig = []
 for i in range(0, num_hashes):
   min_hash_code = next_prime + 1

   for shingle_id in shingle_id_set:
     hash_code = (a[i] * shingle_id + b[i]) % next_prime
```

```
      if hash_code < min_hash_code:
        min_hash_code = hash_code

    doc_sig.append(min_hash_code)
  sigs_list.append(doc_sig)

file_output = open('hash_codes.out', 'w')
for i, hash_code in enumerate(sigs_list):
  if len(hash_code) != num_hashes:
    print 'Bad hash code'
  file_output.write(str(hash_code) + ', ' + str(doc_id_to_category[docs[i]]) + '\n')
```

c. Demo.py

```
from __future__ import print_function

from keras.models import Model
from keras.layers import Input, LSTM, Dense
import numpy as np
import seq2seq
import utils
import sklearn
import numpy as np
import sys
from itertools import chain
import time

def hamming_dist(s1, s2):
    assert len(s1) == len(s2)
    return sum(c1 != c2 for c1, c2 in zip(s1, s2))

def precision(top100, query_class):
    count = 0
    for _, c in top100:
        if c == query_class:
            count+=1
    a_precision = float(count)/100
```

```
    print ("Average precision, ", a_precision)
    return a_precision

if __name__ == "__main__":

    print("\n\nLoading pre-trained model weights...")
    time.sleep(6)
    data_path = '20news-bydate-train'
    utils.DataUtils.get_20news_dataset(data_path)

    batch_size = 64  # Batch size for training.
    epochs = 100  # Number of epochs to train for.
    latent_dim = 256  # Latent dimensionality of the encoding space.
    num_samples = 10000  # Number of samples to train on.
    # Path to the data txt file on disk.

    encoder_input_data, decoder_input_data, decoder_target_data, num_encoder_tokens,
num_decoder_tokens = utils.DataUtils.prep_train_data(0, 1000)
    seq_model = seq2seq.seq2seq(latent_dim, num_encoder_tokens, num_decoder_tokens)
    seq_model.load_existing()

    #enc_test, _, _, _, _= utils.DataUtils.prep_train_data(1000, 1100)
    for seq_index in range(1):
        print("\nSelecting document...")
        time.sleep(5)
        print("\nHashing document\n")
        input_seq = encoder_input_data[seq_index: seq_index + 1]
        states_value = np.array(seq_model.hash(input_seq))
        hash_code = utils.hash(states_value)
        hash_class = utils.DataUtils.Y_train[seq_index].split(".")[0].strip(",")
        h = ''
        for val in hash_code:
            h+=str(val)
        print("Hash value of document: \n", h)
        print ("\nClass of hashed document: ", hash_class)
        print("\n")

    time.sleep(5)
```

```
input_path = "hashed_documents/all_hash_values.txt"
hash_dict = {}
sim_dict = {}
precisions = []
print("Getting hash values of other documents...\n")
time.sleep(5)

with open(input_path, "r") as f:
    classes = {}
    for line in f.readlines():
        hash, c = line.split("]")
        c = c.split(".")[0].strip(",")
        if c in classes:
            classes[c]+=1
        else:
            classes[c] = 1
        hash_string = ''
        for val in hash:
            if val == '1' or val == '0':
                hash_string+=val

        hash_dict[hash_string] = c

query = h
query_class = hash_class
top100 = []
print("\nComparing hash values, finding 100 most similar\n")
for key, key_class in hash_dict.items():
    sim = hamming_dist(query, key)
    top100.append([sim, key_class])
top100 = sorted(top100, key=lambda x: x[0], reverse=True)

print("\nClasses of most relevant documents:\n")
for i, v in enumerate(top100):
    print(str(i) + ". " + "sim: " + str(v[0]) + " class: " + str(v[1]))
    if (i == 100):
        break
precisions.append(precision(top100[:100], query_class))
```

```
   avg_precisions = np.array(precisions)
```

d. Evaluate.py

```
import numpy as np
import sys
from itertools import chain
from sklearn.metrics import average_precision_score

input_path = sys.argv[1]
hash_dict = {}
sim_dict = {}
precisions = []

def hamming_dist(s1, s2):
   assert len(s1) == len(s2)
   return sum(c1 != c2 for c1, c2 in zip(s1, s2))

def precision(top100, query_class, classes):
   count = 0
   for _, c in top100:
       if c == query_class:
           print (c)
           print (query_class)
           count+=1
   a_precision = float(count)/100
   print ("Average precision, ", a_precision)
   return a_precision


with open(input_path, "r") as f:
   classes = {}
   for line in f.readlines():
       hash, c = line.split("]")
       c = c.split(".")[0].strip(",")
       if c in classes:
           classes[c]+=1
       else:
```

```
            classes[c] = 1
        hash_string = ''
        for val in hash:
            if val == '1' or val == '0':
                hash_string+=val

        hash_dict[hash_string] = c

for query, query_class in hash_dict.items():
   top100 = []
   for key, key_class in hash_dict.items():
        sim = hamming_dist(query, key)
        top100.append([sim, key_class])
   top100 = sorted(top100, key=lambda x: x[0], reverse=True)
   if len(top100) > 100:
        top100 = top100[:100]
   precisions.append(precision(top100, query_class, classes))

avg_precisions = np.array(precisions)

print("Mean average precision: ", np.average(avg_precisions))

with open("Mean_average_precision_result.txt", "w") as f:
   f.write("Mean average precision of RNN hasher: ")
   f.write(str(np.average(avg_precisions)))
```

e. Seq2seq.py

```
from keras.models import Model
from keras.layers import Input, LSTM, Dense
import numpy as np
from keras import backend as K
import sys
import utils
import word2vec
import gensim
import word2vec
```

```
def get_session(gpu_num="2", gpu_fraction=0.2):
    import tensorflow as tf
    import os

    os.environ["CUDA_VISIBLE_DEVICES"]=gpu_num

    num_threads = os.environ.get('OMP_NUM_THREADS')
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=gpu_fraction)

    if num_threads:
        return tf.Session(config=tf.ConfigProto(
            gpu_options=gpu_options, intra_op_parallelism_threads=num_threads))
    else:
        return tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))

class seq2seq(object):
    def __init__(self, latent_dim, num_encoder_tokens, num_decoder_tokens):
        self._training_model = None
        self._hashing_model = None
        self.hashing_inputs = None
        self.hashing_states = None
        self.latent_dim = latent_dim
        self.num_encoder_tokens = num_encoder_tokens
        self.num_decoder_tokens = num_decoder_tokens

    @property
    def training_model(self):
        return self._training_model

    @training_model.setter
    def training_model(self, val):
        if self._training_model == None:
            self._training_model = val
        else:
            print("Attempted to train model twice, exiting...")
            sys.exit(0)

    @property
    def hashing_model(self):
```

```
        return self._hashing_model

    @hashing_model.setter
    def hashing_model(self, val):
        if self._hashing_model == None:
            self._hashing_model = val
        else:
            print("Hashing model created twice, exiting...")
            sys.exit(0)

    def _build_training(self):
        #Encoder portion
        encoder_inputs = Input(shape=(None, self.num_encoder_tokens))
        encoder = LSTM(self.latent_dim, return_state=True)
        encoder_outputs, state_h, state_c = encoder(encoder_inputs)
        encoder_states = [state_h, state_c]
        #decoder portion
        decoder_inputs = Input(shape=(None, self.num_decoder_tokens))
        decoder_lstm = LSTM(self.latent_dim, return_sequences=True, return_state=True)
        decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
initial_state=encoder_states)
        decoder_dense = Dense(self.num_decoder_tokens, activation='softmax')
        decoder_outputs = decoder_dense(decoder_outputs)

        self.hashing_inputs = encoder_inputs
        self.hashing_states = encoder_states
        self.training_model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

    def _build_hashing_function(self):
        self.hashing_model = Model(self.hashing_inputs, self.hashing_states)

    def load_existing(self, weight_path="s2s_training_weights.h5"):
        self._build_training()
        self.training_model.load_weights(weight_path)

    def train(self, encoder_input_data, decoder_input_data, decoder_target_data,
              optimizer="rmsprop", loss='categorical_crossentropy', batch_size=100,
epochs=10, validation_split=0.2):
        self._build_training()
```

```
        self.training_model.compile(optimizer=optimizer, loss=loss)
        self.training_model.fit([encoder_input_data, decoder_input_data],
            decoder_target_data, batch_size=batch_size, epochs=epochs,
validation_split=0.2)
        self.training_model.save("s2s_training_weights.h5")

    def hash(self, input_data):
        if self.hashing_model == None:
            self._build_hashing_function()
        states = self.hashing_model.predict(input_data)
        return states



if __name__ == "__main__":

    K.set_session(get_session())
    data_path = '20news-bydate-train'
    utils.DataUtils.get_20news_dataset(data_path)

    batch_size = 64  # Batch size for training.
    epochs = 100  # Number of epochs to train for.
    latent_dim = 256  # Latent dimensionality of the encoding space.
    num_samples = 10000  # Number of samples to train on.
    # Path to the data txt file on disk.

    # Vectorize the data.

    encoder_input_data, decoder_input_data, decoder_target_data, num_encoder_tokens,
num_decoder_tokens = utils.DataUtils.prep_train_data(0, 1000)
    seq_model = seq2seq(latent_dim, num_encoder_tokens, num_decoder_tokens)
    seq_model.train(encoder_input_data, decoder_input_data, decoder_target_data)

    #enc_test, _, _, _, _= utils.DataUtils.prep_train_data(1000, 1100)
    with open("hash_values_random.txt", "w") as f:
        for seq_index in range(1000):
            input_seq = encoder_input_data[seq_index: seq_index + 1]
            states_value = np.array(seq_model.hash(input_seq))
            hash_code = utils.hash(states_value)
```

```
            print("Hash value: \n", hash_code)
            f.write(''.join(str(hash_code)))
            f.write(',')
            f.write(str(utils.DataUtils.Y_train[seq_index]))
            f.write('\n')
```

f. Utils.py

```
import os
import scipy
import tqdm
import numpy as np

def hash(states):
   state_string = states.flatten()
   return [1 if i >0 else 0 for i in state_string]

def map(file_path):
   with open(file_path, "r") as f:
       for line in f.readlines():
           pass


class DataUtils(object):
   #X train contains vector of words representations
   #Y train contains labels
   #Files is the name of the file info originated from
   #Ordered by index
   #Converted to numpy arrays after calling
   X_train = []
   Y_train = []
   files = []
   word_vec_train = []

   @staticmethod
   def get_20news_dataset(file_path):
       for dname in os.listdir(file_path):
           if(dname != ".DS_Store"):
```

```
                dname = os.path.join(file_path, dname)
                for fname in os.listdir(dname):
                    fname = os.path.join(dname, fname)
                    try:
                        with open(fname, "rb") as f:
                            file_array = []
                            DataUtils.files.append(fname)
                            for line in f.readlines():
                                for word in line.split():
                                    file_array.append(word)
                            DataUtils.X_train.append(file_array)
                            DataUtils.Y_train.append(dname.split("/")[1])
                    except FileNotFoundError:
                        print ("Did not find file: ", fname)
        DataUtils.X_train = np.array(DataUtils.X_train)
        DataUtils.Y_train = np.array(DataUtils.Y_train)

        from sklearn.utils import shuffle
        DataUtils.X_train, DataUtils.Y_train = shuffle(DataUtils.X_train,
DataUtils.Y_train, random_state=42)

        DataUtils.files = np.array(DataUtils.files)

    @staticmethod
    def get_vector_embeddings():
        # Load Google's pre-trained Word2Vec model.
        model =
gensim.models.KeyedVectors.load_word2vec_format('./GoogleNews-vectors-negative300.bin'
, binary=True)
        #Create word2vec matrix
        for doc in utils.DataUtils.X_train:
            doc_vec = []
            for word in doc:
                try:
                    try:
                        doc_vec.append(model.get_vector(word.decode('utf-8')))
                    except UnicodeDecodeError:
                        #ignore broken embeddings
                        pass
```

```
            except KeyError:
                print("Word not in vocab, skipping")

    DataUtils.word_vec_train.append(doc_vec)

@staticmethod
def prep_train_data(start_ind=0, end_ind=100):
    input_texts = []
    target_texts = []
    input_characters = set()
    target_characters = set()
    for doc in DataUtils.X_train[start_ind:end_ind]:
        for words in doc:
            try:
                w = words.decode('utf-8')
                input_text = w
                target_text = w
                # We use "tab" as the "start sequence" character
                # for the targets, and "\n" as "end sequence" character.
                target_text = '\t' + target_text + '\n'
                input_texts.append(input_text)
                target_texts.append(target_text)
                for char in input_text:
                    if char not in input_characters:
                        input_characters.add(char)
                for char in target_text:
                    if char not in target_characters:
                        target_characters.add(char)
            except UnicodeDecodeError:
                pass

    input_characters = sorted(list(input_characters))
    target_characters = sorted(list(target_characters))
    num_encoder_tokens = len(input_characters)
    num_decoder_tokens = len(target_characters)
    max_encoder_seq_length = max([len(txt) for txt in input_texts])
    max_decoder_seq_length = max([len(txt) for txt in target_texts])

    input_token_index = dict(
```

```
                            [(char, i) for i, char in enumerate(input_characters)])
    target_token_index = dict(
                             [(char, i) for i, char in enumerate(target_characters)])

    encoder_input_data = np.zeros(
                                 (len(input_texts), max_encoder_seq_length,
num_encoder_tokens),
                                 dtype='float32')
    decoder_input_data = np.zeros(
                                 (len(input_texts), max_decoder_seq_length,
num_decoder_tokens),
                                 dtype='float32')
    decoder_target_data = np.zeros(
                                  (len(input_texts), max_decoder_seq_length,
num_decoder_tokens),
                                  dtype='float32')

    for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
        for t, char in enumerate(input_text):
            encoder_input_data[i, t, input_token_index[char]] = 1.
        for t, char in enumerate(target_text):
            # decoder_target_data is ahead of decoder_input_data by one timestep
            decoder_input_data[i, t, target_token_index[char]] = 1.
            if t > 0:
                # decoder_target_data will be ahead by one timestep
                # and will not include the start character.
                decoder_target_data[i, t - 1, target_token_index[char]] = 1.

    return encoder_input_data, decoder_input_data, decoder_target_data,
num_encoder_tokens, num_decoder_tokens
```

g. wordvec.py

```
import os
import gensim
import word2vec
```

```
#Initializes an iterator class that will iterate through all of the docs in the corpus
and
#create a vector for each doc containing vectors of each line containing a vector of
each word
class MySentences(object):
   def __init__(self, dirname):
       self.dirname = dirname
   def __iter__(self):
       for dname in os.listdir(self.dirname):
           #print(dname)
           if(dname != ".DS_Store"):
               dname = os.path.join(self.dirname, dname)
               for fname in os.listdir(dname):
                   for line in open(os.path.join(dname, fname)):
                       yield line.split()

#creating the vectors from the address provided below, will read a directory of
directories containing the docs
#other directory arrangements will not work in this version
sentences = MySentences('./20news-bydate-train/') # a memory-friendly iterator
list(sentences)
#print(list(sentences))

#passing the vectors into word2vec, dimesion of 256 currently
model = gensim.models.word2vec.Word2Vec(sentences, size=256, sorted_vocab=1)
#model.save('wv.txt')
print(model)
#print(model.wv.vocab)

#each input prints out the vector for the word, just hit "return" to end, or type a
word not in the corpus
word = 'empty_string'
while (word != '\n'):
   word = raw_input("Enter a word: ")
   print(model.wv[word])

#model.build_vocab(sentences)
#print(model)
#print(model.wv.vocab)
```

B. Input/Output

These sets are too large to put into a document such as this. The inputs can be found at the link provided by citation 11. The relevant outputs are shown in Table 3.