Reverse Dictionary

Stanford CS224N Custom Project

Noah Kuo
Stanford University
noahkuo@stanford.edu

H. Billur Engin
Stanford University
bengin@stanford.edu

Ethan Cheng
Stanford University
ethanlc2@stanford.edu

Abstract

In this project, we produce a reverse dictionary, which allows one to find a word that they can't remember, by describing its meaning. We compare multiple architectures to find the most efficient implementation of a reverse dictionary, in terms of accuracy, computational workload and speed. We observe that an encoder-decoder model based on a BERT encoder coupled with either linear decoder layers or an LSTM decoder layer yield the best results.

1 Key Information to include

- Mentors: Gaurav Banerjee, Andrey Kurenkov

2 Introduction

A reverse dictionary takes in a definition as an input query, and returns the top-k candidate words that are most likely to match this definition. The input can be similar to a dictionary definition, or even a colloquial description of the desired word. For example, the input "a small vessel propelled on water" should yield an output of "boat" (Please see Table 1 for more examples).

Reverse dictionaries can be useful in a multitude of scenarios, ranging from tip-of-the-tongue word recall or as a resource for those learning a new language. Depending on the use case, users providing queries might be looking to learn common or rare words, so reverse dictionaries should have large vocabularies that can output both frequent and infrequently used words.

While there are a few reverse dictionaries online[1][2], they often fail to accurately capture the meaning of definitions. The most popular versions simply search through multiple dictionaries and find which one has the highest count of similar words, which can work for simple queries but struggle with anything more complex. For example, searching for "opposite of cold" yields options such as "colder" and "antonym" but not "hot" in the top-10 results.

Reverse dictionaries should not only take into account the meaning of each individual word, but also how the meaning of words change when used in combination. Otherwise, queries such as the one above will return poor results. The size of the problem also poses a challenge, since there are hundreds of thousands of possible words that could be suggested. Many words also have synonyms, and it can be challenging to know which specific one matches to a definition. For example, the input "to come together" could mean "meet", "gather", "assemble", and more. As a result, reverse dictionaries should not just return one possible word, but several possible options.

Sample Input	Sample Output
To come together	meet, gather, assemble, rally, reunite
Small amount	smidgen, scrap, speck, little, bit
Winter sport	skiing, snowboarding, sledding, ice skating, sliding

Table 1: Example of reverse dictionary outputs

Although this could be viewed as a classification task, our approach mainly centered around creating a model to generate word embeddings based on definitions. We fine-tuned BERT[3] to use as an

encoder layer, and created different decoder variants to find out the most effective architecture. Part of our motivation is to also explore the viability of creating and hosting a service online, so we also consider model size and efficiency in our analysis.

For our variants, we experimented with different decoder architectures such as linear layers and LSTM[4], as well as different loss functions and training parameters. Finally, we compared using BERT large models and DistilBERT[5] to analyze trade-offs between speed, computational load and performance.

3 Related Work

Before the wide use of neural networks, a number of research groups[6, 7, 8, 9] attempted to address the reverse dictionary problem via "sentence matching" using different similarity metrics. These attempts mainly returned the words whose dictionary definitions were most similar to the query description. Although successful in certain scenarios, this method was failing for human-written query descriptions as their vocabulary was only dependent on the dictionary definitions.

Most interesting among these older approaches is Thorat et al.'s [8] attempt to construct a reverse dictionary based on a network approach. They create a reverse map, by processing a traditional dictionary and storing connections in a sparse matrix. In their graph, each node corresponds to a word, and each node connects to other words that have common parents. Given an input phrase, they perform a depth one graph traversal search, starting from individual words of the input phrase and return candidate words.

Inspired by these fist attempts of reverse dictionaries, we have used a word-vector similarity based approach for our baseline. Please see the Approach section for more detail.

WantWords[10] is one of the more recent and rare neural network based solutions[11, 12] to building a reverse dictionary. For each input, they calculate a confidence score for every single possible output word in the vocabulary. They utilize BERT encodings, part-of-speech predictions, and morphemes/sememes to calculate the confidence scores. The words with the highest confidence scores are returned as outputs.

Noticing that the performance of WantWords' architecture was not far better than the baseline accuracy scores for BERT, we focused our efforts on optimizing BERT without using the other components (part-of-speech, morphemes, etc.) mentioned in this research paper. Plus, the size of our vocabulary is larger than the English corpus of WantWords. To calculate confidence scores on all unique words would be costly from an efficiency standpoint. We sought to create a lightweight approach.

4 Approach

4.1 Baseline using Word Embeddings

Other related papers reported the results of their own baselines, however we felt it was important to create our own baseline on our own dataset for two main reasons. First, some of these baselines were not well described and it was hard to understand what exactly we would be comparing against. Second, we did not know how the difference in quality and size of datasets would affect baseline performance, since it would be difficult to take our own models which were evaluated on vocabularies of over 400 thousand words, and derive useful comparisons to baselines with unknown vocabularies.

We created our baseline solely using word embeddings. Starting with 100-dimensional vectors (60 for FastText[13]), we simply take the mean of word embeddings of all words in the input query and find the most similar word in the vocabulary by cosine similarity.

We find that the performance of the baseline varied depending on which set of word embeddings we used. First, we tried using GloVe[14] vectors pre-trained on the gigaword dataset[15]. This gave good scores, but we had to evaluate on only a subset of our data because our dataset contained some unknown words. Second, we trained our own vectors using our dataset. Although we did not need to remove any words from the dataset, the performance was lower likely because our corpus may not have been large enough. We attempted to solve this problem by augmenting our corpus with text from Wikipedia, which improved scores somewhat. Finally, we also tried using FastText vectors

trained on our own corpus. See Table 2 for the results of our baseline experiments, compared to random guessing on a 400,000 word dataset. Although the accuracy scores of our FastText baseline

Embeddings	Accuracy (top1 / top10 / top100)
GloVe w/ gigaword	0.51% / 1.99% / 5.02%
GloVe w/ dataset & Wikipedia	0.07% / 0.3% / 1.6%
GloVe w/ dataset	0.02% / 0.15% / 0.76%
FastText w/ dataset	0.44% / 1.51% / 4.29%
Random Guessing	0.00025% / 0.0025% / 0.025%

Table 2: Word embedding baseline accuracies

was reasonable, our qualitative analysis made us believe that our derived Fastext vectors were not reliable. FastText vectors make use of n-grams which enables the unseen words to have embeddings close to similar sounding words. However, with our small dataset, this created a phenomenon where the n-gram similarity dominated, instead of semantic similarity. As a result, the vectors were too close to phonologically similar words instead of semantically similar words (please refer to Table 3). After creating our baseline and analyzing different word embeddings, we decide to proceed with the

Embeddings	Most Similar Words to "Red"
GloVe w/ gigaword	yellow, blue, green, black, white
FastText w/ dataset	pressured, uncovered, weathered, majored, checkered

Table 3: Comparison between GloVe and FastText word embeddings

rest of the project using the gigaword GloVe embeddings since these seem to be the most reliable and accurate. Although FastText provides good baseline scores, we believe that these would not work well with our proposed models since the word vectors do not seem to capture much about the semantic meaning of the words.

4.2 Model Architecture

Given an input query, our model first splits the entire query into a set of tokens using the default BERT tokenizer. For our dataset which is described below, the maximum length of any definition input is 291 tokens. However, we truncate to 200 tokens since most queries are not this long.

For our different variants, we either passed these tokens into BERT large or DistilBERT to encode the tokens. This resulted in an output hidden state of size (200, 768) for each input query, which we passed directly to our decoder layers. Here, we used various approaches to construct the decoder.

In our first model approach, we use a stack of linear, ReLU, and normalization layers to reduce the dimensionality to 100 (refer to Figure 1). The first 2 fully connected linear layers reduce the dimensionality from (200, 768) to (200, 128). Then, we flatten from (200, 128) to 25,600, and add two more linear layers to reduce 25,600 to 100. Using this architecture, we experiment with different loss functions and training parameters.

Our second model approach substitutes the linear layers with an LSTM (refer to Figure 2). The LSTM has 4 layers and a dropout of 0.1.

5 Experiments

5.1 Data

We extracted dictionary-like word (single token): definition (one or more token) pairs from "The Online Plain Text English Dictionary (OPTED)"[16] and WordNET[17]. OPTED has 84K unique word: definition pairs (50K unique words associated with 80K unique definitions), whereas WordNET has 207K unique word: definition pairs (~150K unique words and ~120K unique definitions). OPTED is more likely to provide multiple definitions for a word while WordNET is more likely to provide synonyms linked with a single definition. WordNET definitions' lengths may vary between 1 to 64 words, with an average of 9 words. OPTED definitions' lengths may vary between 1 to 167 words, with an average of 11 words.

We have combined WordNET and OPTED word-definition pairs and used the resulting dataset (~300K unique word: definition pairs that consist of ~180K unique words and ~200K unique definitions) with

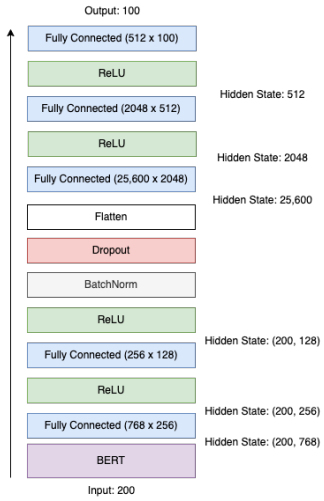


Figure 1: Linear decoder architecture

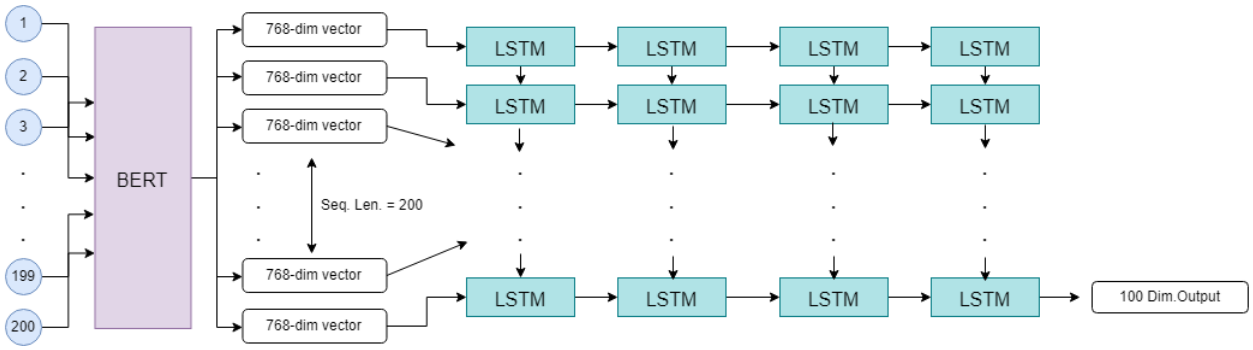


Figure 2: LSTM architecture

an 80/20 train/validation split. We have used the definitions as the input (independent variable) and words as the output (dependent variable). Please see Table 1 for more examples from the dataset.

5.2 Evaluation method

5.2.1 L2 Distance

For each query in our evaluation set, we used our model to output the predicted embedding of the target word. We calculated the L2 distance between the predicted embedding and the embedding of the actual word, and took the average of this distance over the entire evaluation set. We directly optimize for this metric via Mean Squared Error loss, which is the L2 distance squared, multiplied by a constant.

5.2.2 Top-K Accuracy

For each query in our evaluation set, we used our model to output the predicted embedding of the target word. We found the top-1, top-10, and top-100 nearest words by cosine distance and measured the accuracy.

5.3 Experimental details

To set up our experiments, we load the pre-trained BERT model from Hugging Face, and we initialize our own decoders with random values. We also remove word-definition pairs where the target word did not have a corresponding embedding in our pre-trained vectors. We train with a batch size of

32 and utilize an AdamW optimizer for gradient descent. Each epoch took roughly 45 minutes on a TitanRTX GPU. We do not have a pre-set number of epochs, but rather train until the validation accuracy converges.

While running these experiments, we attempted different variations of the loss function and actual training technique. One of the things we tried was a Weighted-MSE loss function, where the MSE loss for each sample was weighted by the **rank** of its similarity to the ground truth. This weight was a parametric with respect to the similarity rank, calculated as max(rank, 100) / 100. For example, if the ground truth word was the top result in the top-100 closest words, then rank=1, so the weight = 0.01. On the other hand, if the ground truth word was the worst result in the top-100 closest words, then rank=100, so the weight = 1. This way, words that were already predicting well were updated less than words that were further down in the top-k similarity. **This experiment failed to improve the results by a significant amount.**

Another thing we tried was to see if we could hold BERT weights constant in certain epochs. We found that holding BERT constant until the end (fine-tuning in the last few epochs) made the network perform significantly worse. However, if the BERT weights were held constant **after the first 5 epochs**, there was no significant reduction in accuracy, and significantly improved training speed.

5.4 Results

We show the results of our experiments in Table 4. "Common words" refers to the top 2000 words from the Wikipedia dataset by frequency. Our scores are higher than we expected compared to

	Training Set			Validation Set		
Model	Avg. MSE Loss	Top 1/10/100	Top 1/10/100 (Common words)	Avg. MSE Loss	Top 1/10/100	Top 1/10/100 (Common words)
Linear	12.48	38/61/77%	50/70/86%	20.15	7.9/20/37%	15/29/47%
LSTM	9.62	69/79/88%	79/92/97%	21.11	7.8/20/35%	14/27/44%

Table 4: Experimental results

the baseline (which we only evaluated on the validation set). This is likely because our model is extracting more semantic meaning about the words in combination than simply taking all words in the definition and adding them together. The LSTM has higher accuracy scores on the training set (Figure 3). Our two models perform similarly on the validation data.

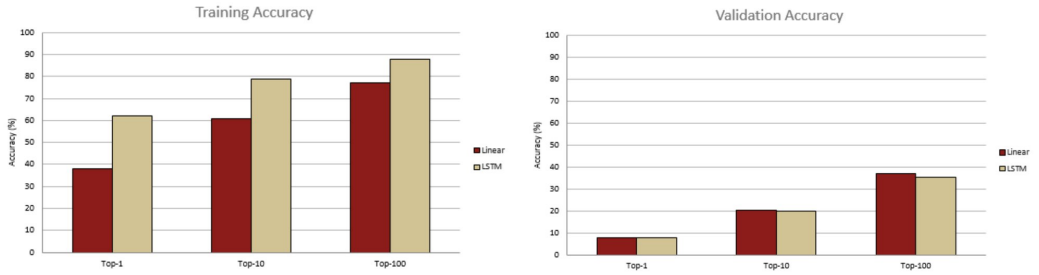


Figure 3: Linear vs LSTM

We also experimented with using DistilBERT as the encoder, rather than BERT. With the same linear-layer decoder architecture, we found that accuracy decreased but not by a significant margin (Figure 4).

6 Analysis

6.1 Result Analysis

While the linear-layer-based decoder and the LSTM decoder had similar results for the validation set, the **LSTM decoder vastly outperforms the linear-layer-based decoder on the training set**.

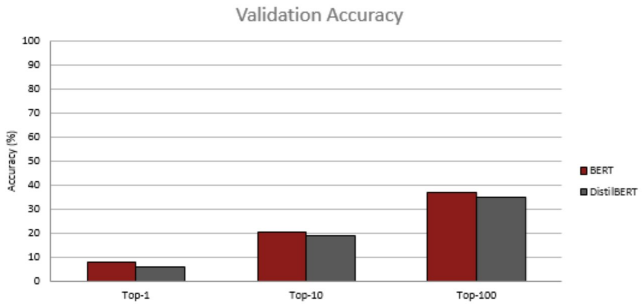


Figure 4: Bert Large vs DistilBert.

While it seems the 2 behave the same on average from these aggregate numbers, through some deeper analysis, it appears that they behave quite differently (Figure 5).

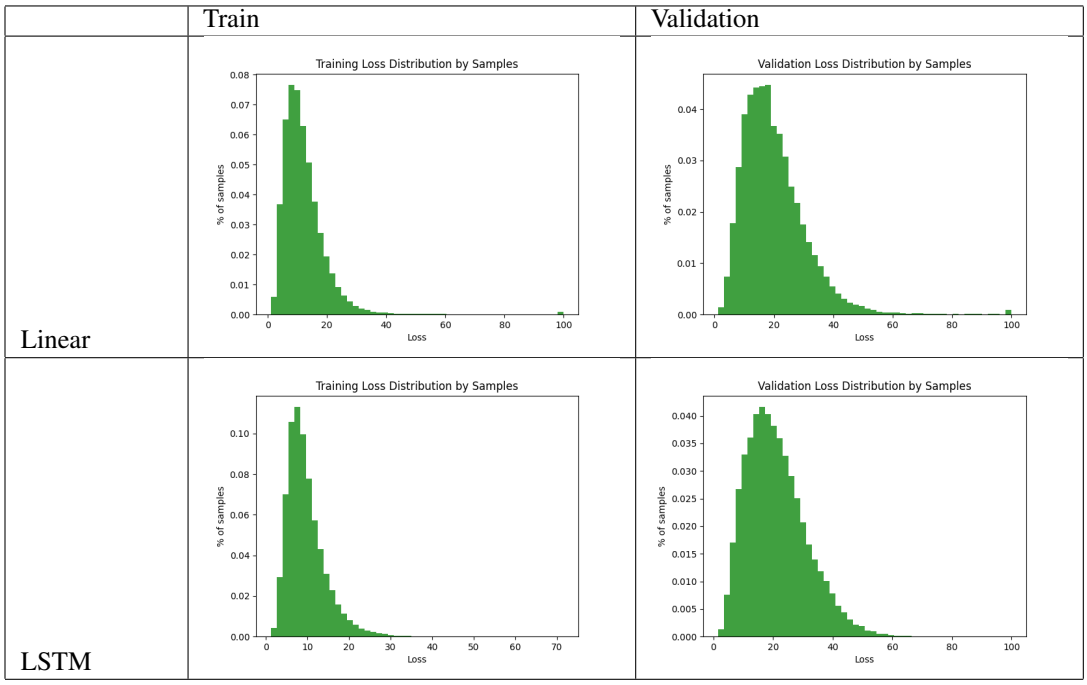


Figure 5: Distribution of loss among samples in training and validation

In these figures, we show the distribution of MSE loss of samples in the training set and validation set. However, the right-most bin (100) actually denotes a loss of 100 or greater. The linear-based decoder has **significantly more outliers** with extremely large loss values (Table 5).

	Training		Validation	
Model	Max Loss	% of samples with loss > 60	Max Loss	% of samples with loss > 60
Linear	1776.4	0.52%	731.50	0.75%
LSTM	71.662	0.0024%	110.25	0.16%

Table 5: Outlier statistics for top 2000 words

Note that loss above 60 is more than 3 SDs above the mean, which is why it is considered an outlier. From these results, the LSTM-based decoder is **more robust** than the linear-based decoder at handling outliers or uncommon tokens. The Top-10 tokens in the training set with the largest MSE loss for the linear-based decoder were all either very uncommon words or had some scientific usage:

‘galliformes’, ‘tortricidae’, ‘baronetage’, ‘phasianidae’, ‘lanceolate’, ‘nitrogen’, ‘acid’, ‘right-handed’, ‘obovate’, ‘c’

While the Top-10 tokens in the training set with largest MSE loss for the LSTM-based decoder had no obvious correlation:

‘herein’, ‘untraded’, ‘lxii’, ‘al’, ‘daybook’, ‘yen’, ‘lah’, ‘republish’, ‘expectable’, ‘de’.

The same analysis above can be done on the top 2000 tokens by frequency in the dataset. The overall loss distribution remains the similar (See Figure 6, Table 7 in the Appendix).

6.2 Training and Runtime Analysis

While training, in order to conserve time to finish before the deadline, we utilized an early stopping mechanism and stopped the training process if there was no improvement in validation set performance in the last 5 epochs while training set performance still increased. With this criteria, the linear-based decoder converged on its validation set performance after 15 epochs (stopped after 20th epoch), while the LSTM-based decoder converged after 100 epochs.

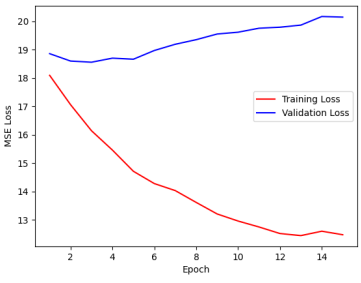
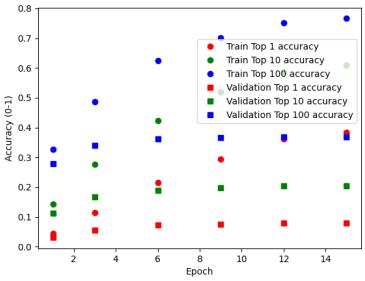
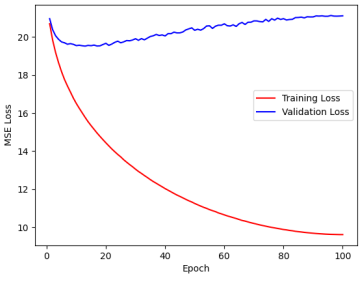
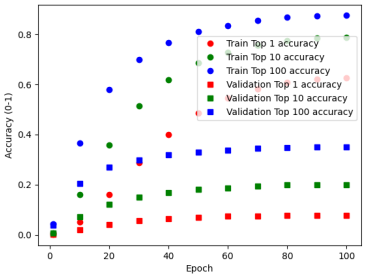
	Loss	Accuracy
Linear		
LSTM		

Table 6: Loss and Accuracies during Training

During training, for both of the models, we noticed a property where the validation top 1/10/100 accuracy would increase, but the validation loss would **also increase** rather than decrease. One explanation for this is our formulation to output a word vector and comparing it to GloVe’s pretrained vectors. Since these are vectors and we are calculating based on a ‘most-similar’ metric, it is possible that the model can output vectors with either continuously growing or shrinking magnitudes, while still maintaining the closest vector as the correct one. We can show this for the linear-based decoder vs the LSTM decoder by outputting the average and largest magnitudes of the output vectors: the linear-based decoder had an average of **15.8** and a max of 2426.01, while the LSTM-based decoder had an average of **12.86** and a maximum of 38.

However, compared to the word vectors in GloVe (using GenSim’s glove-wiki-gigaword-100 vectors), the 2000 most frequent words have an average GloVe vector magnitude of 33.09. However, the more words that are added into this vocabulary, the smaller the average magnitude: the 10,000 most frequent words have an average of 30.03, and the top 100,000 have an average of 23.87.

Since our average vector magnitudes are so much lower in our models, future work would be to see how the model behaves **without weight regularization**, or at least a smaller weight regularization constant, since weight regularization keeps the weight values small, making the actual output values smaller as well.

Compute-wise, the linear model does run faster with a batch size of 32 on a TitanRTX, with an average batch-compute time of **7.7ms** vs the LSTM with an average batch-compute time of **11.8ms**. However, this was a comparison in the PyTorch native runtime. This 53% difference in runtime can possibly be alleviated with custom runtime accelerators like OpenVINO or TensorRT.

7 Conclusion

We believe that given the results, the BERT-encoder + LSTM-decoder is the best model variant from our approaches for the reverse dictionary problem. While it is computationally slower and requires a longer training period to converge, it has much more robust behavior compared to the linear-layer-based decoder. In addition to this, hyperparameters may play a bigger part in model performance than the architecture itself, such as allowing being able to tune the training such that model can output values in the range similar to GloVe vectors.

There is some limitation due to the data that we used to train - we saw that common words with many accurate definitions had strong accuracy, but our model struggled to classify rare, scientific words. Some avenues for future work can include finding more data for either training or validation, or hosting the service online for anyone to use.

References

[1] Doug Beeferman. Onelook reverse dictionary, 2003.

[2] https://reversedictionary.org/.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] Sepp Hochreiter. Ja1 4 rgen schmidhuber (1997)."long short-term memory". *Neural Computation*, 9(8).

[5] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[6] Slaven Bilac, Wataru Watanabe, Taiichi Hashimoto, Takenobu Tokunaga, and Hozumi Tanaka. Dictionary search based on the target word description. In *Proc. of the Tenth Annual Meeting of The Association for Natural Language Processing (NLP2004)*, pages 556–559, 2004.

[7] Oscar Méndez, Hiram Calvo, and Marco A Moreno-Armendáriz. A reverse dictionary based on semantic analysis using wordnet. In *Mexican International Conference on Artificial Intelligence*, pages 275–285. Springer, 2013.

[8] Sushrut Thorat and Varad Choudhari. Implementing a reverse dictionary, based on word definitions, using a node-graph architecture. *arXiv preprint arXiv:1606.00025*, 2016.

[9] Ryan Shaw, Anindya Datta, Debra VanderMeer, and Kaushik Dutta. Building a scalable database-driven reverse dictionary. *IEEE Transactions on Knowledge and Data Engineering*, 25(3):528–540, 2011.

[10] Fanchao Qi, Lei Zhang, Yanhui Yang, Zhiyuan Liu, and Maosong Sun. Wantwords: an open-source online reverse dictionary system. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–181, 2020.

[11] Dimitri Kartsaklis, Mohammad Taher Pilehvar, and Nigel Collier. Mapping text to knowledge graph entities using multi-sense lstms. *arXiv preprint arXiv:1808.07724*, 2018.

[12] Michael A Hedderich, Andrew Yates, Dietrich Klakow, and Gerard De Melo. Using multi-sense vector embeddings for reverse dictionaries. *arXiv preprint arXiv:1904.01451*, 2019.

[13] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[15] Courtney Napoles, Matthew R Gormley, and Benjamin Van Durme. Annotated gigaword. In *Proceedings of the Joint Workshop on Automatic Knowledge Base Construction and Web-scale Knowledge Extraction (AKBC-WEKEX)*, pages 95–100, 2012.

[16] R Sutherland. The online plain text english dictionary. *http://www. mso. anu. edu. au/˜ ralph/OPTED*.

[17] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

A Appendix

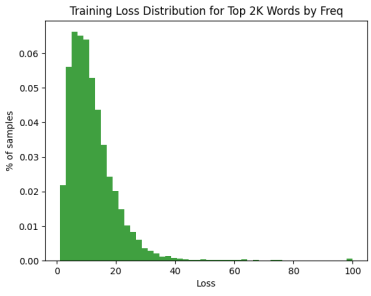
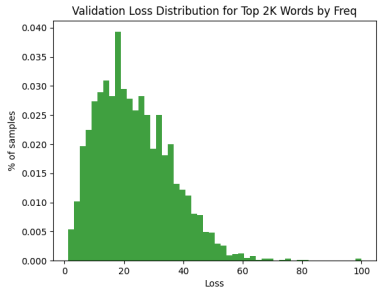
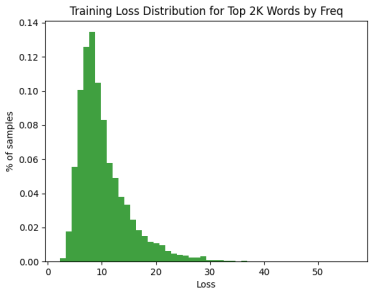
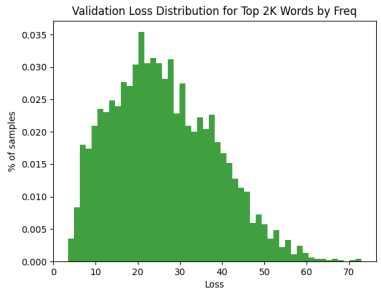
	Train	Validation
Linear		
LSTM		

Figure 6: Distribution of loss among samples in the top 2000 words

	Training		Validation	
Model	Max Loss	% of samples with loss > 60	Max Loss	% of samples with loss > 60
Linear	294.26	0.55%	133.81	0.61%
LSTM	56.405	0%	73.038	0.52%

Table 7: Outlier statistics for top 2000 words